Graph-Walk-based Selective Regression Testing of Web Applications Created with Google Web Toolkit

Matthias Hirzel
Wilhelm-Schickard-Institut
University of Tübingen
hirzel@informatik.uni-tuebingen.de

Herbert Klaeren
Wilhelm-Schickard-Institut
University of Tübingen
klaeren@informatik.uni-tuebingen.de

Abstract

Modern web applications are usually based on JavaScript. Due to its loosely typed, dynamic nature, test execution is time expensive and costly. Techniques for regression testing and fault-localization as well as frameworks like the Google Web Toolkit (GWT) ease the development and testing process, but still require approaches to reduce the testing effort. In this paper, we investigate the efficiency of a specialized, graph-walk based selective regression testing technique that aims to detect code changes on the client side in order to determine a reduced set of web tests. To do this, we analyze web applications created with GWT on different precision levels and with varying lookaheads. We examine how these parameters affect the localization of client-side code changes, run time, memory consumption and the number of web tests selected for re-execution. In addition, we propose a dynamic heuristics which targets an analysis that is as exact as possible while reducing memory consumption. The results are partially applicable on non-GWT applications. In the context of web applications,we see that the efficiency relies to a great degree on both the structure of the application and the code modifications, which is why we propose further measures tailored to the results of our approach.

1 Introduction

Today's web applications are not inferior to desktop applications with respect to functionality. Especially the client side does no longer only display contents to the user but performs sophisticated tasks. The power of these applications is often due to JavaScript, which offers varied possibilities to manipulate contents dynamically and asynchronously using AJAX. But due to this dynamically and loosely typed semantics as well as the prototype-based inheritance in JavaScript, code is more error-prone and hard to test [RT01, ERKFI05, HT09, AS12]. For this reason, several techniques have been proposed to support the fault-localization and testing process [MvDL12, OLPM15] or regression testing [RMD10, MM12]. A summary of further approaches can be found in a recent review of Doğan et al. [DBCG14]. However, the available techniques still require to execute whole test suites. In particular, testing the client-side with web tests might be very time consuming (several hours). Running web tests is the standard to do integration testing of web applications. They simulate the actions of a real user and interact with the software. But often, the results are not available before the next day.

Copyright © 2016 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors.

Frameworks like *Google Web Toolkit* (*GWT*) [Goo13b] avoid the problem of dynamic typing in JavaScripts by using Java as a strongly typed language with a mature debugging system. Here, both server- and client-side code is written in Java. While code on the server-side is compiled into Java bytecode as usual, the code on the client-side is transformed into JavaScript using a *Java-to-JavaScript compiler* (*GWT compiler*) [Goo12] which uses directly the Java source code to perform its task. Bytecode is never considered by the compiler [Cha08, Cha09a]. Code that is shared among server and client will be compiled in either version. As a result, this eases the programming, debugging and testing process [Goo14b, Goo14a, Cha09b]. But again, this does not reduce the effort for end-to-end tests.

In order to provide fast feedback on test cases affected by code changes, techniques like *test suite minimization*, *test case prioritization* and *selective regression testing* (*SRT*; e.g. [CRV94], [RH96]) are adequate. A review of existing approaches [YH12] within these three categories has been published by Yoo et al. The authors remark that SRT-techniques have been investigated extensively due to their definition of a *safe* test selection [RH94] that detects the same test failures as the re-execution of the whole test suite would do (*retest-all*). Here, especially graph-walk technique is the most prevalent approach. In [RH96], Rothermel et al. compare various SRT-techniques regarding inclusiveness, precision, efficiency, and generality and conclude that graph-walk approaches are more precise, but might have higher analysis costs. In another review [ERS10], Engström et al. examine SRT-techniques in terms of the cost and fault detection effectiveness.

Up to now, selective regression testing has only been applied to web tests by a few in order to reduce the test effort. Especially graph-walk based techniques are merely applied to speeding up the test execution in web applications [XXC+03, TIM08, KG12, AMBJ14]. This is also reflected in the review of existing techniques for web application testing of Doğan et al. [DBCG14]. To the best of our knowledge, we are the first who apply a SRT-technique based on a control flow graph (CFG) on web applications. Our technique focuses on the client-side code and localizes code changes in order to select all the web tests affected by these changes. We apply our approach on applications created with GWT to exploit the advantages of a strongly typed language. In this particular environment, it is important to note that none of the existing techniques are directly usable any more as they rely on a homogeneous development and testing environment. That is, the tests and the program are written in the same programming language and are both executed in the same development environment. In GWT, this is not the case. When analyzing the JavaScript-based application, existing fault-localization techniques cannot localize faults in the Java source code. Mapping errors in the target language back to locations in the source language is difficult due to code obfuscation and optimization. Conversely, changes in the Java code cannot be used directly to select web tests.

In our previous work ([Hir14]), we have shown the feasibility and the functional principle of our approach. It is based on *code identifiers* (*CIDs*) which are assigned to every Java code entity (methods, statements or even expressions) in the initial program version P. While transferring the Java code into JavaScript, they are injected as instrumentations into the JavaScript code. When executing the web test suite, the application sends the traversed CIDs to a logging server which inserts them into a database. Hence, the database contains traces for each web test. For determining code changes, our approach relies on an adaptation of an existing safe graph-walk technique that is based on an extended version of Harrold et al.s CFG [HJL+01]. We call it the Extended Java Interclass Graph (EJIG). Our technique creates an EJIG for both the initial version P and the new version P' and does a pairwise comparison of their nodes. (Nodes represent code entities like methods or statements, edges between the nodes represent the control flow.) Code changes can be recognized in differing edges or node labels. By matching the CIDs of the changed nodes with the CIDs in the database, we can determine the test cases affected by the code changes. The analysis and test selection is static and does not require P' to be re-executed.

In this paper, we evaluate the efficiency of our SRT-technique in an industrial environment. Crucial for the efficiency are the instrumentation of the web application, the time required to analyze P and P' and the test selection. In detail, we make the following contributions:

- A discussion of the challenges to apply a CFG-based SRT-technique to web testing in a cost-efficient way. Our technique aims at detecting client-side code changes in order to determine a reduced set of web tests that have to be re-executed due to these code changes;
- An approach to address these challenges by using various levels of analysis, lookaheads and a database for doing fast queries and to support the developer in the bug fixing process;
- An investigation of the impact of the above mentioned parameters on run time, memory consumption, the ability to recognize code changes and the number of selected tests;

- A proposal of a heuristics that targets at finding an efficient trade-off between a low memory consumption and the ability to detect code changes as exactly as possible. The heuristics differs from other approaches reducing the overhead of regression testing [BRR01, AOH07] in such a way, that we assign each class the analysis granularity and the lookahead individually. It resembles the heuristics proposed by Orso et al. ([OSH04] but simplifies it further;
- An evaluation of the cost-efficiency of our SRT-technique compared to a traditional retest-all approach;
- An Eclipse-plugin GWTTestSelection to overcome the common nightly build and test cycle towards a fast executable and repeatable cycle of code changing, test determining, test case executing and bug/test case fixing that resembles continuous integration [Fow06]. In contrast to our first prototype, our technique is now independent from the version of the GWT-compiler in use.

The evaluation shows that our heuristics is able to localize changes at the client-side of a web application in a memory-protecting way. Lookaheads help to detect more code changes in a single analysis than other SRT-techniques. In total, the cost efficiency of our technique depends on the structure of the web application and the kind of code change. This is due to the special nature of web tests. Partially, our technique outperforms the retest-all aproach, partially, further measures are required.

2 Motivation and Challenges

Every SRT-technique has to achieve two goals: a) an exact localization of code changes in order to select only these web tests that are actually affected, and b) the benefit of the technique has to outweigh the overhead of a retest-all approach. As web tests run in a web browser, run time is limited by factors like the client-server communication or the data loading from databases. Thus, the total execution time of the web tests increases with the size of the web application and the test suite. For example, in the company that allowed us to investigate our approach, the test suite consists of 105 tests and takes 9 hours when using the common restest-all approach. Therefore, techniques that aim at reducing the test effort have potential to boost the test selection.

Basically however, the test suite can be split into several parts in order to run them in parallel and to reduce the time consumption. Nevertheless, this is accompanied by an increase of costs. On the one hand, there have to be enough powerful virtual machines available that have to be maintained and kept up to date. On the other hand, every machine requires a license of the testing platform. Especially the costs for licenses and support are considerable[1] and usually cannot be provided in a sufficiently large number to support continuous integration.

With regard to the choice of technique, a safe regression selection technique is preferable as it does not miss test cases that reveal a bug. In contrast, Orso et al. report in [OSH04] that safe selective regression testing techniques are less cost-efficient compared to unsafe techniques in particular when applied in big software systems. According to them, the reason is that the safe technique takes more time than the retest-all approach. As the article is more than ten years old and as today's computers have significantly more internal memory and power, these results seem not to be crucial any more. Instead, the efficiency of our technique might be compromised by the additional time needed for:

- instrumenting the Java code,
- executing the instrumented web application with its transmission of CIDs to the logging server and their insertion into a database for further processing,
- creation of EJIGs for the old program version P and the new program version P',
- comparing the two graphs,
- selecting tests cases by querying the database to find test cases that traverse the CID of a changed node in the EJIG.

Especially the nature of web tests can have a significant impact on the number of selected tests for re-execution. Distinct web tests do not test mostly disjoint functions as unit tests usually do, but might execute the same client-side code. Therefore, modifications in the code may affect easily many web tests, which makes a test suite reduction harder.

[1]Standard business testing tools are priced at almost 1000 € (see e.g. `http://smartbear.com/product/testcomplete/pricing/`)

3 Approach

A main factor that influences the time exposure is the precision used to perform the analysis of the Java code. Here, we can distinguish various levels of precision. For example, the code could be analyzed for code changes rather coarse-grained by comparing method declarations. A more fine-grained analysis could perform this comparison on statement or even on expression level. The precision level impacts the instrumentation of the Java code. By logging the execution of every entity (methods, statements, expressions), the level of precision has a high impact on the performance overhead introduced by instrumentation. Queries to select the tests that have to be re-executed therefore take longer. Besides, when doing a fine-grained analysis of the code, the EJIGs contain more nodes so any comparison of the two graphs potentially takes more time and additionally, it leads to increased memory consumption. However, a fine-grained analysis results in a better fault localization and therefore in a reduced test selection, which is one of our main targets. For this reason, we introduce two levels of precision for our analysis which we call *Body Declaration* and *Expression Star*. They will serve as starting point to define a heuristics for finding a trade-off.

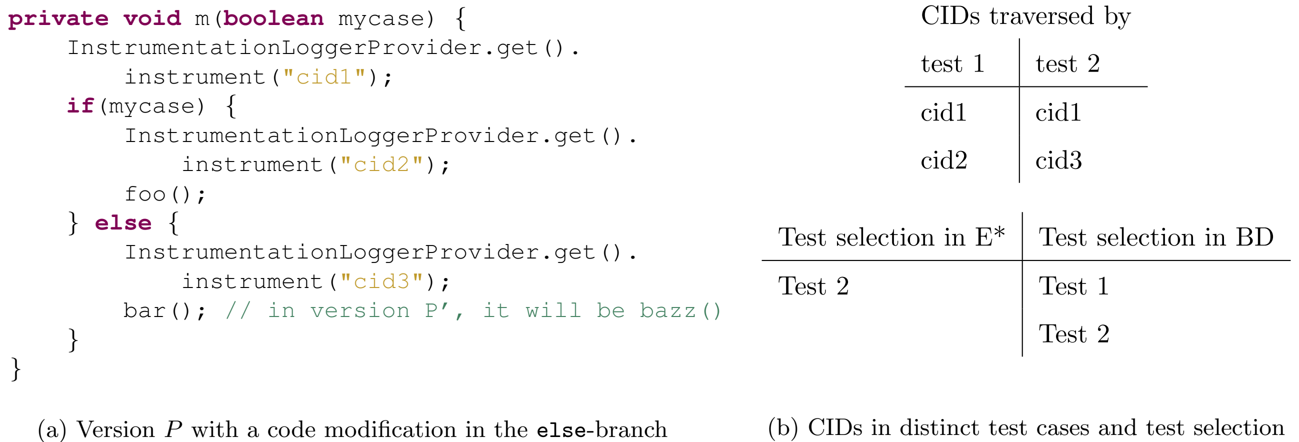
The run time of the analysis is additionally affected by the completeness of the analysis. As soon as a modification has been detected, we are able to do a safe test selection. However, there might be more code changes throughout the remaining program execution that affect other test cases. Without a continuing analysis, bugs in these code changes might not be detected before a future analysis gets started. A look-ahead enables us to do a more in-depth analysis that finds more changes. It defines an upper limit of how many nodes will be investigated to find additional changes. Details follow in section 3.4.

In this section, we describe the details of our approach to deal with the mentioned factors.

3.1 Analysis levels at various precision

We introduce the analysis precision level *Expression Star* (E^*) which calculates CIDs and generates nodes in the EJIG on expression level with a few exceptions. For example, literals have been excluded as they would increase the logging amount enormously without providing any benefit for fault localization.

In the analysis precision level *Body Declaration* (BD), nodes represent body declarations in the code such as methods, types or fields. So, this level is less precise and is not able to distinguish localized modifications. As a consequence, it risks selecting too much tests. The example code in figure 1 shows a method of the initial version P. In P', there will be a code modification in the `else`-branch (see figure 1a: `bar()` will be `bazz()`). Solely, test 2 traverses the changed code (see figure 1b). When comparing P' with the old version P, E* considers the CIDs in the case distinction and selects only test 2. In contrast, BD only considers the CID representing the method declaration (`cid1`) and will select both tests.



```
private void m(boolean mycase) {
    InstrumentationLoggerProvider.get().
        instrument("cid1");
    if(mycase) {
        InstrumentationLoggerProvider.get().
            instrument("cid2");
        foo();
    } else {
        InstrumentationLoggerProvider.get().
            instrument("cid3");
        bar(); // in version P', it will be bazz()
    }
}
```

CIDs traversed by

test 1	test 2
cid1	cid1
cid2	cid3

Test selection in E*	Test selection in BD
Test 2	Test 1
	Test 2

(a) Version P with a code modification in the `else`-branch

(b) CIDs in distinct test cases and test selection

Figure 1: Test selection in various precision levels

The EJIG created by this BD-level contains less nodes than the EJIG created by E* which leads to a reduced memory consumption. Consequently, we would expect an improvement of the run time as there are less nodes to compare. In the creation of the EJIG itself via the BD-precision level, we do not expect a significant speedup. Technically, the EJIGs are created by traversing the *abstract syntax tree* (*AST*) provided by the Eclipse *Java*

Development Tools (*JDT*)[2]. We drill down the Eclipse AST and stop creating nodes for the EJIG as soon as the current node in the Eclipse AST does not match the analysis precision level any longer. Due to the fact that method invocations are expressions, we have to continue walking through the AST even if the BD-level is selected for precision in order to model calls of methods in the control flow. Otherwise, the control flow gets interrupted.

3.2 Dynamically customizable analysis level based on a heuristics

We want to find a reasonable trade-off that considers the strengths and disadvantages of the E*- and BD-precision level in order to optimize run time performance and memory consumption but at the same time guaranteeing both a precise selection of test cases and the identification of code changes in the underlying Java code.

The key idea of a dynamically adaptable analysis level is that there might be cases (e.g. when parts of the code usually never change at all or only to a limited extend), in which a detailed analysis does not really provide more information but requires more memory and potentially loses time in preparing or inspecting code. For this reason, we propose a hybrid form of analysis in order to reduce the gap between precision and performance. In this context, it is crucial to find a competitive decider. Especially in the area of test prioritization, heuristics are frequently used to decide on which test should be selected preferably. We modify this strategy to decide which parts of the code might be investigated less thoroughly.

In a first approach, we have mediated using the change frequency of `java`-files as decider at which precision level a `CompilationUnit` in the Eclipse AST (which corresponds to a class or interface) should be analyzed. Alternatively, we have reflected on analyzing those `CompilationUnit`s on E*-precision level that have been responsible for an increased test selection in a previous analysis. However, we have detected that in our software under test (SUT), the number of changes are neither Gaussian distributed, nor do the test selection prone code changes in P correlate significantly with changes in P'. Of course, this may be different in other SUTs but obviously, the change frequency and the likelihood of `CompilationUnits` being responsible for a high test selection in the past are not suitable criteria for all kinds of applications.

Our heuristics uses a check to decide which source files have been changed. Here, a change can be an addition, modification or removal of a file in P'. (For simplicity, we refer to them as changed files.) This is done by querying the code repository before the creation of the EJIGs starts. Of course, irrelevant changes (e.g. white spaces or blank lines) are ignored. The heuristics takes the list of changed files as input and influences directly the number of nodes both in P and in P'. When traversing the ASTs of P and P', the heuristics checks for each `CompilationUnit` whether it is affected by a code change. If there is a match, the heuristics creates corresponding nodes until the E*-precision level is reached. Otherwise, only body declarations will be represented by nodes in the EJIG.

Our heuristics is similar to the one proposed by Orso et Al. [OSH04]. In favor of a quick and easy computation, it is less precise as we do not analyze any hierarchical, aggregation or use relationships among classes. Orso et al. argue that a heuristics depending solely on modifications in the source files fails to identify and treat declarative changes in the context of inheritance correctly. Besides, they claim that such a test selection is not safe. This is a valid remark in the context of choosing whether a compilation unit should be analyzed at all. In our case however, this claim does not apply since we only use the heuristics for adapting granularity between BD and E*. In our heuristics, this argumentation does not apply as we do not restrict the amount of code. We just represent the code in a `CompilationUnit` more coarse-grained, so our approach is still safe. To illustrate this, we use the relevant part of the same example as Orso et al. employ in their argumentation and extend it by an additional class `HyperA`:

In the example in figure 2, `A.foo()` has been added in P' and that is why `SuperA.foo()` is not traversed any more. In the EJIG, this modification is represented by a changed call edge pointing to `A.foo()`. Our heuristics will consider `A` as changed and it will do a fine-grained analysis on any code within `A`. In terms of our heuristics, this is correct and meets our expectations.

Now imagine that in P', `A` extends from an already existing, unchanged class `HyperA` instead of overriding `SuperA.foo()`. In this case, there is a declarative change in the inheritance chain (`A extends HyperA` instead of `A extends SuperA`). Our heuristics will again analyze `A` in detail, but actually, the code in `A` did not change. A test case will execute the `dummy()`-method of `HyperA`. Nevertheless, this does not affect the safety of our approach as we just represent the `A` more coarse-grained. (In the example, even this is not a problem as `HyperA.dummy()` did not change.) That is, on the one hand, we run the risk that that our heuristics selects additional test cases

[2] `http://www.eclipse.org/jdt/`

```java
public class SuperA {
    int i=0;
    public void foo () {
        System.out.println(i);
    }
}
public class A extends SuperA {
    public void dummy() {
        i--;
        System.out.println(-i);
    }



}

public class HyperA {
    public void dummy() {
        // do something
    }
}
```

(a) Version P

```java
public class SuperA {
    int i=0;
    public void foo () {
        System.out.println(i) ;
    }
}
public class A extends SuperA {
    public void dummy() {
        i--;
        System.out.println(-i);
    }
    public void foo () {
        System.out.println(i+1);
    }
}

public class HyperA {
    public void dummy() {
        // do something
    }
}
```

(b) Version P'

Figure 2: Declarative code change

as described in the previous subsection. But on the other hand, the heuristics is easy to compute and does not require much additional time. Moreover, we analyze the code to a high probability at a high precision level when it is necessary and additionally reduce the memory consumption.

3.3 Trace collection

To record which Java code is executed by the different web tests, we introduce CIDs which represent single Java code entities and inject them as instrumentations into the JavaScript Code. To this end, in previous work [Hir14] we extended the GWT compiler to also add instrumentation when translating from Java to JavaScript. This was very convenient as the compiler took care of the CIDs not getting separated from the code entity they identify. However, we depended on the compiler-version. In this paper we pre-process the Java code to insert instrumentations as additional statements in the Java source code before compiling. In order to avoid polluting the local working copy with instrumentation code, we use a copy of the source code of P. It will be compiled into JavaScript and serves as initial version for collecting the test traces.

Inserting instrumentations in the Java source code involves the danger that the connection between code entity and injected CID gets broken during the GWT-compilation and -optimization process. Our workaround takes advantage of GWT's JavaScript Native Interface (JSNI) [Goo13a]. Here, it is possible to define JavaScript code within the regular Java code. This allows us to write Java instrumentation code consisting of simple method calls to native JavaScript code. We will refer to these method calls as *logging calls* (LC). Each LC passes at least one CID as parameter to the native JavaScript code which in turn sends the CID to our logging server whose task is to persist the CIDs in a database. As the GWT-compiler has to maintain the semantics of the Java code, it will not change the order of method calls during compilation in general and therefore, the position of LCs will not change either. Consequently, we establish the semantic connection between an injected CID and the code entity it represents via its syntactical position. The general rule of thumb is to insert them as statements right in front of the Java code entity the CID belongs to. In some cases, though, the rule of thumb is not applicable due to syntactical restrictions. When considering fields or classes, the LCs have to be inserted in the constructor or initializer. Instrumentation code representing methods, initializers or blocks are added right after the opening curly brackets. So, even if an exception in thrown by the web application, it is ensured that the corresponding instrumentation code is executed before.

The total costs C_{traces} for collecting test traces depend on the costs for the code instrumentation C_{instr} plus

the costs C_{log} for traversing and sending the CIDs to the logging server. Both parts increase linear with the number of instrumentations. We use the number of nodes representing a code entity in E* and BD, respectively, to compare the total costs for collecting traces on our precision levels. Because of code adaptations and optimization preformed by the GWT-compiler, C_{instr} will not be of the same size as C_{log}. However, since the GWT-compiler has to maintain the code semantics, all relevant CIDs will be contained in the JavaScript code. Therefore, we approximate C_{traces} as:

$$C_{traces} = C_{instr} + C_{log} \approx 2 \cdot C_{instr}$$

In real programs, the set of nodes represented by BD is smaller than the set of nodes represented by E*. The BD-precision level is therefore cheaper in terms of test tracing.

The costs for our heuristics basically depend on the number of `CompilationUnits` represented on E*-level and are bounded by the costs for collecting traces on BD-level and E*-level, respectively. So, theoretically, it is $C_{BD_{traces}} \leq C_{H_{traces}} \leq C_{E*_{traces}}$. But as we do not know in advance which `CompilationUnits` will change in P', the entire trace collection for P has to be done on E*-precision level. Hence, the costs $C_{H_{traces}}$ and $C_{E*_{traces}}$ are the same.

3.4 Recognizing more code changes with lookaheads

Code modifications are often not local to one particular position in code. For instance a refactoring of changing an instance variable name may cause multiple method bodies to change. However, most SRT-techniques [HJL+01, TIM08, AMBJ14] compare the program versions P and P' only up to the first occurrence of a code modification. Other changes that occur in the CFG later are not examined by these techniques any more. Their identification requires other techniques like Change Impact Analysis or manual inspection. Both possibilities of course require additional time. Missed impacts of code changes emerge not before the SRT-technique is re-executed.

In order to reduce this overhead to find additional modifications, in [Hir14] we have adopted an approach presented by Apiwattanapong et al. in [AOH07] that uses lookaheads to detect more changes. In contrast to Apiwattanapong et al., we employ a two-staged algorithm which is applied directly on the different nodes of the EJIG. The algorithm uses as input the last matching nodes in P and P'. In a Parallel-Search, we try to find whether a successor node just has been modified in P' (see node n_a in figure 3) and whether there is a common node in P and in P' (n_1) from where the program execution coincides again (n_3). Otherwise, we use a Breadth-First Search to determine whether nodes have been added in P or in P' (in figure 3, the nodes n_a and n_d have been added). If one of the two algorithm succeeds, we are able to determine the kind of code change (added, modified or removed). The comparison of P and P' continues normally to find additional changes that cannot be found by the standard approaches cited above. Otherwise, the algorithm continues to search for a common node in P and in P' by investigating the next successor nodes. This procedure continues until a the maximum number of successor nodes - defined by a lookahead parameter - has been reached.

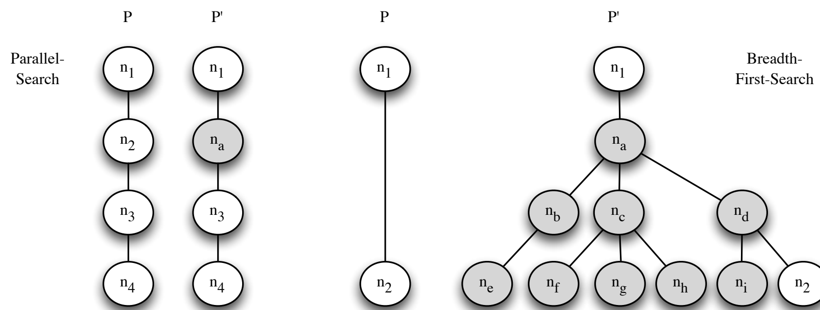


Figure 3: Parallel-Search and Breadth-First Search

Our algorithm implies a longer run time compared to standard approaches. Especially a big lookahead leads to an increasing complexity. In particular, this affects the BD-precision levels because a method has usually many possible successor nodes as there are various call to other methods. In figure 3, the outgoing edges from the grey node could be method calls. In our evaluation, we therefore use various lookaheads to investigate their impact on memory consumption and to find a useful configuration.

4 Tool Implementation

We have implemented our SRT-technique as Eclipse plug-in GWTTestSelection to support an easy and quick usage in the daily development process. It is available for download on GitHub[3].

GWTTestSelection consists of several modules. One of them performs the Java source code instrumentation. To do this, the Eclipse JDT is used to parse the Java code and to insert CIDs as instrumentations into the Java code. Another module implements the functionality for the built-in logging server. It establishes a connection to the web application via the WebSocket protocol. It can be started/stopped manually in the plugin-in or by calling a script. While running the web tests, the logging server buffers the CIDs received by the instrumented web application and writes them to a database[4]. Our tool is completely independent from any tools (e.g. Selenium [Sel14] or TestComplete [Sof15]) suitable to create web tests.

Finally, the tasks of the main modules comprise the calculation of code changes according to the desired analysis granularity and the test selection. Via the settings menu, the user can choose between two static precision levels and our dynamic heuristics. All analyses can be combined with arbitrary settings for lookaheads.

5 Evaluation

In order to assess our solutions to provide an efficient selective regression testing technique for web applications, we discuss our approach in terms of five research questions:

RQ1 To which extent will a fine-grained source code instrumentation and analysis take more time compared to a coarse-grained one? What does it mean for memory consumption?

RQ2 In which sense will looakheads affect the analysis runtime and the detection of code changes?

RQ3 Can a dynamically adaptable, heuristics based analysis level outperform a static, user-predefined analysis level?

RQ4 How many tests are selected during a detailed analysis and how much will the results differ from a retest-all approach?

RQ5 Is our technique cost-effective compared to a retest-all approach?

5.1 Software under evaluation

Our study consists of two web applications, namely Hupa [Hup12] and Meisterplan [itd15]. To evaluate these two applications, we have considered 13 pairs of versions of Hupa and 21 pairs of Meisterplan. Each of these pairs have been evaluated with various settings. In total, we have conducted 272 analyses.

Hupa is a mid-sized open source GWT-based mail client that provides all the basic functionalities of modern mail clients. This includes receiving, displaying, sending and organizing mails. For retrieving mails, Hupa uses the IMAP protocol. We checked out the source code from the public repository starting with the revision number 1577827, consisting of approximately 40.000 non-empty lines of code in 484 classes and interfaces.

In order to assess our approach thoroughly, we have chosen an industrial application as second experimental object. Meisterplan is a highly dynamic and interactive resource and project portfolio planning software for executives. Projects are visualized as Gantt diagrams. To each project, data like the number of employees, their hourly rates and their spent time may be assigned. Meisterplan accumulates allocation data from the different projects and creates histograms. Additionally, the existing capacity is intersected with the allocation data. This way, bottlenecks in capacity become visible. It enables the user to optimize the resource planing by either delaying a project or redistributing resources. Changes in capacity, project prioritization or strategy can be simulated by drag and drop. Dependencies between projects are visualized with the aid of arrows. To enhance project and cost analyses, views and filters are provided.

The source code consists of approximately 170.000 non-empty lines of code (without imports) in roughly 2300 classes and interfaces. The test suite comprises 105 web tests. The software is built and deployed using Maven. This process and the entire testing is part of continuous integration using Jenkins[5]. All web tests are created with the aid of TestComplete [Sof15], an automated testing platform.

[3] `https://github.com/MH42/srt-for-web-apps`

[4] The database is not built-in and has to be setup by the user.

[5] `https://jenkins-ci.org/`

5.2 Experimental setup

Ancient revisions of Hupa contain a large number of changes. This is contrary to the usual way of doing small increments that are regression tested afterwards. Besides, there may be many merge conflicts. The choice of our start revision respects these obstacles. In total, we have selected 6 revisions of the Hupa repository including the most recent one to do our evaluation. In order to get more reliable data, we have asked a college to do error seeding. This way, 4 additional versions have been created. (As they do not compile any more, we use them for localizing faults.) Another four versions have been implemented by ourselves. In order to guarantee real conditions, we have extracted some changes from ancient Hupa revisions. All additional versions are available on GitHub[6].

Our Hupa web test suite comprises 32 web tests created with Selenium. Unfortunately, the developers of the tool do not provide any own web tests. For this reason, we have asked another college to create web tests. He has never seen Hupa or its source code so far. Again, we have created some additional ones. The test suite is also available on GitHub.

The developers of Meisterplan maintain an own web test suite. We have selected revisions used for the nightly retest-all approach and the corresponding web tests to do our evaluation. As we would like to integrate our approach in the continuous integration process, we have additionally selected revisions committed during the day to investigate how our approach performs in this situation.

The evaluation of Meisterplan has been performed on a Intel Xenon 3.2 GHz with 8GB RAM. The Eclipse settings allowed a max. heap size of 6 GB. For Hupa, we have used an Intel Code i5 2.4 GHz with 8 GB RAM.

5.3 Results

In the following subsections, we will discuss the results of our evaluation in terms of our research questions.

5.3.1 Time and memory consumption

Figures 4 and 5 show the run time needed to analyze the version pairs of Hupa and Meisterplan, respectively. As we have 34 pairs in total, we use box plots to report on the results. The horizontal axis represents the parameter settings. We have used the same for both applications. The precision level E* has been investigated with lookahead values 20, 10, 5 and 1. The BD precision level has been tested with lookahead values 5 and 1. Apart from this, we have considered 2 heuristics. The first one (E*-BD L5-5) tries to balance the lookaheads in the E* and the BD level and sets both to 5. The second one (E*-BD L10-1) considers the extremes and defines lookahead = 10 for E* and lookahead = 1 for BD.

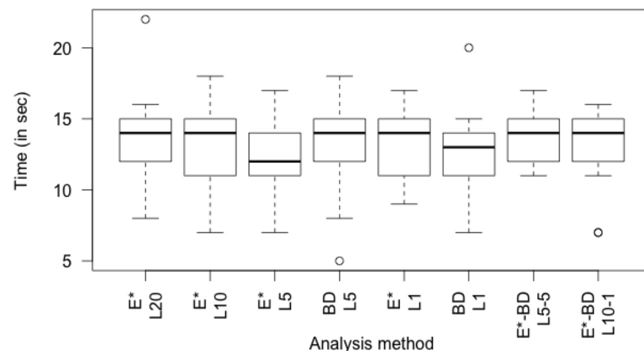


Figure 4: Test duration Hupa

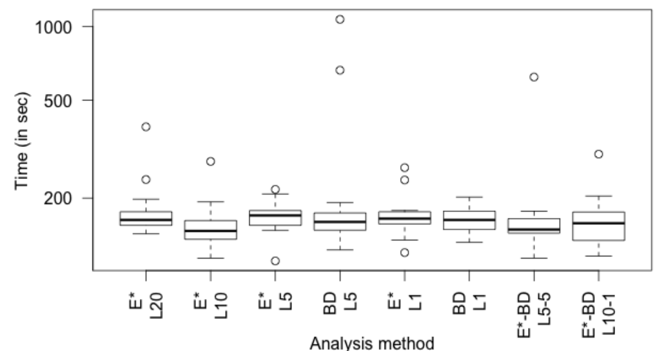


Figure 5: Test duration Meisterplan

The horizontal line of the box plots represent the median. The boxes below/above the median contain the lower/upper the 25% of the values. The vertical lines (whiskers) at both ends of the box show the other values ignoring possible outliers. Our results show that there are only little differences in run times. Due to its medium size, Hupa shows the same median in almost all settings. In the large application, we have learned in a variance analysis that there are significant differences ($p = 5\%, X^2 = 56.06, df = 27$) when setting the outliers to the median. However, the subsequent t-test has shown that only E*-BD L5-5 shows a significant better run time

[6] https://github.com/MH42/srt-for-web-apps

compared to E* L5. This is somewhat contradictory to our expectations. When there is enough internal memory, a fine-grained analysis does not provide any disadvantages. However, when analyzing Meisterplan on E*-level, each EJIG requires 10 times more nodes than at the BD-level. Considering Hupa, there are still 3 times more nodes in the E*-level.

As far as RQ1 is concerend, a more detailed analysis is no problem as long as there is enough memory available. The similar run times are a result of the necessity to traverse the Eclipse AST even on BD-level as described in section 3.2. Besides, the complexity is higher when searching for successor nodes in a BD-level.

5.3.2 Lookaheads

Figure 6 shows the number of code modifications in Hupa for the E* precision level. Our findings indicate that the number of detected code modifications rises with the lookahead. This observation is similar to the one made by Apiwattanapong et al. [AOH07]. However we have noticed that the lookahead should not be selected to large as it might happen that our algorithm may detect some nodes which match accidentally. This is especially true for lookaheads used in an anylsis on BD-level. We have observed that during our experiments to find a suitable lookahead. Additionally, we have observed two outliers (see figure 5). Repeating the same analyses have confirmed that it did not happen by accident. Consequently, memory was at a critical point. Thus, with respect to RQ2, a higher lookahead can be beneficial.

	E* L1	E* L5	E* L10	E* L20
v2v1	28	28	28	28
v3v2	1	1	1	1
v4v3	16	20	20	21
v5v4	6	12	15	15
v6v5	9	9	9	9
v7v2	11	11	11	11
v8v2	16	16	22	25
v9v2	10	11	11	11
v10v2	1	3	3	3
v11v6	22	25	25	25
v12v6	16	17	17	20
v13v6	46	85	85	88
v14v6	10	10	10	10

Figure 6: Code modifications

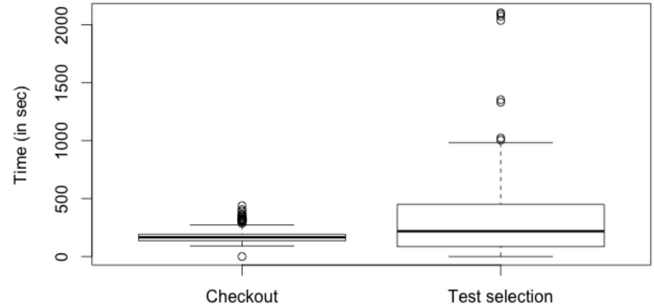


Figure 7: Checkout and Test selection time Hupa

5.3.3 Benefits of a dynamic Heuristics

The main advantage of our heuristics with respect to RQ3 is the reduction of nodes during the analysis. Considering Meisterplan, it is theoretically necessary to keep up to 120 000 nodes per EJIG in the internal memory. With our heuristics, this amount can be reduced dramatically (e.g. Meisterplan: factor of 10). The only disadvantage is that a file could be erroneously analyzed in a more coarse-grained way. The lookahead settings for both of our heuristics are a direct result of the experiences we have made with our static precision levels. Our balanced heuristics performs a bit better than the other one. Besides, it is significant better than the static variant E* L5. So, the settings E* L10, BD* L5 and our balanced heuristics E*-BD L5-5 could be the settings of choice.

We can see that our heuristics provide no run time improvement compared to static analysis levels. Looking at the test selection (see discussion in 5.3.4), especially H L5-5 unifies in many cases the best results (see e.g. v7v5 in fig. 9) and offers a tradeoff in memory consumption. So, a heuristics outperforms a static analysis level.

5.3.4 Test selection

Figures 8 and 9 show how many tests are selected for re-execution. Each row represents a pair of versions with 8 possible settings. The value in the last column indicates how many tests actually failed during the execution. (In table 8, four versions have been used for fault localization only as mentioned above.) Our findings show that the BD-level sometimes select more tests as the E*-level as expected (see e.g. Hupa, v3v2, BD L5 and BD L1 or Meisterplan, v13, v12, BD l5 and BD L1). However, we have observed one case in which even the E* L1 has selected more test cases than BD (see Hupa, v8v2) due to a false positive. The same is true for v7v5, E* L5 in the Meisterplan results.

	E* L10	E* L5	B L5	E* L1	B L1	E* L20	H L5-5	H 10-1	Exp
v2v1	0%	0%	0%	0%	0%	0%	0%	0%	0%
v3v2	0%	0%	47%	0%	47%	0%	0%	0%	0%
v4v3	97%	97%	97%	97%	97%	97%	97%	97%	6%
v5v4	97%	97%	97%	97%	97%	97%	97%	59%	0%
v6v5	9%	9%	13%	9%	13%	9%	9%	9%	0%
v7v2	0%	0%	0%	0%	0%	0%	0%	0%	9%
v8v2	97%	97%	47%	97%	47%	97%	97%	97%	3%
v9v2	44%	44%	44%	44%	44%	44%	44%	44%	0%
v10v2	97%	97%	97%	97%	97%	97%	97%	97%	0%
v11v6	9%	9%	97%	9%	97%	9%	9%	9%	fault loc.
v12v6	97%	97%	97%	97%	97%	97%	97%	97%	fault loc.
v13v6	97%	97%	97%	97%	97%	97%	97%	97%	fault loc.
v14v6	97%	97%	97%	97%	97%	97%	97%	97%	fault loc.

Figure 8: Test selection Hupa

	E* L10	E* L5	B L5	E* L1	B L1	E* L20	H L5-5	H 10-1	Exp
v2v1	100%	100%	100%	100%	100%	100%	100%	100%	21%
v3v2	100%	100%	100%	100%	100%	100%	100%	100%	4%
v4v3	6%	6%	6%	6%	6%	6%	10%	6%	3%
v5v4	100%	100%	100%	100%	100%	100%	100%	100%	0%
v6v5	62%	0%	0%	0%	0%	0%	0%	9%	0%
v7v5	9%	12%	9%	9%	9%	29%	9%	9%	1%
v8v3	100%	100%	100%	100%	100%	100%	100%	100%	6%
v9v8	99%	19%	19%	25%	19%	28%	19%	99%	0%
v10v8	99%	99%	99%	19%	99%	99%	99%	99%	0%
v11v10	100%	100%	100%	55%	55%	100%	100%	60%	0%
v12v10	100%	100%	100%	100%	100%	100%	100%	100%	0%
v13v12	51%	51%	100%	51%	100%	100%	100%	100%	0%
v14v8	99%	99%	99%	99%	99%	99%	99%	99%	0%
v15v14	100%	100%	0%	100%	0%	100%	0%	0%	1%
v16v15	100%	100%	100%	99%	100%	99%	99%	99%	0%
v17v15	100%	100%	100%	100%	100%	100%	100%	0%	1%
v18v17	100%	100%	96%	100%	96%	100%	96%	100%	0%
v19v18	100%	100%	100%	100%	100%	100%	100%	100%	0%
v20v19	100%	100%	100%	100%	100%	100%	100%	100%	0%
v21v20	97%	97%	97%	100%	97%	97%	97%	100%	0%
v22v21	99%	99%	99%	99%	99%	99%	99%	99%	0%

Figure 9: Test selection Meisterplan

There are cases, in which only a small subset of the test suite is selected for re-execution. This is especially true for the revisions v4v3, v6v5 and v7v5 which have been committed by the developers during the day. Hupa also has versions which do not need to be retested with all of the web tests in the test suite.

Nevertheless, there are also many cases, in which almost all tests are selected for re-execution. Here, it becomes evident that web tests are more complex than unit tests due to side-effects on other code. In many cases, we have observed that only a few modifications are responsible for selecting almost all web tests for re-execution. Conversely, this means that each web test will execute the modified code. Therefore, a solution might be to select only one of these test cases to get a quick feedback whether this special test execution already results in an error. But of course, the validity of such a simplification is weak and the execution of an omitted test might reveal a fault due to a different state of the application. And most importantly, the approach is not save any longer when reducing the test result artificially.

In the end, our technique decreases the testing effort by up to 100% compared to a retest-all. But in many cases there is even no improvement. The crucial factors are the structure of the application and the code modifications.

5.3.5 Efficiency

To find a response to RQ5, we have to look at the overall costs which depend on the application itself. Meisterplan takes 4:30 min to instrument the code on E*-level. Executing the web tests and logging the CIDs has an overhead of 90 min. That is, on average, each test takes 51 sec longer due to instrumentation overhead. Comparing the current version with a previous one requires a checkout. Figure 7 shows the resulting boxplot for Meisterplan. According to this, the median is 166 seconds. The analysis of P and P' with our heuristics E*-BD L5-5 takes additional 149 sec (which is 2 sec slower as E* L10). Finally, the test selection requires 218 sec (see figure 7). In total, when applying our approach to Meisterplan, the extra effort is 13:23 min for doing the analysis plus 90 min for logging the CIDs. As the retest-all approach takes 9 hours, a single test takes on average 5:09 min. In order to be efficient, our approach should decrease the amount of tests selected for re-execution by 21 tests (20%). Consequently, our approach is efficient for the versions v4v3, v6v5, v7v5 and v14v8.

Considering Hupa, we have to deal with the following values. Instrumentation: 40 seconds; 6 additional minutes to finish testing and logging; checkout: 40 sec; median for analyzing the code: 14 sec; test selection time: < 2 sec. In total, our approach should decrease the test suite by 27 tests (84%). Here, it becomes apparent that especially large applications with a big test suite gain from our approach. Using our test suite, Hupa does not require loading any settings or databases. For this reason, the usual test execution can proceed immediately whereas the instrumented execution has to make sure that the CIDs have been memorized. This delay leads to the necessary test suite reduction.

As described in section 3.3, the BD-level gains a lot from a lower instrumentation and logging time. In Hupa, $C_{BD_{traces}} \approx 1/3 \cdot C_{E*_{traces}}$, in Meisterplan even 1/10. These costs affect the most expensive part of our approach: the logging process. Consequently, this using the BD-level might improve the efficiency. However, an analysis on BD-level tends to select more tests as shown in our evaluation. The benefit might therefore be case dependent.

Regarding RQ5, our technique is efficient even in medium sized applications with small test suites. In big systems with large test suites, our technique is efficient in particular as long as changes do not affect all tests.

5.4 Discussion and Threats to Validity

Our evaluation shows that our technique is able to reduce the testing effort. However, the approach has to be refined in order to deal even with those situations when the test selection is not able to reduce the test suite. At the beginning of our experiment, some of the results have even been worse. It turned out that this was due to modifications in fields. As soon as a web test traversed the constructor, the CID representing the field has been executed. Now, we consider field modifications only if their value is really used in methods executed by a test.

Technically, the CIDs are injected into the JavaScript code via the GWT compiler. Here, we have to rely on the fact that the CIDs and the code they are representing will not get separated by the compiler. However, the compiler may not change the order of method calls. So this is not an issue.

6 Related Work

Research has not payed much attention to selective regression testing of web applications so far. Hence, there exist only a few examples apart from our previous work [Hir14].

Tarhini et al. [TIM08] propose event dependency graphs to represent an old and a new version of a standard web application, consisting of pages and dependencies (events, links and others). They determine changed as well as possibly affected elements and select tests that are touched by the changes.

Another regression technique has been presented by Xu et al. It is based on slicing [XXC^{+}03] and tries to analyse which pages or elements in a page of a web application have been added, modified or removed.

The last two approaches do not consider the use of JavaScript or AJAX in detail.

Mesbah et al. in [MvD09, MvDL12] introduce a tool called CRAWLJAX. It defines constraints (assertions) defined for the user interface which allows to detect faults. Based on a state flow graph, paths through the web application are calculated which in turn are transformed into JUnit test cases. Roest et al. [RMD10] apply this approach to perform regression testing of AJAX-based web applications. Ocariza et al. [OLPM15] present an improvement that localizes faults in a more precise way. This approach is powerful, but always works with JavaScript code. This is only helpful if the code has been written by a developer. With regard to automatically, highly optimized code, these approaches are not applicable to GWT-based web applications. Besides, the approaches is unable to select a subset of web tests that have to be re-executed due to code changes.

The technique of Mirshokraie et al. [MM12] for regression testing JavaScript code is based on invariants. They run a web application many times with different variables and log all the execution traces. Based on the data collected, they try to derive invariants as runtime assertions and inject them in the next program versions. It is a dynamic analysis technique, whereas our analysis is (apart from the initial test execution) static. In our approach, there is no need to re-execute the application to detect changes that may cause failures. Beyond that, [MM12] have to re-execute the entire web application to ascertain that all the assertions hold. A selective test selection comparable to ours is not available.

Asadullah et al. [AMBJ14] also investigate a variant of the SRT-technique published by Harrold et al. in the context of web applications. They consider frameworks like JavaServer Faces. However, they do not address the problem of reducing the execution time of web tests. Instead, they focus on the server side.

From a technical point of view, the present work is related to other work in several areas. Instrumenting code is a well-known technique usually applied to determine code coverage. In [LMOD13], Nan et al. have checked whether the results of a bytecode instrumentation differ from those obtained in a source code instrumentation. Besides, they have discovered that there exist only a few tools doing source code instrumentation. Due to GWT, only these tools are relevant. However, our instrumentation differs completely from the usual one as our target is not to determine whether a Java code element is covered by a branch of the CFG. Web tests only execute JavaScript code, so client-sided Java code is never executed directly. Our instrumentation is highly specialized in such a way that the GWT compiler is able to maintain the binding to the underlying source code when transferring Java code into JavaScript. This is very important in mapping Java code modifications to the branches executed by a web test.

Apiwattanapong et al. [AOH07] extend an existing approach based on Hammocks. In an initial analysis, they compare two program versions in several iterations, starting on class and interface level. Based on these data, they compare methods and afterwards single nodes of a special CFG. However, they do not analyse the code dynamically. Besides, target applications are common desktop applications.

Gligoric et al. [GEM15] analyze code dynamically, but use class or method granularity which is less precise. Besides, they do not consider web applications.

A different technique focusing on the reduction of analysis overhead has been presented by Orso et al. [OSH04]. They try to investigate the code at two stages and perform a high-level analysis narrowing down the choice of code that should be analysed in detail in a subsequent code investigation. Although we share this idea in reducing the analysis overhead, we use a simplified approach and demonstrate that our approach is not insecure.

The approach of Bible et al. [BRR01] is close to our suggested technique. They report on advantages and problems of a coarse-grained safe regression testing technique and another safe technique that analyses the code on statement level. On this basis, they develop a prototype for a hybrid technique that tries to combine the best properties of both approaches. However, as already explained in the previous paragraph, they have no facility to adjust the analysis level as needed.

7 Conclusion and Future Work

Today's web applications rely heavily on JavaScript, but the development is difficult due to the dynamic typing of JavaScript. Frameworks like Google Web Toolkit ease coding. Nevertheless, testing the front end using web tests may take several hours to finish. To the best of our knowledge, we are the first who applied a graph-walk based SRT-technique to web applications in order to select a subset of web tests so as to reduce the time effort for execution of web tests. Our heuristics helps to reduce memory consumption during the analysis and lookaheads enable us to localize as much code changes as possible.

According to our results, especially the heuristics is a good tradeoff between precision and low memory consumption. The lookaheads help to find more code changes. With regard to efficiency, the results revealed a big dependency on the kind of code change. Our approach is efficient as long as modifications do not affect the whole web application. This is of course always valid in other contexts (e.g. desktop applications) as well, but it may happen more frequently in web applications. In our future work, we will try to find solutions to be able to reduce the test selection additionally.

Acknowledgements

We are indebted to the itdesign GmbH for allowing us to evaluate our approach on their software and to Jonathan Brachthäuser as well as Julia Trieflinger for creating web tests and seeding errors.

References

[AMBJ14] Allahbaksh Asadullah, Richa Mishra, M. Basavaraju, and Nikita Jain. A call trace based technique for regression test selection of enterprise web applications (sortea). In *Proceedings of the 7th India Software Engineering Conference*, ISEC '14, pages 22:1–22:6, New York, NY, USA, 2014. ACM.

[AOH07] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. JDiff: A differencing technique and tool for object-oriented programs. *Automated Software Engg.*, 14(1):3–36, March 2007.

[AS12] A. Arora and M. Sinha. Web Application Testing: A Review on Techniques, Tools and State of Art. *International Journal of Scientific & Engineering Research*, 3(2):1–6, February 2012.

[BRR01] John Bible, Gregg Rothermel, and David S. Rosenblum. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):149–183, April 2001.

[Cha08] Sumit Chandel. Please Don't Repeat GWT's Mistake! GoogleGroups. `https://groups.google.com/d/msg/google-appengine/QsCMpKbyOJE/HbpgorMhgYgJ`, October 2008. [Last access: 24. March 2014].

[Cha09a] Sumit Chandel. Is GWT's compiler java->javascript or java bytecode -> javascript? GoogleGroups. `https://groups.google.com/d/msg/google-web-toolkit/SIUZRZyvEPg/OaCGAfNAzzEJ`, July 2009. [Last access: 24. March 2014].

[Cha09b] Sumit Chandel. Testing methodologies using GWT. http://www.gwtproject.org/articles/testing_methodologies_using_gwt.html, March 2009. [Last access: 25. March 2014].

[CRV94] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. TestTube: A System for Selective Regression Testing. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 211–220, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[DBCG14] Serdar Doğan, Aysu Betin-Can, and Vahid Garousi. Web application testing: A systematic literature review. *Journal of Systems and Software*, 91:174 – 201, 2014.

[ERKFI05] Sebastian Elbaum, Gregg Rothermel, Srikanth Karre, and Marc Fisher II. Leveraging User-Session Data to Support Web Application Testing. *IEEE Trans. Softw. Eng.*, 31(3):187–202, March 2005.

[ERS10] Emelie Engström, Per Runeson, and Mats Skoglund. A Systematic Review on Regression Test Selection Techniques. *Inf. Softw. Technol.*, 52(1):14–30, January 2010.

[Fow06] Martin Fowler. Continuous Integration. http://martinfowler.com/articles/continuousIntegration.html, May 2006. [Last access: 15. September 2014].

[GEM15] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 211–222, New York, NY, USA, 2015. ACM.

[Goo12] Google. Understanding the GWT Compiler. https://developers.google.com/web-toolkit/doc/latest/DevGuideCompilingAndDebugging#DevGuideJavaToJavaScriptCompiler, October 2012. [Last access: 13. March 2013].

[Goo13a] Google. Coding Basics JSNI. http://www.gwtproject.org/doc/latest/DevGuideCodingBasicsJSNI.html, December 2013. [Last access: 11. December 2013].

[Goo13b] Google. Overview. http://www.gwtproject.org/overview.html, December 2013. [Last access: 03. December 2013].

[Goo14a] Google. Architecting Your App for Testing. http://www.gwtproject.org/doc/latest/DevGuideTesting.html, 2014. [Last access: 22. Aptil 2014].

[Goo14b] Google. Developing with GWT. http://www.gwtproject.org/overview.html#how, March 2014. [Last access: 25. March 2014].

[Hir14] Matthias Hirzel. Selective Regression Testing for Web Applications Created with Google Web Toolkit. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 110–121, New York, NY, USA, 2014. ACM.

[HJL+01] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for java software. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 312–326, New York, NY, USA, 2001. ACM.

[HT09] Phillip Heidegger and Peter Thiemann. Recency Types for Dynamically-Typed, Object-Based Languages . In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 2009.

[Hup12] Hupa. Overview. http://james.apache.org/hupa/index.html, June 2012. [Last access: 24. May 2014].

[itd15] itdesign. Take Your Project Portfolio Management to a New Level. https://meisterplan.com/en/features/, 2015. [Last access: 03. May 2015].

[KG12] A. Kumar and R. Goel. Event driven test case selection for regression testing web applications. In *Advances in Engineering, Science and Management (ICAESM), 2012 International Conference on*, pages 121–127, March 2012.

[LMOD13] Nan Li, Xin Meng, J. Offutt, and Lin Deng. Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (experience report). In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 380–389, Nov 2013.

[MM12] Shabnam Mirshokraie and Ali Mesbah. JSART: Javascript Assertion-based Regression Testing. In *Proceedings of the 12th International Conference on Web Engineering*, ICWE'12, pages 238–252, Berlin, Heidelberg, 2012. Springer-Verlag.

[MvD09] Ali Mesbah and Arie van Deursen. Invariant-based Automatic Testing of AJAX User Interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.

[MvDL12] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-Based Web Applications Through Dynamic Analysis of User Interface State Changes. *ACM Trans. Web*, 6(1):3:1–3:30, March 2012.

[OLPM15] Frolin S. Ocariza, Guanpeng Li, Karthik Pattabiraman, and Ali Mesbah. Automatic fault localization for client-side javascript. *Software Testing, Verification and Reliability*, pages n/a–n/a, 2015.

[OSH04] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 241–251, New York, NY, USA, 2004. ACM.

[RH94] Gregg Rothermel and Mary Jean Harrold. A framework for evaluating regression test selection techniques. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 201–210, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[RH96] Gregg Rothermel and Mary Jean Harrold. Analyzing Regression Test Selection Techniques. *IEEE Transactions on Software Engineering*, 22, 1996.

[RMD10] Danny Roest, Ali Mesbah, and Arie van Deursen. Regression Testing Ajax Applications: Coping with Dynamism. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 127–136, Washington, DC, USA, 2010. IEEE Computer Society.

[RT01] Filippo Ricca and Paolo Tonella. Analysis and Testing of Web Applications. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 25–34, Washington, DC, USA, 2001. IEEE Computer Society.

[Sel14] SeleniumHQ. Selenium-IDE. `http://docs.seleniumhq.org/docs/02_selenium_ide.jsp`, May 2014. [Last access: 31. May 2014].

[Sof15] SmartBear Software. TestComplete. `http://smartbear.com/product/testcomplete/overview/`, 2015. [Last access: 19. December 2015].

[TIM08] Abbas Tarhini, Zahi Ismail, and Nashat Mansour. Regression Testing Web Applications. *Advanced Computer Theory and Engineering, International Conference on*, 0:902–906, 2008.

[XXC+03] Lei Xu, Baowen Xu, Zhenqiang Chen, Jixiang Jiang, and Huowang Chen. Regression Testing for Web Applications Based on Slicing. In *Proceedings of the 27th Annual International Conference on Computer Software and Applications*, COMPSAC '03, pages 652–656, Washington, DC, USA, 2003. IEEE Computer Society.

[YH12] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012.