Mälardalen University Press Dissertations
No. 154

AUTOMATING REUSE IN WEB APPLICATION DEVELOPMENT

Josip Maras

2014

School of Innovation, Design and Engineering

Copyright © Josip Maras, 2014
ISBN 978-91-7485-140-3
ISSN 1651-4238
Printed by Arkitektkopia, Västerås, Sweden

Mälardalen University Press Dissertations
No. 154

AUTOMATING REUSE IN WEB APPLICATION DEVELOPMENT

Josip Maras

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap
vid Akademin för innovation, design och teknik kommer att
offentligen försvaras torsdagen den 17 april 2014, 13.15 i Gamma,
Mälardalens högskola, Högskoleplan 1, Västerås.

Fakultetsopponent: Associate Professor Martin Robillard, McGill University

Akademin för innovation, design och teknik

Abstract

Web applications are one of the fastest growing types of software systems today. Structurally, they are composed out of two parts: the server-side, used for data-access and business logic, and the client-side used as a user-interface. In recent years, thanks to fast, modern web browsers and advanced scripting techniques, developers are building complex interfaces, and the client-side is playing an increasingly important role.

From the user's perspective, the client-side offers a number of features. A feature is an abstract notion representing a distinguishable part of the system behavior. Similar features are often used in a large number of web applications, and facilitating their reuse would offer considerable benefits. However, the client-side technology stack does not offer any widely used structured reuse method, and code responsible for a feature is usually copy-pasted to the new application. Copy-paste reuse can be complex and error prone - usually it is hard to identify exactly the code responsible for a certain feature and introduce it into the new application without errors.

The primary focus of the research described in this PhD thesis is to provide methods and tools for automatizing reuse in client-side web application development. This overarching problem leads to a number of sub-problems: i) how to identify code responsible for a particular feature; ii) how to include the code that implements a feature into an already existing application without breaking neither the code of the feature nor of the application; and iii) how to automatically generate sequences of user actions that accurately capture the behavior of a feature? In order to tackle these problems we have made the following contributions: i) a client-side dependency graph that is capable of capturing dependencies that exist in client-side web applications, ii) a method capable of identifying the exact code and resources that implement a particular feature, iii) a method that can introduce code from one application into another without introducing errors, and iv) a method for generating usage scenarios that cause the manifestation of a feature. Each contribution was evaluated a suite of web applications, and the evaluations have shown that each method is capable of performing its intended purpose.

ISBN 978-91-7485-140-3
ISSN 1651-4238

To my family

Sažetak

Domena web aplikacija je jedna od najbrže rastućih i najraširenijih aplikacijskih domena danas. Web aplikacije se sastoje od dva jednako važna dijela: serverske aplikacije, koja omogućava pristup podacima i implementira poslovnu logiku te klijentske aplikacije koja služi kao korisničko sučelje. U zadnje vrijeme, zbog brzih, modernih Internet preglednika i naprednih skriptnih tehnika, razvijaju se sve složenija korisnička sučelja pa klijentska aplikacija ima sve veću ulogu.

Sa stajališta korisnika, klijentska aplikacija obično nudi niz funkcionalnosti. Slične funkcionalnosti se često koriste u više različitih web aplikacija pa bi pružanje podrške pri njihovu ponovnom korištenju moglo olakšati razvoj. Međutim, među tehnikama i tehnologijama koje se koriste za razvoj web aplikacija ne postoji široko rasprostranjena, strukturirana metoda ponovnog korištenja; kôd koji implementira određenu funkcionalnost se najčešće kopira u novu web aplikaciju. Takav način ponovnog korištenja je kompleksan i sklon pogreškama – obično je teško i identificirati kôd određene funkcionalnosti i umetnuti ga u novu aplikaciju bez uvođenja grešaka.

Glavni cilj istraživanja opisanog u ovoj disertaciji je razvoj metoda i alata za automatizaciju ponovnog korištenja pri razvoju klijentskih web aplikacija. Ovaj problem vodi do tri manja pod-problema: *i)* kako identificirati kôd koji implementira određenu funkcionalnost; *ii)* kako umetnuti kôd neke funkcionalnosti u već postojeću aplikaciju, bez uvođenja grešaka; *iii)* kako automatski generirati nizove korisničkih akcija koji pokreću funkcionalnost? Kao odgovore na te probleme, predložili smo sljedeće doprinose: *i)* graf ovisnosti klijentskih web aplikacija koji predstavlja ovisnosti koje postoje unutar klijentske web aplikacije; *ii)* metoda za identifikaciju kôda i resursa koji implementiraju određenu funkcionalnost; *iii)* metoda za umetanje kôda jedne aplikacije u drugu aplikaciju,

bez uvođenja pogreški; i *iv)* metoda za generiranje nizova korisničkih akcija koji pokreću određenu funkcionalnost aplikacije. Svaki od doprinosa je evaluiran na nizu web aplikacija.

Populärvetenskaplig sammanfattning

Webbutveckling är ett av de snabbast växande och mest utbredda mjukvaruområdena och webbapplikationer används nu i nästan varje aspekt av våra liv: på jobbet, för våra sociala kontakter, eller för e-handel. Moderna webbläsare och avancerade scriptingtekniker har gjort det möjligt för kan utvecklare att bygga interaktiva, sofistikerade och komplexa applikationer även för webben.

Ur användarens perspektiv erbjuder en webbapplikation ett antal funktioner, och liknande funktioner används ofta i en rad olika applikationer (till exempel bildvisare, avancerade webbformulär eller chattsystem). Utveckling av nya applikationer skulle vara mycket effektivare om dessa återkommande funktioner enkelt kunde återanvändas i stället för att programmeras på nytt varje gång, men sådan återanvändning är ofta svår och tidskrävande. Det är svårt att identifiera de delar av koden som ansvarar för en viss funktion, och när de väl identifieras är det svårt att lägga in dem i ett befintligt program utan att orsaka fel.

Huvudsyftet med forskningen som presenteras i den här avhandlingen är att tillhandahålla metoder och verktyg för att automatisera återanvändning vid utveckling av webbapplikationer. Utifrån detta mål har vi åstadkommit följande bidrag: *i)* en beroendegraf som representerar de kodberoenden som finns i en webbapplikation, *ii)* en metod föra att identifiera den exakta koden och resurserna som implementerar en viss funktion, *iii)* en metod som kan flytta kod från en applikation till en annan utan att introducera fel, och *iv)* ett sätt att generera användningsscenarier som täcker en funktion väl.

Abstract

Web applications are one of the fastest growing types of software systems today. Structurally, they are composed out of two parts: the server-side, used for data-access and business logic, and the client-side used as a user-interface. In recent years, thanks to fast, modern web browsers and advanced scripting techniques, developers are building complex interfaces, and the client-side is playing an increasingly important role.

From the user's perspective, the client-side offers a number of features. A feature is an abstract notion representing a distinguishable part of the system behavior. Similar features are often used in a large number of web applications, and facilitating their reuse would offer considerable benefits. However, the client-side technology stack does not offer any widely used structured reuse method, and code responsible for a feature is usually copy-pasted to the new application. Copy-paste reuse can be complex and error prone – usually it is hard to identify exactly the code responsible for a certain feature and introduce it into the new application without errors.

The primary focus of the research described in this PhD thesis is to provide methods and tools for automating reuse in client-side web application development. This overarching problem leads to a number of sub-problems: *i*) how to identify code responsible for a particular feature; *ii*) how to include the code that implements a feature into an already existing application without breaking neither the code of the feature nor of the application; and *iii*) how to automatically generate sequences of user actions that accurately capture the behavior of a feature? In order to tackle these problems we have made the following contributions: *i*) a client-side dependency graph that is capable of capturing dependencies that exist in client-side web applications, *ii*) a method capable of identifying the exact code and resources that implement a particular feature,

iii) a method that can introduce code from one application into another without introducing errors, and *iv*) a method for generating usage scenarios that cause the manifestation of a feature. Each contribution was evaluated on a suite of web applications, and the evaluations have shown that each method is capable of performing its intended purpose.

Acknowledgements

There are a number of people that have contributed to the existence of this Croato-Swedish thesis. In Croatia, I would like to thank Maja Štula and Darko Stipaničev for encouraging me to enroll in the PhD program and in Sweden, Ivica Crnković and Jan Carlson for giving me the opportunity to do a double degree with MDH.

Due to the double degree nature of my thesis I have been awarded with a somewhat greater number of supervisors than is usual, but I feel that I have gotten the best out of this arrangement: Maja, Ivica, and Jan, thank you for everything. You have manged to steer my vague, incoherent ideas and ramblings on web application reuse into something that can (hopefully) be called a PhD thesis. Maja, thank you for your guidance, both personal and professional; Ivica, thank you for keeping my thoughts on the big picture, pushing me to present ideas through publishing papers, and for putting up with my: "I just need a week more of implementing" requests; and Jan, thank you for all your proof-reads, detailed discussions and comments.

My warmest thanks go to my friends and colleagues: Ivo, Josip, Ljiljana, Marin, Marko, Maja, Petar, and Toni at FESB; and Adnan, Aida, Ana, Aneta, Cristina, Dag, Huseyin, Irfan, Juraj, Leo, Luka, Nikola, Severine, Stefan, Svetlana, Zdravko, and all others at MDH who have made the not-working parts of the day fun.

And finally, I cannot thank enough to my whole family: Jere, Josipa, two Marijas, Vitomir, and Zdenka for their love and support. A special thanks goes to Josipa, whose stern: *"Piši!"*, followed by a threatening look, has surely hastened the submission of the thesis.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Research questions . . . 5
- 1.3 Contributions . . . 6
- 1.4 Research Methodology . . . 7
- 1.5 Publications . . . 8
- 1.6 Thesis outline . . . 11

2 Background **13**
- 2.1 Web Applications . . . 13
  - 2.1.1 Client-side Web Application Primer . . . 14
- 2.2 Software Reuse . . . 17
- 2.3 Features . . . 17
  - 2.3.1 Feature Location . . . 18
- 2.4 Dynamic Analysis . . . 19
- 2.5 Automated Testing . . . 20
- 2.6 Program Slicing . . . 21
  - 2.6.1 Static Slicing . . . 21
  - 2.6.2 Dynamic Slicing . . . 23

3 The Reuse Process Overview **25**
- 3.1 Client-side Features . . . 25
- 3.2 Defining the Reuse Process . . . 26
- 3.3 The Reuse process . . . 27
  - 3.3.1 Feature Identification . . . 27
  - 3.3.2 Specifying Scenarios . . . 28
  - 3.3.3 Application Analysis . . . 29

3.3.4 Feature Integration 29
3.4 Conclusion 30

4 Client-side Dependency Graph **31**
4.1 Defining the dependency graph 31
4.1.1 Formal Graph Definition 34
4.1.2 Example 35
4.2 Graph Construction Process 37
4.3 Conclusion 45

5 Automatic Scenario Generation **47**
5.1 Overview 47
5.1.1 Terminology 49
5.2 Detailed process description 50
5.2.1 Example application 51
5.2.2 Selecting Scenarios 52
5.2.3 Scenario Execution 53
5.2.4 Extending event chains 54
5.2.5 Modifying input parameters 56
5.2.6 Filtering Scenarios 58
5.3 Evaluation 60
5.3.1 Generating scenarios for the whole page 60
5.3.2 Comparison with Artemis 61
5.3.3 Evaluating prioritization functions 62
5.3.4 Generating Feature Scenarios – a case study 64
5.4 Conclusion 66

6 Identifying Code of Individual Features **67**
6.1 Feature manifestations 67
6.2 Overview of the Identification process 69
6.2.1 Example 71
6.3 Interpretation 73
6.4 Problems with slice unions 75
6.5 Graph Marking 77
6.5.1 Marking Feature Code 77
6.5.2 Fixing Slice Union problems 81
6.6 Evaluation 82
6.6.1 Extracting Library Features 83
6.6.2 Extracting Features 87

6.6.3 Page optimization . . . 90
6.6.4 Threats to validity . . . 91
6.7 Conclusion . . . 92

7 Integrating Features **93**
7.1 Overview . . . 93
7.1.1 Goal . . . 93
7.1.2 Process Overview . . . 94
7.1.3 Running Example . . . 96
7.2 Conflict Types . . . 99
7.2.1 DOM conflicts . . . 100
7.2.2 JavaScript conflicts . . . 100
7.2.3 Resource conflicts . . . 101
7.3 Resolving Conflicts . . . 101
7.3.1 Resolving DOM conflicts . . . 102
7.3.2 Resolving JavaScript conflicts . . . 105
7.4 Merging code . . . 108
7.5 Verification . . . 111
7.6 Experiments . . . 113
7.7 Conclusion . . . 115

8 Firecrow tool **117**
8.1 Tool organization . . . 117
8.1.1 DoppelBrowser . . . 118
8.1.2 Feature Locator . . . 119
8.1.3 Scenario Generator . . . 119
8.1.4 Feature Integrator . . . 120
8.2 Conclusion . . . 121

9 Related Work **123**
9.1 Software Reuse . . . 123
9.2 Automatic Testing of Web Applications . . . 125
9.3 Feature Location . . . 126
9.4 Program Slicing . . . 127

10 Conclusion **129**
10.1 Identification of feature implementation details . . . 129
10.2 Integration of feature code . . . 130
10.3 Automatic Scenario Generation . . . 130

10.4 Future Work 131
10.4.1 The client-side dependency graph 131
10.4.2 Automatic Scenario Generation 131
10.4.3 Identifying Feature Code 132
10.4.4 Firecrow 132
10.4.5 Extending the approach to server-side applications 133
10.4.6 Extending the approach to other domains 133

Bibliography **135**

Chapter 1

Introduction

Web applications are among the most commonly used applications today. Structurally, they are composed out of two equally important parts: the server-side, realized as a procedural application implementing data-access and business logic, and the client-side, realized as an event-driven application that acts as a user-interface (UI). One of the important benefits of the domain is easy update and deployment – no installation is required, and the user always has access to the latest application version. However, this means that web applications are usually subjected to short release cycles. Lately, by using modern web browsers and advanced scripting techniques, developers are able to build highly interactive, sophisticated, and complex applications. Unfortunately, the techniques and tools used to support their development are not as advanced as in other, more mature, software engineering disciplines. In particular, the developers are faced with poor support when trying to achieve reuse.

Building new software systems by reusing already existing artifacts has long been advocated as way to reduce development time and decrease defect density [41, 55, 12, 37]. It has been shown that reuse can lead to improved quality [23], increased productivity [7], and more satisfied customers [56]. Due to these benefits, a number of approaches aimed at facilitating reuse has been developed. Most of these approaches, such as component-based development [41] or software product-lines [47], target pre-planed reuse, in which certain software entities are explicitly built in a reusable fashion. However, there is often a desire to reuse parts of existing code that was not originally developed with reuse in mind. In

such cases, identifying the code to be reused, as well as integrating it into an already existing software system is a challenging task.

From the user's perspective, the behavior of a client-side application is composed of distinguishable parts, i.e. features, that manifest at runtime. Similar features are often used in a large number of applications, and facilitating their reuse can offer significant benefits in terms of easier and faster development. Currently, the prevailing method of reuse is pragmatic [29], copy-paste reuse, which is complex and error-prone. It is hard to identify the code for reuse, and to introduce it into a new application without errors. In addition, in the web domain, reuse is made particularly difficult for the following reasons: *i)* the application executed in the browser is a result of interplay of three different languages: HTML for defining structure, CSS for presentational aspects, and JavaScript for the behavior, and there is no trivial mapping between the source code and the application displayed in the browser; *ii)* JavaScript is a highly dynamic scripting language with characteristics that complicate code analysis; *iii)* the global application state plays a much bigger role than in most other domains, and there are many implicit dependencies within the application; *iv)* currently there is no built-in support for structuring code in a way that facilitates safe reuse, e.g. as independent components with well-defined interfaces; and *v)* code responsible for a certain feature is often intermixed with irrelevant code. This means that a single application feature, rather than being implemented by a single package, class or a method, is usually implemented by a number of code fragments spread across three different languages with many implicit dependencies. All these challenges mean that it is usually hard to identify the code responsible for the implementation of the desired feature, and even if the code is identified, it is difficult to introduce it into an existing application without errors – there is need for automating reuse.

1.1 Motivation

Consider two open-source WordPress[1] applications shown in Figure 1.1.

The top application has two features: feature A represented by the image slider control denoted with a top dashed red rectangle (mark A), and feature B, represented by the container denoted by a bottom dashed red rectangle (mark B). Feature A is triggered by clicking on the arrow

[1] http://wordpress.org/

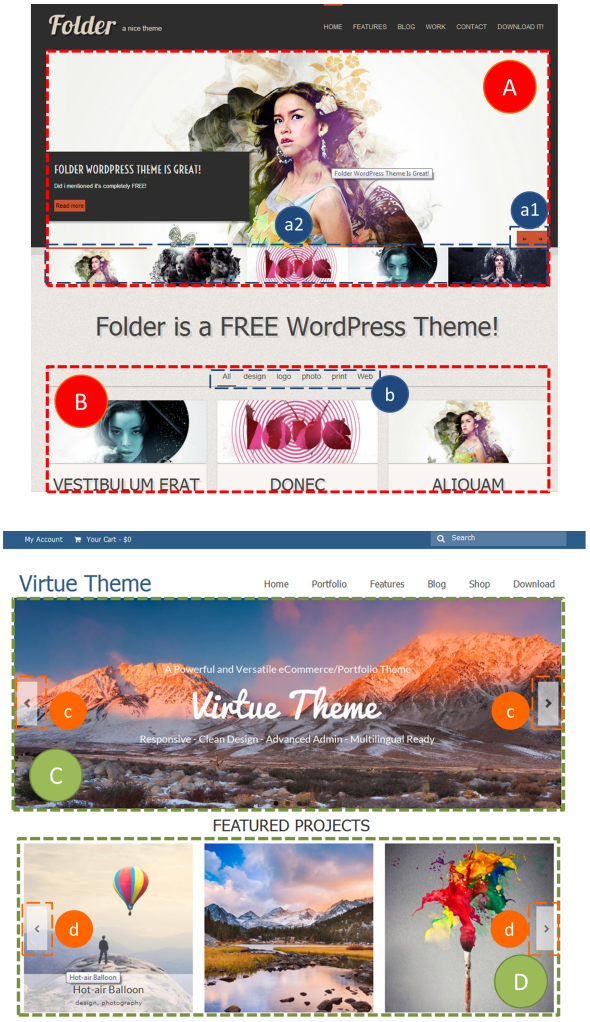


Figure 1.1: The UI of the two applications from the motivating example

buttons (mark a1) or by clicking on the image thumbnails (mark a2), and manifests with a slide effect from the current image to the subsequent image. Feature *B* is triggered by clicking on one of the labels (mark b),

and manifests by fading out the articles not described by the label, and rearranging the remaining ones.

The bottom application also has two features: feature C, represented by the image toggler denoted with the top dashed green rectangle (mark C), and feature D, represented by the image slider denoted by the bottom dashed green rectangle (mark D). Feature C is triggered either automatically, after a given timeout period has expired or by clicking on one of the arrow buttons (mark c). The feature manifests by fading out the current image and fading in the subsequent image. Feature D is triggered by clicking on the edge arrow buttons (mark d) and manifests with a visual slide effect to the subsequent image.

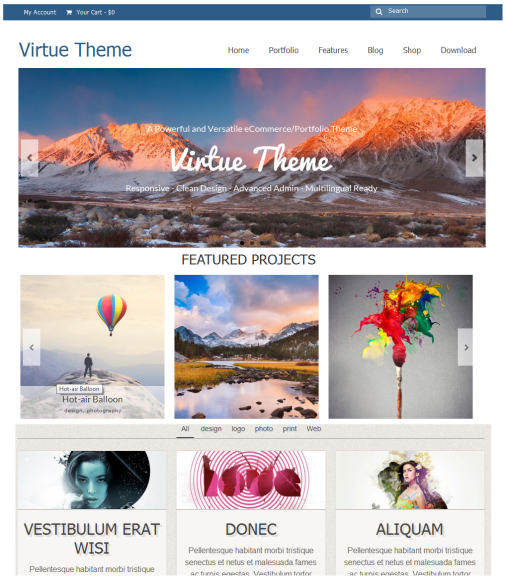


Figure 1.2: Motivating example reuse result

Consider a case where a developer needs to add a feature that can show and hide articles described with a certain label (feature B from the first application) to the second application. Instead of developing one from scratch, the developer wants to reuse feature B. In order to do this, the developer has to identify the code that implements the feature in the first application and has to embed it into the second application

(end result shown in Figure 1.2). Both of these tasks are difficult and time consuming. Identifying the exact code is difficult because the code responsible for the implementation is intermixed with irrelevant code, and embedding the code is difficult because a number of conflicts, which can break the behavior and presentation of features B, C, and D, can occur.

1.2 Research questions

The main research challenge of automating reuse in client-side web application development is broken down into a set of more concrete questions which have guided different research phases.

In order to reuse a certain feature, first we have to be able to isolate its implementation details. This is a challenging task, for the following reasons: *i*) the code responsible for feature implementation is usually intermixed with code irrelevant in respect to this feature; *ii)* the feature is implemented by a combination of different languages, where the most complex one is a dynamic scripting language (JavaScript); and *iii)* a feature manifests when a user performs certain actions. These problems lead to the first research question:

> **Research Question 1**: How can we identify the subset of the web application source code and resources that implement a particular feature?

Once the code and resources of a feature have been identified and extracted we have to enable their inclusion into the target application. This is a complex problem, because by doing this we change the execution environment that both the feature and the target application rely on for their behavior. This can cause a number of problems and conflicts in both the feature and the target application which have to be detected and fixed. This leads to the second research question:

> **Research Question 2**: How can we introduce the source code and resources of a feature into an already existing application, without breaking the functionality of neither the feature nor the target application?

Client-side web applications are event-driven UI applications in which features manifest when the user performs certain sequences of actions

(scenarios). Specifying feature scenarios that capture the complete behavior of a feature is often a time-consuming activity. This leads to the third and final research question:

> **Research Question 3**: How can we automatically generate scenarios that cause the manifestation of a client-side feature?

1.3 Contributions

The overall contribution of the thesis is a method and the accompanying tool for automating feature reuse in client-side web application development. When related to the research questions (RQ), the contributions can also be defined as:

1. **A client-side dependency graph** (RQ1, RQ2)
   We have defined a client-side dependency graph capable of tracking dependencies that exist in a particular scenario. We have also defined algorithms for its construction and traversal.
2. **A method for identifying feature code and resources** (RQ1)
   We have developed a method that is able to, by analyzing the execution of an application and a client-side dependency graph, determine a subset of the application's code and resources responsible for the implementation of a given feature.
3. **A method for integrating feature code in an existing application** (RQ2)
   We have identified a set of problems that can occur when introducing code from one application into another, and have defined a method capable of detecting and fixing those problems. The method is based on dependency graph analysis and the dynamic analysis of application execution.
4. **Automatic generation of scenarios** (RQ3)
   We have defined a method for automatic generation of scenarios. The method works by analyzing the application source code and systematically exploring the event and value space of the application. It is capable of generating scenarios that cause the manifestation of a certain feature, as well as the scenarios that target the whole application.

1.4 Research Methodology

This research is motivated by a practical, industrial problem – enabling reuse of web application features not necessarily designed for reuse. For this reason, the research falls into the category of applied research, but with solutions that contribute to disciplines of web application analysis and reuse in general. The basic research methodology was to observe existing and to propose better solutions to problems at hand; build, develop, measure, analyze, and validate the solutions and repeat the process until no more improvements appeared possible. In essence, we have performed the following steps in several cycles:

1. Perform a literature review on the current research problem.
2. Formulate a candidate solution based on the state of art and state of practice.
3. Construct a tool prototype that implements the proposed solutions.
4. Verify by performing experiments on case study applications.

More specifically, in our case this meant that we first developed a tool prototype that instrumented the browser and dynamically analyzed the execution of the web application in order to identify code related to certain behavior. While performing the experiments we noticed that not all code expressions executed during a certain behavior are important for that behavior, and that there is a significant number of executed code constructs that are irrelevant for the target behavior. This led us to the first research question: identifying feature code. In order to solve this problem we have studied the state of the art in program slicing, have defined a client-side dependency graph and the algorithms for its construction. We have defined an identification process based on the dynamic analysis of application execution and dependency graph traversal. We have evaluated the approach by performing experiments based on different usages of the method. The evaluation has shown that the method is capable of identifying the implementation details of individual features, and that by extracting the identified code considerable savings, in terms of code size and increased performance, can be achieved. Next, in order to reuse the code, we have developed a method capable of integrating code from one application into another application. We have

evaluated the reuse process based on user-specified scenarios on a number of case study web applications. The experiment has shown that, in the case study applications, the method was capable of identifying and fixing problems that happen when introducing feature code into an already existing application. Both the identification method and the reuse method are based on dynamic analysis of application behavior while certain scenarios are exercised. Since specifying these scenarios is often a difficult, error-prone and time-consuming activity, for the third research question we decided to focus on how to automatically generate application scenarios. We studied the state of the art in web application testing, and have developed a method that generates scenarios that target specific features in client-side web applications. We have tested a method on a number of case study applications, and have compared them to other, similar approaches. The evaluation has shown that the method is able to generate scenarios that target specific features, and that, in certain cases, the method is able to achieve higher coverage than state of the art methods.

1.5 Publications

This section gives a short description of the papers the thesis is based on. For all papers I have been the main author, while other coauthors have contributed with valuable discussions and reviews.

Paper A

Extracting Client-side Web User Interface Controls, Josip Maras, Maja Štula, Jan Carlson, International Conference of Web Engineering 2010 poster session (short paper).

Summary: In this paper we present our first results on extracting easily reusable web user-interface controls. We give a first description of a tool called Firecrow that we have developed to facilitate the extraction of reusable client-side controls by profiling a series of interactions, carried out by the developer. This research was our first step in answering the first research question, and is directly related to the second contribution (*2. A method for identifying feature code and resources*).

Paper B

Reusing Web Application User-Interface Controls, Josip Maras, Maja Štula, Jan Carlson, International Conference on Web Engineering 2011.
Summary: The paper defines a method for reusing client-side web application user-interface controls. It is focused on defining how to use the profiling data gathered during the execution of a sequence of actions, to identify the code responsible for the behavior of a certain UI control. We also introduce a simple method for including the identified code into another application, thereby achieving reuse. This research is directly related to the first and second research question, and to the second and third contribution (*2. A method for identifying feature code and resources*, *3. A method for integrating feature code in an existing application*).

Paper C

Client-side web application slicing; Josip Maras, Ivica Crnković, Jan Carlson, Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, 2011. (short paper).
Summary: In papers A and B, we have relied on profiling to create a connection between the executed code and the UI control selected by the user, and we have considered all lines visited while manifesting a certain behavior as important. But, as is shown in this work, code constructs that implement a certain behavior are actually a subset of the executed constructs. In this short paper we present our first work on defining the client-side dependency graph capable of capturing control and data dependencies between different parts of the client-side web application, and we improve upon the process presented in Paper B. This paper directly contributes to the first research question, and to the first and second contribution (*1. A client-side dependency graph*, *2. A method for identifying feature code and resources*).

Paper D

Extracting Client-side Web Application Code, Josip Maras, Jan Carlson, Ivica Crnković, Proceedings of the 21st international conference on World Wide Web. ACM, 2012.
Summary: This paper is a direct expansion of paper C. We show how by analyzing the application execution while a scenario is being exercised,

code responsible for a certain behavior can be identified, how dependencies between different parts of the application can be tracked by defining a client-side dependency graph, and how in the end only the code responsible for a certain behavior can be extracted. The evaluation has shown that the method is capable of extracting stand-alone behaviors, while achieving considerable savings in terms of code size and application performance. This paper directly contributes to the first research question, and to the first and second contribution (*1. A client-side dependency graph*, *2. A method for identifying feature code and resources*).

Paper E

Towards Automatic Client-side Feature Reuse, Josip Maras, Maja Štula, Jan Carlson, Ivica Crnković, Web Information System Engineering, WISE 2013, (short paper).
Summary: In this paper we present the extensions and improvements to the reuse process described in paper B. Introducing the code that implements a feature from one application into another can introduce a number of different types of errors that are time-consuming to detect and fix. We present a method for performing feature reuse. We identify problems that occur when introducing code from one application into another, present a set of algorithms that detect and fix those problems, and perform the actual code merging. We have evaluated the approach on a number of representative case studies that have shown that the method is capable of performing feature reuse. This research directly contributes to answering the second research question, and represents the third contribution (*3. A method for integrating feature code in an existing application*).

Paper F

Generating feature usage scenarios in Client-side Web Applications, Josip Maras, Jan Carlson, Ivica Crnković, International Conference on Web Engineering 2013.
Summary: In many software engineering activities (e.g. testing) representative sequences of events (i.e scenarios) that execute the application with high code coverage are required. Specifying these scenarios is a time-consuming and error-prone activity that often has to be performed multiple times during the development cycle. In this paper we present

a method and a tool for automatic generation of scenarios. The method can be configured to target either the whole web application, or certain visual and behavioral units (UI controls). The method is based on dynamic analysis and systematic exploration of application's event and value space. We have also tested the approach on a suite of web applications, and have found out that a considerable increase in code coverage, when compared to the initial coverage achieved by loading the page and executing all registered events, can be achieved. This research directly contributes to answering the third research question, and is the fourth contribution of this thesis (*4. Automatic generation of scenarios*).

Paper G

Identifying Code of Individual Features in Client-side Web Applications, Josip Maras, Maja Štula, Jan Carlson, Ivica Crnković, IEEE Transactions on Software Engineering, vol. 39 no. 12, 2013.
Summary: In this journal paper we aggregate and expand ideas and results presented in papers A, B, C, and D. The paper defines the client-side web application conceptual model, specifies the process of identifying code and resources of a feature, and defines a client-side dependency graph. It presents algorithms for building the dependency graph, identifying important nodes and edges that capture the behavior of a feature, and algorithms for identifying code and resources that implement a feature by traversing the client-side dependency graph. We have evaluated the approach, and the experiments have shown that the method is able to identify the implementation details of individual features, and that by extracting the identified code considerable savings in terms of code size and increased performance can be achieved.

1.6 Thesis outline

The rest of the thesis is organized as follows: Chapter 2 – *Background*, introduces the notions and techniques necessary to understand the approach. The chapter gives an introduction to web applications and features, and presents dynamic analysis, automated testing and program slicing as techniques vital to our approach. Chapter 3 – *The Reuse Process Overview*, gives an overview of the whole approach and, in high detail explains each of the necessary steps. In Chapter 4 – *Client-side*

Dependency Graph, we define the dependency graph used to capture dependencies that exist in a client-side web application. Chapter 5 – *Automatic Scenario Generation*, introduces a technique, based on the systematic exploration of the application's event and value space, for automatically generating scenarios that target either the behavior of the whole application or particular application features. Chapter 6 – *Identifying Code of Individual features*, presents a method for the identification of feature implementation details, and Chapter 7 – *Integrating Features*, describes a technique for the integration of feature code into an already existing application, thereby achieving reuse. Chapter 8 – *Firecrow*, presents a tool that implements algorithms and processes described throughout the thesis, and Chapter 9 presents the related work. Finally, Chapter 10 gives a conclusion and describes possible suggestions for future work.

Chapter 2

Background

This chapter introduces important technical concepts used throughout the thesis. It provides a web application primer, and gives an introduction to three important techniques: *i)* feature location, *ii)* automatic test generation, and *iii)* program slicing, which are used throughout the processes described in the thesis.

2.1 Web Applications

Web applications are structurally composed out of two equally important parts: the server-side and the client-side. The server-side is usually realized as a sequential application implementing data-access and business logic, while the client-side is an event-driven applications that acts as a user-interface (UI) to the server-side.

The life-time of a web application (shown in Figure 2.1) begins with the user typing in a URL[1] in the browser or clicking on a URL link in an already existing application. The URL contains all information needed to target a specific application on a specific web server. Based on the provided URL, the browser creates an HTTP[2] request to the server requesting the specified application. The server processes the request, finds the file, executes any associated server-side code, and responds with an HTTP response that contains the HTML[3] document that defines

[1] Uniform Resource Locator
[2] HyperText Tranfer Protocol
[3] HyperText Markup Language

the application. The browser parses the code, builds the layout of the page, and for each referenced external file (e.g. images, videos, audio, JavaScript, style files) creates an additional HTTP request that is sent to the server.

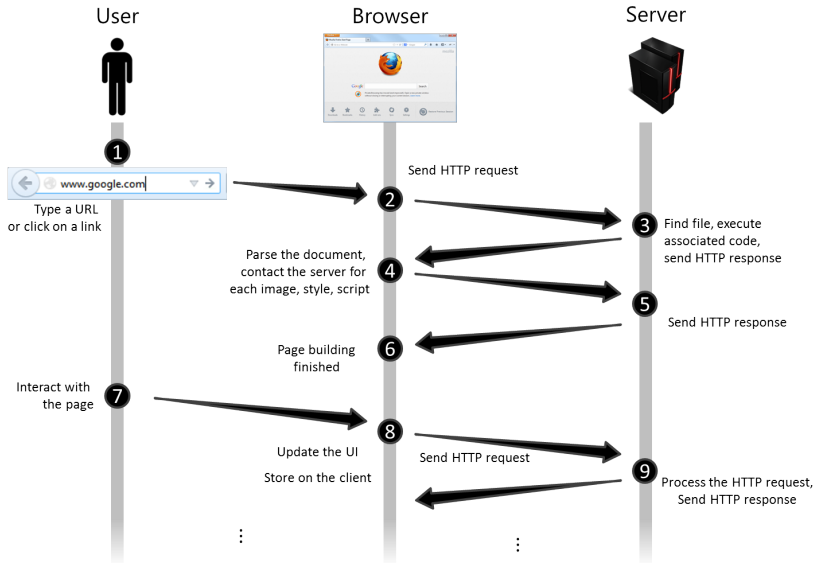


Figure 2.1: Life-time of a web application

Once the application is built and displayed in the browser, the user can interact with it. At any time, during the life-time of the application, the browser can initiate communication with the server-side (by sending an HTTP request), can update the UI of the application, or store information in the client browser.

2.1.1 Client-side Web Application Primer

A client-side web application is an HTML page that includes JavaScript code, CSS code, and various resources (e.g. images and fonts), and the interplay of these elements produces the result displayed in the browser.

HTML (HyperText Markup Language) is a markup language used for specifying the structure of a Web application. The markup takes the

form of *elements* applied to content, typically text. Additional properties can be assigned to each HTML element by adding one or more *attributes*.

```
1  <html>
2   <head>
3    <title>Example Page</title>
4   </head>
5   <body>
6    <div id="container">
7     <img src="image.png"/>
8     <span class="caption">Test</span>
9    </span>
10  </body>
11 </html>
```

Listing 2.1: Example HTML code

For example, Listing 2.1 presents a simple HTML document composed out of HTML elements (e.g. html, head, title), attributes (e.g. attribute *src="image.png"* of an *img* element in line 7 that specifies the location of the image), and text content (string *"Test"* in line 8). Based on the HTML markup, the browser builds the structure and content of a web application.

CSS (Cascading Style Sheets) is a declarative language used to specify the presentational aspects of HTML elements. The CSS code is composed of CSS rules, each rule consisting of a CSS selector and a set of property-value pairs. A CSS selector, by combing node type, node attributes (e.g. id, class), and node position in the page hierarchy, is used to specify to which HTML elements the given property-value pairs will be applied to.

```
1  img { border-style:solid; border-color: red;  }
2  #container {color:blue; }
3  #container .caption { font-weight: bold; }
```

Listing 2.2: Example CSS code

Listing 2.2 gives a simple example of CSS rules: the rule at line 1 specifies that a property *border-style* with a value *solid* and a property *border-color* with a value *red* should be applied to any element of type *img*, while the rule at line 3 specifies that a property *font-weight* with value bold should be applied to elements with a class attribute *caption* that are descendants of an element with the *id* attribute *container*.

JavaScript is a weakly typed, imperative, object-oriented scripting language with prototype based inheritance. It has no type declarations

and only run-time checking of calls and field accesses. Functions are first-class objects and can be manipulated and passed around like other objects. JavaScript is a dynamic language and everything can be modified at runtime, from fields and methods of an object to its prototype. One object can act as a prototype to a number of different objects, and changes to that prototype affect all derived objects. The language also offers an *eval* function which can execute an arbitrary string of JavaScript code. JavaScript code, at least when discussing client-side web applications, is executed inside a browser which offers a number of globally available built-in objects: e.g. the global *window* object which stores all global variables and acts as an interface to the window in which the application is executed; the *document* object used as an interface to the structure of the page, etc. Due to the dynamicity of JavaScript each built-in object can be accessed and modified from any point in the application.

```
1  function addText(element) {
2   element.textContent += " added by JavaScript";
3  }
4
5  var caption=document.querySelector("#container .caption");
6  addText(caption);
```

Listing 2.3: Example JavaScript code

Listing 2.3 presents a simple example of a JavaScript application that selects an element from the page by using the global *document* object (line 5), and appends a text string to it (line 2).

Life-cycle. Client-side web applications are event-driven UI applications, and a majority of code is executed as a response to user-generated events. Their life-cycle can be divided into two phases: *i)* page initialization and *ii)* event-handling. The purpose of the page initialization phase is to build the UI of the web page. The browser achieves this by parsing the HTML code and building a representation of the HTML document – the Document Object Model (DOM). When parsing the HTML code the DOM is constructed one HTML element at a time. After the last element is parsed and the UI is built, the application enters the event-handling phase, where code is executed in response to events. All UI updates are done by JavaScript modifications of the DOM, which can go as far as completely reshaping the DOM, or even modifying the code of the application.

2.2 Software Reuse

Software reuse is the process of creating software systems from existing artifacts rather than building them from scratch. The types of artifacts that can be reused are not limited to source code, and may include requirements, design documents, architecture, and even software development processes. Reuse has often been touted as a process that can reduce development time, reduce defect density, and increase developer productivity [41, 37, 11]. Numerous approaches to software reuse have been developed, ranging from preplanned approaches where software artifacts are constructed with reuse in mind (e.g. object-oriented inheritance [17], software components [41], and software product lines [47]), all the way to more pragmatic approaches [30], which facilitate reuse of artifacts not explicitly designed for reuse.

Preplanned reuse approaches, even though they should, in theory, provide all reuse benefits, suffer from three main drawbacks [32]: *i)* the economic infeasibility of developing all software in a reusable fashion [24, 9]; *ii)* the difficulty in predicting which pieces of software should be built as reusable [60, 24, 61]; and *iii)* even software designed as reusable embeds a set of assumptions about how it should be reused that can hamper its ability to be deployed in many contexts [9, 25].

In contrast, the pragmatic approaches [30] such as code scavenging [37], ad-hoc reuse [49], opportunistic reuse [51] enable the reuse of source code that was not explicitly designed for reuse. Pragmatic reuse tasks are often stigmatized as the "wrong" way to reuse code, mostly due to their non-systematic, ad-hoc nature [32]. However, by facilitating these tasks and making them more automatic and systematic, we can access the untapped potential of already existing code, potentially increasing developer productivity and lowering overall development costs.

2.3 Features

From an external perspective, a user understands a system as a collection of features that correspond to system behaviors designed to implement system requirements. The exact meaning of the term *feature* depends on the context. For example, the IEEE [1] defines the term as: a distinguishing characteristic of a system item that includes both functional and nonfunctional attributes such as performance and reusability. On

the other hand, within the program comprehension community the feature is taken to be a specific functionality that is defined by requirements and accessible to developers and users [19, 48, 20].

In this work, we adopt the term feature, defined in [20] as: a user-triggerable activity of a system. We consider that a feature is an abstract description of a system's expected behavior that manifests itself at runtime, when the user provides the system with adequate input.

It is important to note that, unlike the feature definition given by the IEEE [1], the feature definition used in the program comprehension community does not include non-functional requirements (e.g. performance, maintainability). So, in the context of this thesis, only functional features are relevant.

2.3.1 Feature Location

Features are relatively straightforward to distinguish from the user's perspective. However, the same cannot be said for their implementation details. In general, it is difficult and time-consuming to exactly identify the source code responsible for the implementation of a particular feature. At the same time, understanding software features and their implementation details is a vital activity in software maintenance – up to 60% of software-engineering effort is spent on understanding the software system at hand [14, 22]. Identifying the locations in the source code that correspond to a specific feature is know as *feature location* (or concept location) [10, 50].

Feature location techniques utilize different types of analyses in order to establish a traceability between a feature of interest, specified by the user, and the artifacts implementing that feature. The most common types of analyses include textual analysis, static analysis, and dynamic analysis (and their combinations). Textual approaches analyze the source code text based on the idea that identifiers and comments encode domain knowledge, and that a feature may be implemented using a similar set of words throughout the system. Static analysis examines structural information such as control or data flow dependencies, for all possible program inputs, often overestimating the code related to a feature [18]. Dynamic analysis, on the other hand, relies on examining the execution of an application, and it is often used for feature location when features can only be invoked and observed during runtime. Feature location using dynamic analysis generally relies on execution trace analysis,

and feature-specific scenarios are developed that invoke the desired feature. Then, the scenarios are exercised and execution traces that record information about the code that was invoked are collected and analyzed.

2.4 Dynamic Analysis

There are two distinct ways of analyzing program properties: static and dynamic analysis. While static analysis examines the program's source code in order to derive properties that hold for all executions, dynamic analysis derives properties that hold for one or more executions by examining the running program [6].

According to [15], dynamic analysis, when compared to static analysis has both benefits and limitations. The benefits of dynamic analysis are:

- Higher precision with regard to the actual behavior of the software system.
- The fact that a goal-oriented strategy can be used, which entails the definition of an execution scenario such that only the parts of interest of the software system are analyzed.

Dynamic analysis also has a number of limitations:

- The inherent incompleteness of dynamic analysis, as the behavior or traces under analysis capture only a small fraction of the usually infinite execution domain of the program under study.
- The difficulty of determining which scenarios to execute in order to trigger the program elements of interest. In practice, test suites or recorded executions involving user interaction with the system can be used.
- The scalability of dynamic analysis due to the large amounts of data that may be introduced in dynamic analysis, which affects performance, storage, and the cognitive load humans can deal with.
- The observer effect, the phenomenon in which software acts differently when under observation, might pose a problem in cases where timing issues play a role.

2.5 Automated Testing

Software testing typically consumes between 30% and 50% of the total cost of building new software [8, 21]. For this reason, it is often presumed that test automation can lower the cost and increase the reliability of software systems. The generation and evaluation of tests can generally be categorized in two classes: black-box testing and white-box testing. The main artifact in black-box testing is the specification of the software system. The goal of black-box testing is to test the correctness of the external system behavior, according to the specification, by studying only the relationship between input and output values. After a suite of black box tests is unable to find any additional errors, there is some confidence that the system indeed behaves according to the specification. The biggest disadvantage of black-box testing is the lack of objective criteria to evaluate the quality of test suites.

The main artifact in white-box testing is the source code of the software system. By using the source code it is possible to set-up different criteria to asses the quality of the white-box test suite. Test suite quality is usually measured in respect to which extent elements of the program's source code are covered by the test suite. The most widely used coverage criteria are: *i)* statement coverage, which aims to execute each statement in the program's source code at least once; *ii)* branch coverage, which aims to cover all branches of each control structure (in an if-else statement, that both the if and else branches get executed); *iii)* condition coverage, which aims to cover each boolean expression, in a way that they are evaluated at least one time as *true* and one time as *false*.

Over the years, a number of different approaches to test-case generation have been developed [4], such as:

- *Random testing*, where test cases are randomly generated across the input domain.
- *Search-based testing*, in which search-based optimization algorithms are used to automate the search for test data guided by a fitness function that captures the current test objective.
- *Model-based test case generation* [4], where models of software systems are used to generate test suites.
- *Symbolic execution* [36], where symbolic values are used instead of concrete values as program input.

In *Symbolic execution*, symbolic values are used as program input, and program variables are represented as symbolic expressions of those inputs. During program execution, all encountered control-flow branches (e.g. if statements, conditional expressions) whose branching conditions are expressions that contain symbolic variables are added to the so called *path-constraint* which carries information about how the control-flow of the execution depends on the input parameters. By modifying the path-constraint and solving newly obtained input constraints, concrete values which cause a specific control-flow can be obtained. These values can then be used to specify test cases. A number of variations to symbolic execution have been developed, e.g. the technique of *concolic execution* [54], which executes the program for both concrete and symbolic values.

2.6 Program Slicing

Program slicing [62] is a method that starting from a subset of a program's behavior, reduces that program to a minimal form which still produces that behavior. A program slice consists of the parts of a program that affect the values computed at a point of interest, a so called slicing criterion. The original concept of a program slice was introduced by Weiser [62], as a reduced, executable program obtained from a program by removing statements, such that the slice replicates a part of the behavior of the original program. Another common definition of a slice is a subset of the statements and control predicates of the program that directly or indirectly affect the values computed at the criterion, but that do not necessarily constitute an executable program. There is a number of existing program slicing methods, and they can be divided into two different categories: *i)* static and *ii)* dynamic slicing. Static slicing makes no assumptions on the program input, while dynamic slicing studies program execution with specific program inputs.

2.6.1 Static Slicing

Static slicing is the original slicing approach proposed by Weiser [62]. In static slicing, programs are analyzed statically, regardless of the program input (i.e. for all possible program inputs). A static slicing criterion is usually specified by a program point and a set of variables. Consider the example program that asks the user for a number n, and calculates the

sum and the product of the first n positive numbers (Listing 2.4).

```
1  var n = prompt();
2  var i = 0;
3  var sum = 0;
4  var product = 1;
5
6  while(i < n) {
7    sum += i;
8    product *= i;
9    i++;
10 }
11
12 document.writeln(product);
13 document.writeln(sum);
```

Listing 2.4: Example program

```
1  var n = prompt();
2  var i = 0;
3  var sum = 0;
4
5
6  while(i < n) {
7    sum += i;
8
9    i++;
10 }
11
12
13 document.writeln(sum);
```

Listing 2.5: Line 13 static slice

The example program has two behaviors: *i)* calculating the product, and *ii)* calculating the sum of the first n positive numbers. If, for example, we want to obtain only the second behavior (calculating the sum), we would set the slicing criterion to line 13 (*document.writeln(sum)*), and would obtain the resulting slice shown in Listing 2.5. Notice how all lines related to the calculation of the product have been removed from the original program, due to not contributing to the calculation of the sum.

In Weiser's original approach, slices are computed by computing consecutive sets of transitively relevant statements, with regard to data and control dependencies. There are also a number of methods, first proposed by Ottenstein and Ottenstein [46], that define the slicing problem as a reachability problem in a program dependence graph (PDG). A PDG is a directed graph whose vertices are statements and control predicates, and edges data and control dependencies.

In general, slices can also be computed in two ways: *i)* by the backward traversal of the PDG – backward slicing, as in the example from Listing 2.5; and *ii)* by the forward traversal of the PDG – forward slicing. A backward slice is composed of all statements that the slicing criterion is dependent on, while a forward slice is composed of all statements dependent on the slicing criteria.

2.6.2 Dynamic Slicing

In contrast to static slicing, dynamic slicing is performed only for certain values of the program input – a particular program execution is analyzed. A dynamic slicing criterion is typically composed out of an input, the occurrence of program statement, and a set of variables.

```
1  var n = prompt();
2  var i = 0;
3  var sum = 0;
4
5
6
7
8
9
10
11
12
13 document.writeln(sum);
```

Listing 2.6: Dynamic slice for line 13 and n=0

```
1  var n = prompt();
2  var i = 0;
3  var sum = 0;
4
5
6  while(i < n) {
7    sum += i;
8
9    i++;
10 }
11
12
13 document.writeln(sum);
```

Listing 2.7: Dynamic slice for line 13 and n=2

Consider the same program from Listing 2.4. If dynamic slicing is performed with a slicing criterion set to line 13, with a input *n=0*, the resulting dynamic slice would be equal to the code presented in Listing 2.6 (since the while loop is not executed for $n = 0$ none of its statements are relevant). However, if we set the slicing criterion with respect to *n=2*, then the while loop will be executed, its statements contribute to the value of the *sum* variable, and thus are included in the resulting slice shown in Listing 2.7.

Chapter 3

The Reuse Process Overview

The goal of the process is to automate reuse in web application development. More concretely, the goal is to enable code-level reuse of features from one client-side application into an already existing client-side application. In this chapter, we define what a client-side feature is and give an overview of the reuse process.

3.1 Client-side Features

A client-side web application is defined with its structure, the presentational aspects of that structure, and the behavior that occurs on that structure (Figure 3.1). Client-side applications act as user interfaces to server-side applications, and their two primary functions are: *i)* to communicate with the user over their UI, and *ii)* to communicate with the server by exchanging messages.

An application offers a number of features. In general, a feature is a user-triggerable activity of a system [20]. In the context of client-side applications, we define a client-side feature as: a subset of the application's behavior, triggered by sequences of user-generated events, that manifests at runtime with: *i)* changes to certain parts of the application structure, and/or *ii)* communications with the server.

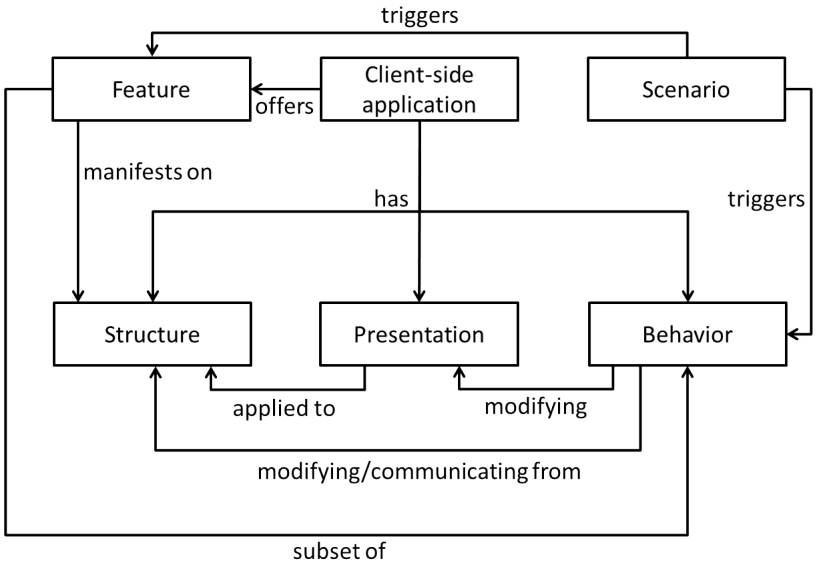


Figure 3.1: A conceptual model of a client-side web application

3.2 Defining the Reuse Process

Let A and B be two client-side web applications, each defined with its HTML code, CSS code, JavaScript code, and resources. An application offers a set of features F, and each feature is implemented by a subset of the application's code and resources. However, identifying the exact subset is a challenging task because the code responsible for the desired feature is often intermixed with code that is irrelevant from the perspective of the feature. The goal of the reuse process is to identify the code and resources of feature f_a from application A and to include them, without errors, into application B. With this inclusion, a new application B' that offers both the feature f_a from A and the features F_B from B, is created.

3.3 The Reuse process

Figure 3.2 presents the overview of the reuse process. As input, the process receives the source code of both the application A from which the feature will be extracted, and the application B where the feature code will be reused. The process also receives *Feature Descriptors* which define structural parts of application A where the feature manifests, the scenarios that trigger the manifestation of the feature (*Feature Scenarios*), the scenarios that trigger the behavior of the target application (*Application Scenarios*), and the *Reuse Positions* which define where the feature related structure will be included in application B.

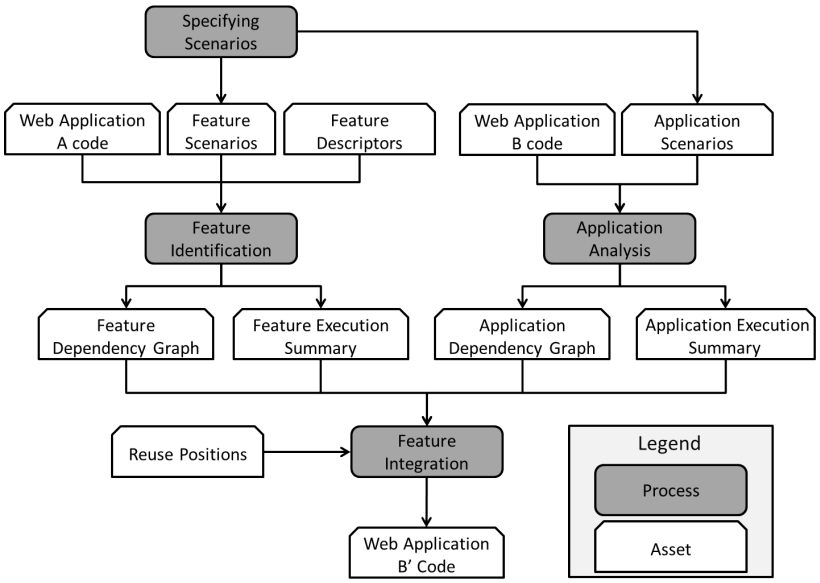


Figure 3.2: Overview of the reuse process

3.3.1 Feature Identification

The first step of the process is to identify the subset of application A that implements the target feature. Due to the fact that a feature manifests when a user performs a certain scenario, the feature identification

process is based on the dynamic analysis of application execution triggered by a scenario. In essence, this phase identifies the subset of the application's source code and resources that implement the target feature manifested by the scenarios on parts of the page specified by the *Feature Descriptors*. This identification is performed by the means of a client-side dependency graph (Chapter 4), which captures dependencies that exist in a client-side web application. The *Feature Identification* phase is described in Chapter 6. Since a prerequisite for the feature identification is the existence of feature scenarios, in the next section we describe how the scenarios can be specified.

3.3.2 Specifying Scenarios

Scenarios causing the manifestation of a particular feature can be obtained in two ways: *i)* by the developer manually specifying them, or *ii)* with automatic scenario generation techniques.

Since there does not have to exist a clear-cut set of application behaviors that constitute a feature, manual specification of usage scenarios enables users to precisely specify application behaviors that, from their perspective, constitute a feature. However, this is often a complex and time-consuming activity that requires knowledge about the internal details of the application, and comes with a cost of inadvertently forgetting certain behaviors. For this reason, we have developed an *Automatic Scenario Generation* technique (Chapter 5) that systematically explores the event and value space of the application. Even in the case where manual scenarios are used, the users can benefit from the automatically generated scenarios, because they can be used to raise awareness about different application behaviors.

Regardless of the method used to obtain scenarios, using scenarios for feature identification offers the following advantages: *i)* it does not require any formal specification of the feature, something that is rarely done in web application development; and *ii)* it enables the dynamic tracking of code dependencies, which can not be accurately done statically for a language as dynamic as JavaScript. The downsides of the approach are: *i)* scenarios are primarily suited for functional features with observable behaviors that can be triggered by the user; *ii)* the accuracy and the completeness of the captured feature is dependent on the quality of the scenarios.

3.3.3 Application Analysis

When introducing code from one application into another a number of problems can arise (Chapter 7). Since client-side applications are highly dynamic, the locations and types of these problems can not accurately be determined statically. For this reason, in addition to the dependency graph and execution summary of the feature, we also have to obtain the dependency graph and execution summaries of the target application where the feature will be reused into. This is done in the *Application Analysis* phase. This phase is similar to the *Feature Identification* phase, and the only difference is that there is no explicit phase of identifying feature code – only the dependency graph is created and the execution summary logged. As the *Feature Identification* phase, the *Application Analysis* phase is based on the dynamic analysis of application execution while scenarios triggering application behavior are exercised.

3.3.4 Feature Integration

The inclusion of code and resources of a feature into another application changes the situation in both the feature code and the application code – a new page, whose DOM is different from what is expected by the code of each individual application, is created. Also, both the feature code and the application code can make modifications to the internal objects provided by the browser, which can lead to application states not expected by the code of the other application. These unanticipated changes can create a number of possible errors and conflicts that have to be detected and fixed. For this reason, we analyze the dependency graphs and execution summaries derived in the *Feature Identification* and *Application Analysis* phase, detect the potential problems, and perform fixes on the dependency graphs. Once this is done, the process performs code merging, where the main idea is to merge the matching nodes of both applications, and to move the HTML nodes that specify the structure of the feature to a position designated by the user. Finally the process can check if the behavior of the resulting application is in accordance with the behavior in the originating applications. The *Feature Integration* phase is described in Chapter 7.

3.4 Conclusion

In this chapter, we have given an overall description of the reuse process. We have defined what exactly is meant by the term *feature*, and have specified a dynamic feature reuse process composed of three distinct phases: *scenario generation*, *feature identification*, and *feature integration*. The rest of the thesis will describe an important artifact used in the process – the *client-side dependency graph*, and will, in greater detail, present each of the distinct steps of the automatic client-side feature reuse process.

Chapter 4

Client-side Dependency Graph

In order to establish dependencies between the handling of different user actions (Chapter 5), or to accurately identify the source code and resources that implement a particular feature (Chapter 6), we have to be able to track application dependencies. One way of capturing dependencies is through a program dependency graph [46], that makes explicit both the data and control dependencies that exist in an application. In this chapter, we define a client-side dependency graph that is capable of capturing dependencies that exist in a highly-dynamic, multi-language environment that is the client-side of a web application. We also define algorithms and processes for the construction of such graphs.

4.1 Defining the dependency graph

The client-side application is composed of four different parts, HTML code, CSS code, JavaScript code, and resources, that are intertwined and must be studied as a part of the same whole. Because of this, we define the client-side dependency graph consisting of four types of vertices: HTML vertices, CSS vertices, JavaScript vertices, and resource vertices (Table 4.1). Since the client-side of the web application is extremely dynamic (e.g. new HTML elements are regularly created by JavaScript code and inserted into the DOM of the application, but also new JavaScript

and CSS code can be dynamically created with JavaScript code), we also differentiate between static (directly present in source code) and dynamic vertices (dynamically created with JavaScript code).

Table 4.1: Nodes in the client-side dependency graph

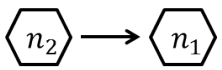
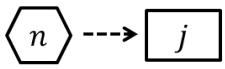
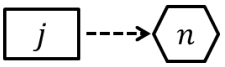
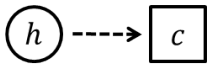
Definition	Static vertex	Dynamic vertex
h-vertex		
c-vertex		
j-vertex		
r-vertex		

Different parts of the application code can be in different relationships: *i)* a code construct can be contained within another construct (e.g. a child HTML node within the parent HTML node), *ii)* a construct can obtain its data from another construct (e.g. one JavaScript expression defines a variable used by another JavaScript expression, or an HTML node has its styles defined by a CSS rule), or *iii)* the execution of one expression depends on the value of another expression (e.g. the execution of an if statement body depends on the if statement condition). Because of these reasons the graph can have three types of arcs: structural dependency arcs, data flow arcs, and control flow arcs.

Table 4.2 shows the definition of different arc types. A straight full arrow represents structural dependencies, a straight dashed arrow data dependencies, and a curved dashed arrow control dependencies; h denotes HTML vertices, j JavaScript vertices, c CSS vertices, r resource vertices, and n denotes a vertex of arbitrary type.

Because of the inherent hierarchical organization of HTML documents the HTML layout translates very naturally to a graph representation. Except for the top one, each element has exactly one parent element, and can have zero or more child elements. The parent-child relation is the basis for forming structural dependency arcs between h-vertices. A directed structural dependency arc between two h-vertices

Table 4.2: Arcs in the client-side dependency graph

Dependency	Notation	Definition
Structural		n_2 is a child of n_1
Data		j writes data to n
Data		j reads data from n
Data		the style of h is defined in c
Data		n uses data from resource r
Control		Execution of j_2 depends on j_1

represents a parent-child relationship from a child to the parent. HTML elements can include different resources (e.g. images, videos, sounds) so there can exist a data dependency arc between an r-vertice and an h-vertice.

CSS rules are represented with c-vertices. All CSS code is contained within HTML elements, and every c-vertex has a structural dependency towards the parent h-vertex. Also, since a CSS style can be created with JavaScript code, there can exist a data dependency between a c-vertex and a j-vertex. CSS styles often reference resources such as images (e.g. defining the background of an HTML element), so there are data dependencies between c-vertices and r-vertices. Since the main goal of a CSS rule is to define styling parameters for HTML elements, there can exist a data dependency between an h-vertex and a c-vertex.

JavaScript code constructs that occur in the program are represented with j-vertices, and are derived from a simplified Abstract Syntax Tree (AST). All JavaScript code is contained in an HTML element, so a j-vertex can have a structural dependency towards a parent h-vertex. Two j-vertices can also have structural dependencies between themselves denoting that one construct is contained within the other (e.g. a relation-

ship between a function and a statement contained in its body). Data dependency arcs can be formed between j-vertices and all other types of vertices: a data dependency from one j-vertex to another denotes that the former is using the values set in the latter; an arc from a j-vertex to an h-vertex, that the j-vertex is reading data, while an arc from the h-vertex to the j-vertex means that the j-vertex is writing data to the h-vertex. An arc from a j-vertex to a c-vertex means that JavaScript code is reading data from the c-vertex. A j-vertex can also have a control dependency towards another j-vertex (e.g. statements in an if-statement towards the if-statement condition).

The main purpose of the graph is to capture the dependencies that exist in a client-side web application, and to enable the identification of source code responsible for the implementation of features, in a particular scenario. One of the problems in accurately capturing dependencies that exist in a scenario, is accounting for the execution context in which the dependencies are created (both within different function calls and loop executions, since statements are often executed multiple times). For this reason, each arc is associated with a label that uniquely specifies the context in which it was created.

4.1.1 Formal Graph Definition

A client-side dependency graph is a labeled multidigraph. In general, a multdigraph is a directed graph which is permitted to have multiple arcs (arcs with the same end vertices). Formally, it is defined as an 8-tuple $G = (V, A, \Sigma_V, \Sigma_A, s, t, L_V, L_A)$ where:

- V is a set of vertices,
- A is a set of arcs,
- Σ_V is a set of available vertex labels,
- Σ_A is a set of available arc labels,
- $S : A \to V$ is a map indicating the source vertex of an arc,
- $T : A \to V$ is a map indicating the target vertex of an arc,
- $L_V : V \to \Sigma_V$ is a map describing vertex labeling,
- $L_A : A \to \Sigma_A$ is a map describing arc labeling.

A vertex label $l_V \in L_V$ is a tuple $\langle c_i, c_c, c_n, c_r \rangle$ where c_i is a unique vertex identifier, c_c a corresponding code construct, c_n the code construct type (HTML, CSS, JavaScript, or resource), and c_r the creation type (either static or dynamic). An arc label $l_A \in L_A$ is a tuple $\langle i, d \rangle$, where i is the arc identifier that includes the identifier of the context in which the arc was created and the order of arc creation, and d is the dependency type (structural, data, or control dependency).

4.1.2 Example

In order to illustrate the client-side dependency graph, we present Listing 4.1, which shows a simple program that enables the user to enter a number n and then, when a user clicks on a button, calculates the sum and the product of the first n numbers.

```
1  <html>
2  <head>
3   <style>
4    #sCont { color: red; }
5    #pCont { color: blue; }
6    body { background-image:url("b.png")}
7    label {font-weight: bold; }
8   </style>
9  </head>
10 <body>
11  <input id="input"/>
12  <button id="cButton">Calc</button>
13  <div id="sC"></div>
14  <div id="pC"></div>
15  <script>
16   var number;
17   function add(a, b) { return a + b; }
18   var inputCont = document.querySelector("#input")
19   var sumCont = document.querySelector("#sC")
20   var pCont = document.querySelector("#pC")
21   var cButton = document.querySelector("#cButton")
22
23   inputCont.oninput = function(){
24     number = parseFloat(inputCont.value);
25   };
26
27   cButton.onclick = function() {
28    var sum = 0;
29    var product = add(0,1);
30    for(var i = 1; i <= number; i++) {
31     sum = add(sum, i)
```

```
32       product *= i
33      }
34      sCont.innerHTML="<label>Sum:" + sum + "</label>"
35      pCont.innerHTML="<label>Product:"+product+"</label>"
36     };
37  </script>
38  </body>
39  </html>
```

Listing 4.1: Example web page that calculates the sum and product of the first n numbers

Since the graph is built dynamically, for a particular application execution, in this example we will show how the graph is constructed for the scenario in which the user inputs $n = 2$ (Listing 4.2) and clicks on the button (Figure 4.1). The number at the top of the node denotes the order in which the nodes are created; each j-vertex is labeled with $@i - t$, where i is the line number and t the type of the matching code construct.

```
1  [{  "filePath": "example.html",
2       "line": 23,
3       "currentTime": 1346312751631,
4       "thisValue": "/html/body/input",
5       "args": {
6         "target": "/html/body/input",
7         "type": "input",
8         "inputStates": [{
9             element:"/html/body/input",
10            value: "2"
11        }]}},
12 {   "filePath": "example.html",
13      "line": 27,
14      "currentTime": 1346312751662,
15      "thisValue": "/html/body/button",
16      "args": {
17        "target": "/html/body/button",
18        "clientX": 124, "clientY": 242,
19        "type": "click",
20        "inputStates": [{
21          element:"/html/body/input",
22          value: "2"
23 }]}}]
```

Listing 4.2: Example event trace

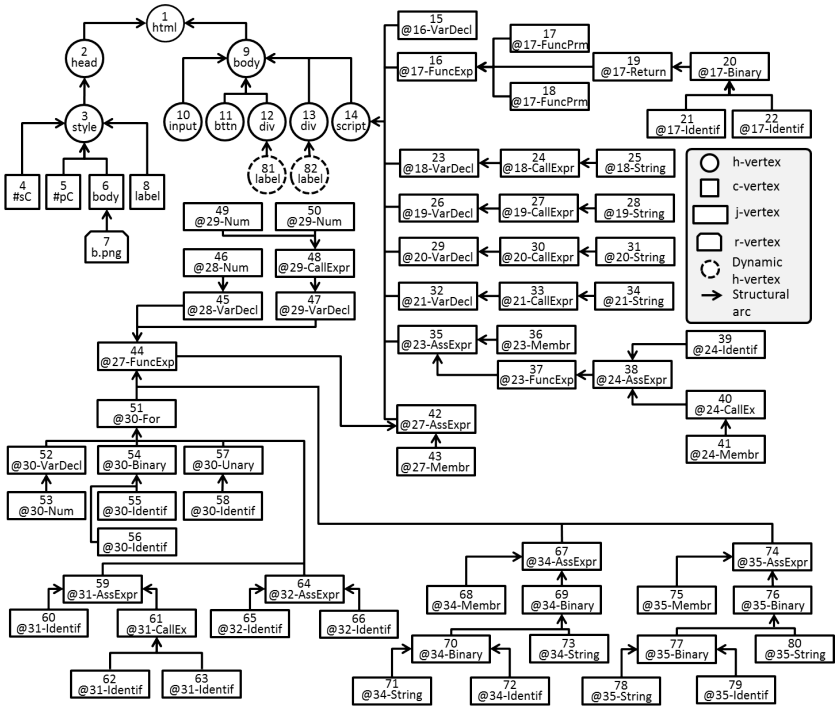


Figure 4.1: Vertices and structural dependencies of the graph constructed for the example application from Listing 4.1

4.2 Graph Construction Process

Since the dependency graph captures dynamic dependencies that exist in a particular scenario, as input the process receives the source code of the application and the event trace of the scenario (Algorithm 1). The main idea of the process is that web application code is interpreted with an event-trace as a guideline, according to standard rules of web application interpretation.

From a technical perspective, we have developed a custom JavaScript interpreter (Chapter 8) based on the process by which the browser executes the web page, an interpreter that is able to, besides evaluating web application code, to keep track of information necessary to estab-

lish dependencies between different parts of the application. With this information, while expressions are being evaluated, the process is able to create matching graph vertices and arcs.

Algorithm 1 Graph Construction

```
 1: function buildGraph(code, eventTrace)
 2:     createInitialGraph(code)
 3:     mainHtmlNode ← getRoot(code)
 4:     processSubtree(mainHtmlNode)
 5:     for all event in eventTrace do
 6:         interpretJs(getHandler(event))
 7:     end for
 8: end function
 9: function processSubtree(hNode)
10:     if isScriptElement(hNode) then
11:         interpretJs(hNode)
12:     else if isNotCssElement(hNode) then
13:         traverseCssRulesAndCreateDeps(hNode)
14:         for all hChild in hNode do
15:             processSubtree(hChild)
16:         end for
17:     end if
18: end function
```

In the first step of the algorithm, the function *createInitialGraph* (line 2, Algorithm 1) parses the whole source code of the application (including HTML, CSS, and JavaScript code) and creates all static vertices and structural dependencies that exist directly in the source code of the application: one h-vertex for each HTML element, one j-vertex for each element in the AST of JavaScript code, one c-vertex for each CSS rule, and one r-vertex for each resource directly referenced by either a CSS rule or an HTML element. Next, the process follows standard rules of web application interpretation and visits all contained HTML elements, in the *processSubtree* function.

The *processSubtree* function is shown in line 9, Algorithm 1. If the processed node is a script element then the process starts interpreting the contained JavaScript code in order to create dynamic vertices and dynamic arcs that capture data and control dependencies. If the processed node is not a CSS element (style or link element), i.e. is a standard HTML node, then all CSS rules are traversed and if the *hNode* satisfies a CSS rule, a data dependency is created from the matching h-vertex to the c-vertex that matches the CSS rule (line 13). Next, for each child of

the *hNode* the *processSubtree* function is recursively called.

Complexity. In total, the *processSubtree* function will be executed once per HTML node, and for each HTML node all CSS rules will be traversed (*traverseCssRulesAndCreateDeps*). In lines 6 and 11, Algorithm 1, the algorithm invokes the *interpretJs* function, whose execution directly depends on the number of execution steps in a particular scenario. In total, the complexity of the algorithm can be approximated as: $O(|h| \cdot |c| + s)$, where h is the set of HTML nodes, c a set of CSS rules, and s the number of execution steps in a scenario.

Example. For the example application shown in Listing 4.1, the graph is shown in Figure 4.1. First, by calling the *createStaticNodes* function, all static graph vertices and their structural dependencies (full arrows in Figure 4.1) are created. In this case, this means that all static h-vertices (full circles in Figure 4.1) matching the HTML nodes at lines 1 – 3, 10 – 15 (Listing 4.1), all c-vertices (squares in Figure 4.1) matching the CSS rules at lines 4 – 7, and all j-vertices (rectangles in Figure 4.1) matching the JavaScript code at lines 16 – 36 are created. Next, the HTML nodes up to line 15 are visited, and data-dependencies from h-vertices to c-vertices matching the CSS rules are created: from the body HTML element to the CSS rule in line 6, from the div element in line 13 to the CSS rule in line 4, and from the div element in line 14 to the CSS rule in line 5 (Figure 4.2).

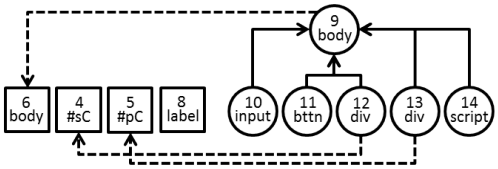


Figure 4.2: Data (dashed lines) and structural (full lines) dependencies between h-vertices and c-vertices, lines 1–14, Listing 4.1

Interpreting JavaScript code

Algorithm 2 shows how dependencies are created at runtime, when evaluating JavaScript code. The process evaluates the AST nodes in the source code of the script element according to the rules of JavaScript interpretation (line 3, *getNextNode* selects the next AST node for evaluation). Then, the AST node is evaluated (line 4) and the matching

Algorithm 2 Interpreting JavaScript

```
1: function INTERPRETJS(scriptNode)
2:     programAST ← getAST(scriptNode)
3:     while astNd ← getNextNode(programAST) do
4:         evalRes ← evalute(astNd)
5:         jVrtx ← getVertex(astNd)
6:         if isInLoopOrBranchStatement(astNd) then
7:             addControlDep(jVrtx, getParentConditionVertex(astNd))
8:         else if isEnteringCatchStatement() then
9:             addControlDep(jVrtx, getErrorThrowingVertex())
10:        else if isEnteringEventHandler() then
11:            addControlDep(jVrtx, getEventRegVertex())
12:        else if isEnteringFunction(astNd) then
13:            addControlDep(jVrtx, getCallExpVertex())
14:        end if
15:        if isBreakContinueOrReturn(astNd) then
16:            addControlDep(getImportantParent(astNd), jVrtx)
17:        end if
18:        if isAccessingIdentifier(astNd) then
19:            addDataDep(jVrtx, getLastAssignVertex(astNd))
20:        else if isEvaluatingComplexExpression(astNd) then
21:            addDataDep(jVrtx, getChildVertices(astNd))
22:        end if
23:        if isCreatingJsCode(astNd) then
24:            parseAddedCodeCreateASTNodes(astNd)
25:        else if isCreatingHtmlOrCss(evalRes) then
26:            addDataDep(createDynamicVertices(evalRes), jVrtx)
27:            traverseCssRulesAndCreateDeps()
28:        end if
29:        if isModifyingDOM(evalRes) then
30:            modifiedVertices ← getModifVertices(evalRes)
31:            addDataDep(modifiedVertices, jVrtx))
32:            traverseCssRulesAndCreateDeps(modifiedVertices)
33:        else if isSendingRequest(evalRes) then
34:            addDataDep(jVrtx, getOpenConnectionVertex(evalRes))
35:        else if isAccessingResponse(evalRes) then
36:            addDataDep(jVrtx, getSendRequestVertex(evalRes))
37:        end if
38:    end while
39: end function
```

j-vertex is obtained (line 5). If the evaluated AST node is in a loop or a branch, a control dependency from the current j-vertex to the condition expression j-vertex (line 7) is created. If the evaluated expression is in a catch expression, then a control dependency towards the vertex that matches the expression causing the exception (line 8) is created.

```
...
/*11*/ <input id="input"/>
...
/*15*/<script>
/*16*/  var number;
/*17*/  function add(a, b) { return a + b; }
/*18*/  var inputCont = document.querySelector("#input")
...
/*23*/  inputCont.oninput = function(){
/*24*/    number = parseFloat(inputCont.value);
/*25*/  };
...
```

Listing 4.3: Excerpt from Listing 4.1

Example. In the example, lines 16 and 17, Listing 4.3 have no dependencies. In line 18, the call expression calls the API method that returns the element with an id *input*, and a data-dependency from the call expression j-vertex to the HTML node h-vertex in line 11 is created. Also, since this is an internal method call, a data-dependency from the call expression j-vertex to the string literal "#input" j-vertex is created. Next, since the value of the call expression is assigned to a variable, a data-dependency from the variable declaration expression to the call expression is created. The algorithm is similar for lines 19, 20, 21. In line 23, the variable *inputCont* from *inputCont.oninput* obtains its value from the variable declared in line 18, and a matching data-dependency is created. The assignment expression in the same line assigns a function to an HTML object property, and two data-dependencies from the HTML node h-vertex in line 11 are created: one towards the function expression j-vertex and the other towards the assignment expression in line 23 j-vertex (Figure 4.3). The process is similar for line 27.

If the process is currently handling an event (line 10, Algorithm 2), then a control dependency towards the event registering expression j-vertex is created. When executing function code (line 12), a dependency is created towards the call expression causing the function execution. If the currently evaluated expression is reading an identifier, then a data dependency is created towards the last expression assigning the value of

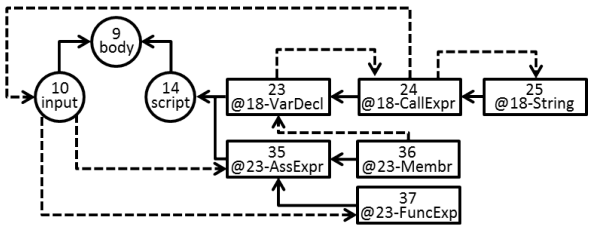


Figure 4.3: Data (dashed lines) and structural (full lines) dependencies between h-vertices and j-vertices, lines 11, 18, 23, Listing 4.1

that identifier (line 19). If the currently evaluated expression is complex (e.g. binary, unary, call expression), then a data-dependency towards j-vertices that represent its children are created (e.g. from a binary expression to its left and right child expressions), line 21.

Example. The scenario is composed of two events: the input of value 2 in the HTML input element, line 11, Listing 4.1 and the click on the button in line 12. The first event causes the execution of the event-handler in lines 23 – 25. First, a control-dependency from the function expression j-vertex towards the assignment expression j-vertex from line 23 is created, since this is where the event was registered. Next, the evaluation of the member expression (*inputCont.value*) causes the creation of a data-dependency towards the variable declaration j-vertex in line 18. The *parseFloat* call expression matches an internal method, and a data-dependency towards the member expression argument j-vertex is created. The identifier *number* on the right-hand side is declared at line 16, and a matching data-dependency is created alongside a data-dependency towards the *parseFloat* call expression j-vertex (Figure 4.4). The process continues by handling the click event with a handler in lines 27 – 36, and proceeds in manner similar to the previously described steps up until line 34.

Next, the algorithm is dealing with cases when the evaluation of JavaScript code creates new nodes (lines 23 – 28, Algorithm 2). If the evaluation of JavaScript code is creating new JavaScript code (by using the *eval* function), then the string passed to the *eval* function is parsed and new dynamic j-vertex nodes are created (line 24). If the JavaScript code is creating new HTML or CSS nodes (e.g. by writing to the *innerHTML* property, by calling the *createElement* function of the global

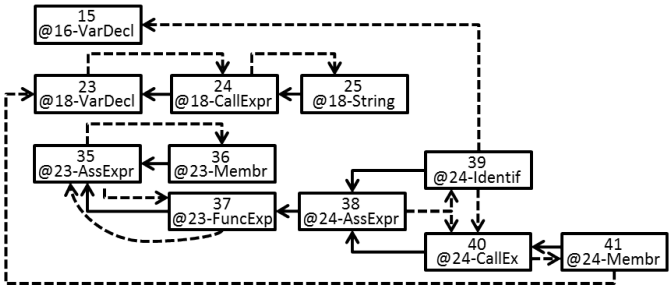


Figure 4.4: Data (straight dashed lines) and control (arched dashed lines) dependencies between h-vertices and j-vertices, for line 24, Listing 4.1

document object, or by calling the *addRule* or *insertRule* method of the stylesheet object), then the matching dynamic vertices are created with a dependency towards the j-vertex that matches the expression causing the creation (line 26). The created dynamic vertices can be of any type (because the *innerHTML* property can be used to create any type of code). Since new HTML and CSS nodes are created, we have to establish new dependencies. For this reason the *traverseCssRulesAndCreateDeps* function is called.

```
...
/*7*/ label {font-weight: bold;}
...
/*13*/ <div id="sCont"></div>
...
/*34*/ sCont.innterHTML = "<label>Sum:" + sum + "</label>"
...
```

Listing 4.4: Excerpt from Listing 4.1

Example. In line 34, Listing 4.4 a new HTML element is created by assigning an HTML string to the *innerHTML* property of an HTML element object. This causes the creation of a new h-vertex with a data-dependency towards a j-vertex matching the assignment expression in line 34 and a structural dependency towards the h-vertex matching the HTML element from line 13. Since a new HTML element is created, all CSS rules are traversed, and a data-dependency from the newly created h-vertex to the c-vertex matching the CSS rule in line 7 is created (Figure 4.5).

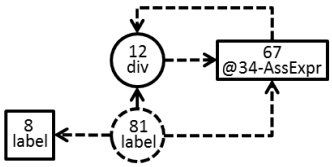


Figure 4.5: Data-dependencies between h-vertices and j-vertices, for line 34, Listing 4.1

The algorithm continues by creating dependencies caused by UI modifications and server-side communications (lines 29 – 37, Algorithm 2). If the DOM of the page is modified, a dependency from all impacted nodes to the j-vertex matching the expression causing the modification is created (line 31). Since the DOM of the page has changed, different CSS rules could be applied to different nodes, so the *traverseCssRulesAndCreateDeps* function is called. Next, the algorithm deals with server-side communications. If the JavaScript expression is sending a request, then a data dependency towards a j-vertex representing the expression that opens the connection is created (line 34), and if the JavaScript expression is accessing a server-side response, a control dependency to the j-vertex matching the expression sending the request is created (line 36).

Example. In line 34, Listing 4.4, alongside the creation of a new HTML element, the content of an existing HTML element from line 13 is also modified. This constitutes a UI modification, and a data-dependency from the h-vertex matching the HTML element from line 13 towards the j-vertex matching the assignment expression in line 34 is created (Figure 4.5). The process is similar for line 35.

All dependencies created when evaluating an expression are labeled with the current evaluation position, in order to differentiate between dependencies created on different function calls and loop executions.

```
...
/*17*/ function add(a, b) { return a + b;}
...
/*29*/ var product = add(0, 1);
/*30*/ for (var i = 0; i <= number; i++) {
/*31*/  sum = add(sum, i);
```

Listing 4.5: Excerpt from Listing 4.1

Example. Figure 4.6 shows the data-dependencies created when executing the call expression in line 29 (Listing 4.5). The numbers near

each arc represent arc labels. In this example, by using this labeling, we can accurately follow arcs created in different contexts, e.g. arcs created due to call expressions in line 29 in contrast to the arcs created when executing the call expression in line 31.

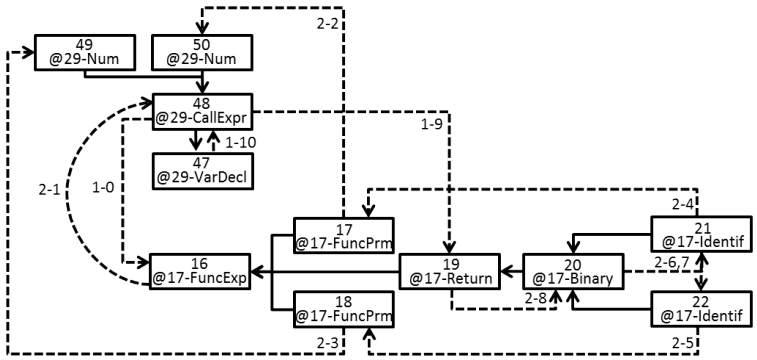


Figure 4.6: Dependencies for the function call in line 29, Listing 4.1

4.3 Conclusion

In this chapter, we have defined a client-side dependency graph that captures all dependencies that exist in a client-side application, for a particular scenario. The graph is composed of four types of vertices: HTML vertices, CSS vertices, JavaScript vertices, and resource vertices; and three types of arcs: structural dependency arcs, data flow arcs, and control flow arcs. Each vertex can either be static, which means that it is directly present in the source code, or dynamic if the matching code construct is dynamically created by evaluating JavaScript code. Since the graph captures dynamic dependencies, each arc is uniquely labeled, which enables differentiation of arcs created in different execution contexts. We have also presented algorithms that describe how the graph is constructed during the web application execution.

In the following chapters, we show how the dependency graph is used to generate feature scenarios, identify feature code, and facilitate reuse.

Chapter 5

Automatic Scenario Generation

The method for automatic feature reuse relies on the analysis of application execution caused by certain scenarios; scenarios that either capture the behavior of the whole application or cause the manifestation of a particular feature. Specifying such scenarios, for most features, is fairly straightforward, but in some cases it can be time-consuming and require in-depth knowledge of the internal details of the target application. For this reason, a method for automatic generation of scenarios would be beneficial. In this chapter, we introduce such a method based on dynamic analysis and systematic exploration of the application's event and value space.

5.1 Overview

The goal of the process is to generate scenarios that capture either the behavior of the whole application or the behavior of a particular feature. The process is composed of two phases: *i) Scenario Generation* and *ii) Scenario Filtering* (Figure 5.1).

The first phase – *Scenario Generation* – starts by creating an initial scenario that represents the process of loading the page in a default browser, without providing any user input. The approach then proceeds by iteratively selecting a scenario, executing it and dynamically analyz-

ing the execution. New scenarios are generated by: *i)* extending event chains – all event registrations and data-dependencies during scenario execution are tracked, and new scenarios are generated by extending the event chain of the current scenario with newly registered events, or with previously executed events whose execution potentially depends on the variables and objects modified by the current scenario; and by *i)* modifying the input parameters, i.e. the internal state of the browser (e.g. the initial window size, the type of the browser) and the event parameters – the process tracks how the input parameters influence the control-flow of the application, and generates new scenarios by modifying those inputs. New scenarios are created and analyzed until a certain coverage is achieved, a given time-budget expended, or a target number of scenarios reached.

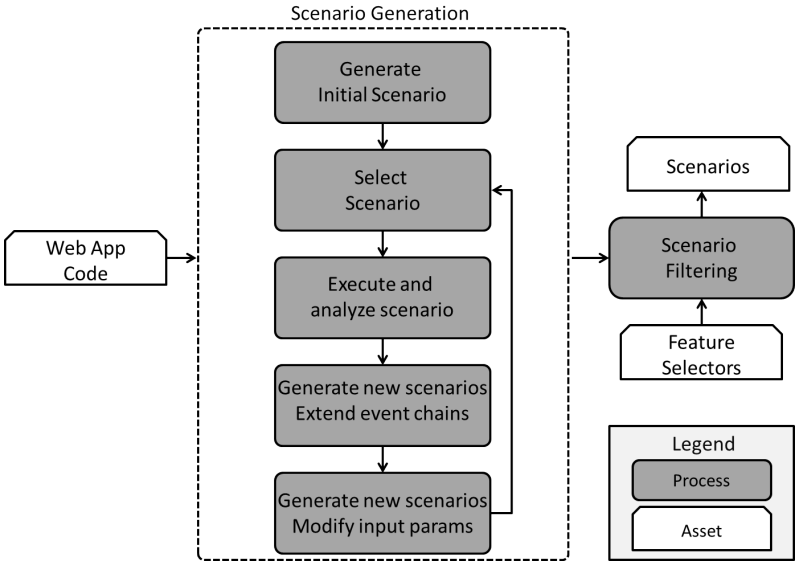


Figure 5.1: The process of generating feature scenarios

In the second phase – *Scenario Filtering* – execution traces of all executed scenarios are analyzed, and the set of scenarios is filtered. If the process targets certain application features, all scenarios that do not cause the manifestation of those features are removed. In addition, we also remove scenarios whose removal does not lower the overall coverage.

5.1.1 Terminology

An event e is defined as a tuple $e = \langle o, t \rangle$, where o is the ID of the object on which the event occurs (an HTML node, the global window, or the global document object), and where t is an event type. At run-time, when an event occurs, it is parametrized with properties of three different types [5]: *i)* event properties, a map from strings (property names) to numbers, booleans, strings and DOM nodes (e.g. the *which* property of the mouse click event identifies the clicked mouse button), *ii)* form properties, which provide string values for the HTML form fields (e.g. e-mail formatted string for an HTML input element), and *iii)* the execution environment properties, which represent values for the browser's state (e.g. window size) that can be influenced by the user. A parametrized event e^p consists of an event e and parameters p associated with that event. The goal of the process is to compute a set $S = \{s_0, s_1, ..., s_n\}$ of scenarios that achieves high code coverage. A scenario s_i is defined as a sequence of parametrized events $s_i = \langle {e^p}_0, {e^p}_1, ..., {e^p}_m \rangle$. Each scenario exercises a certain subset of the application behavior.

During scenario execution, application execution can be influenced by the internal state of the browser. In order to represent this, we use one special event [5]: ${e_0}^d = \langle window, \text{“}main\text{”} \rangle$ that denotes the loading of the target web page. This event has properties that define the internal state of the brower, e.g. the initial window size, type of the browser, cookie string. By modifying these properties, we can simulate different browser configurations that can influence the control-flow of the application.

We can also establish a relationship between an application feature and a scenario: a scenario causes the manifestation of a particular feature if its last parametrized event ${e^p}_m$, in some way, affects the parts of the page structure where the feature manifests. A parametrized event affects a part of the page structure if: *i)* it is called on an HTML node that is a part of the structure; *ii)* it modifies the structure; *iii)* in the case of server-side communication events, if there is a data dependency from the message-sending request to the structure. If ${e^p}_m$ affects the structure, this means that all previous parametrized events ${e^p}_j; j < m$ are also important from the perspective of the structure, because event chains are created only out of interdependent events (described in the following sections).

5.2 Detailed process description

The process described in Section 5.1 is in more details presented by Algorithm 3.

Algorithm 3 generateScenarios(*webAppCode*, *selectors*)

```
1:  S ← emptyArray
2:  S_A ← emptyArray
3:  M_E ← emptyMap
4:  s_0 ← emptyScenario
5:  s_0 ← appendEventToScenario(s_0, createDefaultLoadingEvent())
6:  S ←push(S, s_0)
7:  do
8:      s ← selectScenario(S)
9:      if s == null then
10:         break
11:     end if
12:     executionInfo ← executeScenario(s)
13:     associate(M_E, s, executionInfo)
14:     recalculateCoverage(webAppCode, executionInfo)
15:     if hasAchievedTargetCoverage(webAppCode) then
16:         break
17:     end if
18:     S ← createByExtendingEventChains(s, S, executionInfo, M_E)
19:     S ← createByModifyingInputParameters(s, S, executionInfo)
20:     push(S_A, s)
21: while |S_A| < LIMIT and hasTime()
22: S_A ← filterScenarios(S_A, selectors, M_E)
23: return S_A
```

The first phase – *Scenario Generation* (lines 1 – 21) starts by creating two empty arrays S and S_A, where S will hold all scenarios generated by the process, and S_A all scenarios analyzed in the process; and M_E, a map that connects scenarios and their execution info. Initially, the process creates one empty scenario s_0 (line 4) and adds to it a default loading event (line 5). Next, the process iteratively selects the next scenario for analysis (line 8), executes it (line 12), creates a mapping between the executed scenario and the execution info (line 12), and generates new scenarios by extending event chains (line 18) and modifying

input parameters (line 19). The process exits the first phase if there are no remaining scenarios for analysis (lines 9, 10), the target coverage is achieved (lines 15, 16), the target number of scenarios analyzed, or a given time budget expended (line 21). After the first phase is complete, the process enters the second phase – *Scenario Filtering* (line 12), where the analyzed scenarios that do not contribute to the selected feature and the scenarios that do not lower the overall coverage are removed.

5.2.1 Example application

Throughout this chapter we give a detailed description of how new scenarios are created, and for this we will use the example from Listing 5.1.

```
1  <html><head>
2   <style>
3    .c{ width: 100px; height: 100px;}
4    #fc{background:rgb(255,0,0);}
5    #sc{background:rgb(0,0,255);}
6   </style></head>
7  <body>
8   <div id="fc" class="c"></div><div id="sc" class="c"></div>
9   <script>
10   var fc = document.getElementById("fc");
11   var sc = document.getElementById("sc");
12
13   var clicks = 0;
14   fc.onmousedown = function(e) {
15    if(e.which == 1)
16     fc.onmousemove = function(e) {
17      var val = e.pageX % 256;
18      this.style.background="rgb("+val+","+val+","+val+")";
19     }
20    else if(e.which == 2)
21      if(++clicks % 2 == 0)
22       this.textContent = "Even";
23      else
24       this.textContent = "Odd";
25   }
26   sc.onclick = function(e) {
27    this.textContent = e.pageX + ";" + e.pageY;
28   }
29  </script>
30 </body>
31 </html>
```

Listing 5.1: Example application

The UI of the application is composed of two squares, one corresponding to the div element with the ID *fc*, and the other to the div element with the ID *sc*, line 8, Listing 5.1. The example application has two features: Feature 1, that manifests on the first square (the HTML node with ID *fc*, line 8), which consists of two behaviors: *i)* when the user clicks on the square with the left mouse button, the application subscribes to the mouse move events which change the color of the first square background depending on the position of the mouse, *ii)* it counts the number of middle mouse button clicks on the first square, and outputs whether this number is even or odd; and Feature 2, that manifests on the second square (the HTML node with ID *sc*, line 8), with the following behavior: *i)* when the user clicks on the second square it outputs the current mouse position. This is an example of an event-driven application where code coverage depends both on the events raised by the user, and the properties of the raised events (e.g. which mouse button was clicked). Throughout this section, we will show how the process generates scenarios that target the first feature.

5.2.2 Selecting Scenarios

Our method creates new scenarios by systematically exploring the event and value space of the application. This means that the number of generated scenarios grows considerably with application complexity, and the procedure by which the next scenario for analysis is picked could influence how fast the scenario generation method achieves good coverage. For this reason, we have considered several prioritization functions that determine the order in which the scenarios will be analyzed. The prioritization functions are the following: *i) FIFO* (First In, First Out), where scenarios are picked based on their order of creation; *ii) Random*, where the next scenario is randomly picked from the pool of non analyzed scenarios; *iii) Event Length*, where the next scenario for analysis is the scenario with the shortest sequence of events. We also use two more complex prioritization functions: *iv) Coverage* and a *v) Custom* prioritization function.

The *Coverage* prioritization function is based on the intuition that executing scenarios with events that have already achieved high code coverage is likely to be less useful than executing scenarios with events with low coverage [5]. After the execution of every scenario, for every function visited during the evaluation of each event e_i, we recalculate

the coverage achieved so far. Based on this coverage, each scenario is assigned a weight (or a priority). We use the following formula [5]:

$$P(s_i) = 1 - cov(e_0) \cdot cov(e_1) \cdot ... \cdot cov(e_m),$$

where $s_i = \langle e_0, e_1, ..., e_m \rangle$ is the scenario whose weight is being determined, $e_0, e_1, ..., e_m$ the event chain of s_i, and $cov(e_j)$ a function that calculates statement coverage of all functions visited during the execution of an event achieved so far by the whole process (i.e. by all previously analyzed scenarios) in all functions executed during the handling of event e_j. The next scenario for analysis is then chosen with weighted random selection.

The *Custom* prioritization function is based on the following intuition: if there is an unexecuted scenario created by exploring the value space (i.e. modifying the input parameters), or a scenario whose last event has not yet been executed in the scenario generation process, then the process selects such a scenario based on the order of creation. If there are no such scenarios, i.e. only the scenarios created by extending the event chain with events already executed in the process, then select the next scenario by using the *Coverage* prioritization function.

5.2.3 Scenario Execution

As discussed for the dependency graph creation (Chapter 4), the dynamic nature of web applications means that the scenario generation process must be based on dynamic rather than static analysis. For this reason, we execute and analyze each scenario. The scenario execution is performed with our custom-made browser simulator (Chapter 8) which has a JavaScript interpreter capable of performing both concrete and symbolic execution of web applications, and building a dependency graph (Chapter 4). On top of this, the browser simulator can be instantiated with different internal browser states, in that way mimicking different browsers.

Example. The first scenario being executed is the scenario that contains only the default loading event: $s_o = \langle {e_0}^d \rangle$, where ${e_0}^d$ is the default loading event. In the example from Listing 5.1, this means constructing the DOM of the page (based on lines 1 – 9) and executing JavaScript statements in lines 10 – 14, 26. At this point, no user-generated events are executed. The loading of the example page does not depend on the internal browser state. After the page is loaded, the browser is aware of

two event registered event handlers (*onmousedown* in line 14 and *onclick* in line 26). This information will be used by the process to generate new scenarios.

5.2.4 Extending event chains

When generating scenarios by exploring the event space the goal is to extend event chains, either with events that are encountered for the first time by the scenario generation process, or with already executed events that are still registered at the end of scenario execution (Algorithm 4).

Algorithm 4 createByExtendingEventChains(s, S, *executionInfo*, M_E)

```
1:  g ← getDependencyGraph(executionInfo)
2:  for all e : getRegisteredEvents(executionInfo) do
3:      if wasInstanceExecuted(e, S) then
4:          for all e^p : getPreviousParametrizations(e, S) do
5:              if connectionExists(g, getExecutionInfo(M_E, e^p)) then
6:                  s_n ← createCopy(s)
7:                  s_n ← appendEventToScenario(s_n, e^p)
8:                  S ← appendScenario(S, s_n)
9:              end if
10:         end for
11:     else
12:         s_n ← createCopy(s)
13:         e^p ← parametrizeWithDefaults(e)
14:         s_n ← appendEventToScenario(s_n, e^p )
15:         S ← appendScenario(S, s_n)
16:     end if
17: end for
```

After the execution of a scenario, the process traverses all events that are still registered at the end of the execution. If the event was already executed (lines 3 – 10), i.e. at least one parametrization of that event already exists in previously analyzed scenarios, then execution logs for each event parametrization are traversed. During the execution of each scenario we build a dependency graph (Chapter 4) which captures both static and dynamic dependencies between code constructs. We consider that there is a potential connection between an event and a scenario (line 5) if the scenario modifies variables and/or objects on which the

control-flow of the event depends on (either directly or indirectly). If a connection exists, then a new scenario is created by appending the parametrized event to the current scenario. If the event has not yet been executed (lines 12 – 15), then the newly registered event is parametrized with default parameters, and a new scenario is created by appending the parametrized event to the events from the current scenario.

Example. In the example from Listing 5.2, after the execution of the initial s_0 scenario, the process is aware of two registered events (line 2, Algorithm 4): *onmousedown* in line 14 and *onclick* in line 26. Since s_0 is the first analyzed scenario, these events have not been executed so far, and the algorithm follows the procedure from lines 12 – 15. This causes the creation of two new scenarios. Based on the *onmousedown* event registration, the scenario $s_1 = \langle {e_0}^d, \langle \#fc, onmousedown \rangle^{\{which:1\}} \rangle$ (the default parameter, left button – is represented by the value 1 of the *which* property), and based on the *onclick* mouse registration from line 26 the scenario $s_2 = \langle {e_0}^d, \langle \#sc, onclick \rangle^{\{pageX:50, pageY:150\}} \rangle$ (the default behavior is to click in the middle of the target element).

```
/*13*/  var clicks = 0;
/*14*/  fc.onmousedown = function(e) {
/*15*/   if(e.which == 1)
/*16*/    fc.onmousemove = function(e) { ...}
/*20*/   else if(e.which == 2)
/*21*/     if(++clicks % 2 == 0)
/*22*/      this.textContent = "Even";
/*23*/     else
/*24*/      this.textContent = "Odd";
/*25*/  }
/*26*/  sc.onclick = function(e) { ... }
```

Listing 5.2: Excerpt from Listing 5.1

When analyzing the execution of scenario s_1, a new event, which has not been encountered so far, is registered in line 16. This leads to the creation of a new scenario: $s_3 = \langle {e_0}^d, \langle \#fc, onmousedown^{\{which:1\}} \rangle; \langle \#fc, onmousemove \rangle^{\{pageX:50, pageY:50\}} \rangle$. Next, imagine that a scenario $s_5 = \langle {e_0}^d, \langle \#fc, onmousedown \rangle^{\{which:2\}} \rangle$, generated by the other part of the process (will be described in the following section) is analyzed. When analyzing the execution of s_5, we can see that its *onmousedown* event, writes to the variable *clicks*, created outside of the event context (line 21). That same variable influences the control flow of the

event (there exists a data dependency from the variable *clicks* to the if statement condition) – s_5 is dependent on itself, and a new scenario $s_6 = \langle {e_0}^d, \langle \#container, onmousedown \rangle^{\{which:2\}}, \langle \#container, onmousedown \rangle^{\{which:2\}} \rangle$ is created.

5.2.5 Modifying input parameters

In order to generate scenarios by modifying input parameters, we use concolic testing [26, 54]. The main idea is to execute the scenario both with concrete (e.g. default values for the initially created scenarios) and symbolic values for input parameters. The input parameters encompass both the variables that describe the internal state of the browser and the event parameters. While the application is executed, all expression are evaluated both concretely and symbolically. During the execution all encountered control-flow branches (e.g. if statements, conditional expressions) whose branching conditions are expressions that contain symbolic variables are added to the so called path-constraint, which carries information about how the control-flow of the execution depends on the input parameters. In order to build a scenario that exercises another path through the application we have to modify the input parameters based on the path constraint. This is usually done by systematically dropping and negating the constraints that compose the path-constraint, and in our approach we use generational search [27]. Constraints obtained in this way are solved with a constraint solver, which gives new input parameter values that exercise different execution paths. Currently we are using an off-the-shelf constraint solver – Choco [35].

Algorithm 5 createByModifyingInputParemeters(s, S, *executionInfo*)

```
1: pathConstraint ← getPathConstraint(executionInfo)
2: for all invertedFormula : getInvertedFormulas(pathConstraint) do
3:     result ← solveFormula(invertedFormula)
4:     if result ≠ null then
5:         ⟨e_0, ..., e_n⟩ ← getAffectedEvents(s, result)
6:         ⟨e^p_0, ..., e^p_n⟩ ← parametrizeEvents(⟨e_0, ..., e_n⟩, result)
7:         s_n ← createScenario(⟨e^p_0, ..., e^p_n⟩)
8:         S ← appendScenario(S, s_n)
9:     end if
10: end for
```

Handling parameter domains – In addition to the constraints gathered during concolic execution, some of the event parameters always fall into a certain domain (e.g. the *which* property of the mouse event handler can have only three values: 1, 2, or 3; or the mouse position parameters, such as pageX and pageY, are constrained by the position of the element the event occurs upon). For this reason, when constructing constraints that will be sent to the solver, constraints that capture the domain of each parameter are also added.

Handling the internal browser state – the execution of a scenario can be significantly influenced by the internal state of the browser. This state encompasses properties such as cookies, initial size of the window, the user agent string, etc.; but it also includes the internal browser objects that can be browser specific. By studying the control-flow dependencies towards internal browser objects, constraints that capture the type of browser can be derived. This leads to the creation of browser-specific scenarios. In our case, this is not a problem, since we use a custom-made JavaScript interpreter (Chapter 8) that can mimic different browsers.

```
/*14*/  fc.onmousedown = function(e) {
/*15*/   if(e.which == 1)
/*16*/    ...
/*20*/   else if(e.which == 2)
```

Listing 5.3: Excerpt from Listing 5.1

Example. After the execution of the s_1, we study the path constraint obtained from the if statement in line 15, Listing 5.3: $which = 1$. In order to cover another execution path, we invert that constraint and obtain – $which \neq 1$, and add the constraints inherent to the *which* property: $which = 1, which = 2, which = 3$. For these constraints, we imagine that the constraint solver obtains the result $which = 3$. This causes the creation of a new scenario $s_4 = \langle {e_0}^d, \langle \#fc, onmousedown \rangle^{\{which:3\}} \rangle$. When we execute the scenario s_4 the resulting path constraint is $which \neq 1, which \neq 2$, because both the condition of the if statement in line 15, and the condition of the if statement in line 20 were evaluated to false. By inverting these constraints we obtain: $which \neq 1, which = 2$ and $which = 1$. With the constraint solver we get two solutions: $which = 2$ and $which = 1$. The solution $which = 1$ is discarded since the scenario with the exact parameters already exists, and out of $which = 2$ we obtain

a new scenario $s_5 = \langle {e_0}^d, \langle \#fc, onmousedown \rangle^{\{which:2\}} \rangle$.

5.2.6 Filtering Scenarios

In order to achieve high coverage, the process generates a number of scenarios. However, we are typically interested in obtaining a minimal number of scenarios that still achieve the same coverage. The main purpose part of the process is to remove scenarios that are not necessary for the goal of scenario generation.

Algorithm 6 filterScenarios(S, *selectors*, M_E)

```
1: if targetsFeature(selectors) then
2:     for all s ∈ S do
3:         if notRelatedToFeature(s, selectors, M_E) then
4:             S ← removeScenario(S, s)
5:         end if
6:     end for
7: end if
8: jointCoverage ← getJointCoverage(S, M_E)
9: for all s ∈ sortDescendingByNoOfEvents(S) do
10:    if canScenarioBeRemoved(s, jointCoverage) then
11:        jointCoverage ← removeScenarioCoverage(jointCoverage, s))
12:        S ← removeScenario(S, s)
13:    end if
14: end for
```

If the process of automatic scenario generation is executed with the goal of generating scenarios that cause the manifestation of certain application features (instead of targeting the whole application behavior), then for every executed scenario, the process checks whether the scenario is related to the specified parts of page structure where the feature manifests (Chapter 5.1.1) – if it is not, the scenario is filtered away. The process then calculates joint scenario coverage, which is a map that shows, for each code expression, how many scenarios have executed that expression. Then, all scenarios are traversed in descending order, starting from the scenario with the longest event chain. For each scenario, the algorithm checks whether the joint coverage would remain the same if that scenario was removed. If so, the scenario is removed from the set of scenarios, and its coverage from *jointCoverage*.

Example. In the example application, the scenario generation phase has generated the following six scenarios:

- $s_0 = \langle {e_0}^d \rangle$;
  $cov_0 = \{10 - 14, 26\}$

- $s_1 = \langle {e_0}^d, \langle \#fc, onmousedown \rangle^{\{which:1\}} \rangle$;
  $cov_0 = \{10 - 16, 26\}$

- $s_2 = \langle {e_0}^d, \langle \#sc, onclick \rangle^{\{pageX:50, pageY:150\}} \rangle$;
  $cov_1 = \{10 - 14, 26, 27\}$

- $s_3 = \langle {e_0}^d, \langle \#fc, onmousedown \rangle^{\{which:1\}}$;
  $\langle \#fc, onmousemove \rangle^{\{pageX:50, pageY:50\}} \rangle$;
  $cov_4 = \{10 - 18, 26\}$

- $s_4 = \langle {e_0}^d, \langle \#fc, onmousedown \rangle^{\{which:3\}} \rangle$;
  $cov_2 = \{10 - 15, 20, 26\}$

- $s_5 = \langle {e_0}^d, \langle \#fc, onmousedown \rangle^{\{which:2\}} \rangle$;
  $cov_3 = \{10 - 15, 20, 21, 22, 26\}$

- $s_6 = \langle {e_0}^d, \langle \#fc, onmousedown \rangle^{\{which:2\}}$;
  $\langle \#fc, onmousedown \rangle^{\{which:2\}} \rangle$;
  $cov_5 = \{10 - 15, 20, 21, 22, 24, 26\}$

First, all scenarios are traversed in order to remove the ones that do not contribute to the feature. In this case, this means the removal of scenario s_2 because it neither occurs on, nor does it modify the specified parts of the web application structure ($\#fc$). Next, a joint coverage for the remaining scenarios is calculated. Here, we will discuss in terms of code lines, but the algorithm in general works on AST nodes. Joint coverage, from the perspective of executed lines, for the remaining scenarios $s_0, s_1, s_3, s_4, s_5, s_6$ is: 10–14: 6, 15: 5, 16: 2, 17–18: 1, 20: 3, 21: 2, 22: 2, 24: 1, 26: 6. The first processed scenario, s_6 can not be removed because it is the only scenario that executes line 24, while scenarios s_5 and s_4 can be removed, because their lines are executed by at least one other scenario. Scenario s_3 can not be removed because no other scenario executes lines 17 and 18, while scenarios s_1 and s_0 can be removed.

5.3 Evaluation

The evaluation of the automatic scenario generation technique is based on four experiments. In the first experiment, we study the coverage the process was able to achieve, and the number of generated scenarios and their events, when generating test cases for a suite of web applications. In the second experiment, we compare the results of our method, with the results obtained with a related method – Artemis [5]. For the third experiment, the goal was to compare the effectiveness of different prioritization functions, in terms of achieved statement coverage. Finally, in the forth experiment, we study how the process is able to generate feature scenarios for a case study application. All results were obtained with the Firecrow tool[1] (Chapter 8) which implements the algorithms described in this chapter. The code of all web applications, the generated scenarios, and other experiment results can be obtained from: `www.fesb.hr/~jomaras/download/usageScenarioGenerator.zip`.

5.3.1 Generating scenarios for the whole page

The first experiment was performed on a suite of web applications, most of them obtained from the 10k and 1k JavaScript challenges[1].

The goal of the first experiment was to compare the statement coverage achieved by simply loading the page, with the coverage achieved by using our method. Table 5.1 shows the results. For each application it shows: the lines of code (LOC); the statement coverage that can be achieved by simply loading the page (L-Cov); the maximum statement coverage the generated scenarios were able to achieve (A-Cov); the gain of our method, when compared to simply loading the page; the number of kept scenarios (S); and the total number of events in the kept scenarios (E). On average, the process was able to achieve additional 40 percent statement coverage when compared to the coverage achieved by loading the page. In order to cover these additional statements, our process, on average, generates 9 scenarios with 19 events.

[1] https://github.com/jomaras/Firecrow

[1] `http://10k.aneventapart.com/` and `http://js1k.com/`

Table 5.1: Experiment results for analyzing 100 scenarios for the whole application. LOC (Lines of Code), Cov (statement Coverage), L (Loading), A (Achieved), S (Scenarios), E (Events)

Application	LOC	L-Cov	A-Cov	Gain	S	E
ajaxtabscontent	238	66%	91%	25%	7	7
angelJump	313	66%	71%	5%	2	11
minesweeper	177	60%	94%	34%	4	7
prism3D	402	61%	91%	30%	15	33
snake	211	58%	99%	41%	6	13
dynamicArticles	178	35%	85%	50%	3	6
fractalViewer	1630	36%	77%	41%	14	42
rentingAgency	297	41%	100%	59%	14	15
tinySlider	128	47%	76%	29%	4	5
snowpar	353	20%	99%	79%	24	32
floatwar	458	18%	67%	49%	7	39
Average	398.6	46%	86%	40.1%	9	19

5.3.2 Comparison with Artemis

For the second experiment, we have compared our method with an already existing method – Artemis [5], in terms of maximum coverage the methods were able to achieve. We have used the same web application suite as in the experiments described in [5] (and we have excluded the same libraries from the results, e.g. jQuery), except for one application (we have excluded the *AjaxPoll* application because the application functionality is dependent on the state of the server, which is a feature that our approach does not yet take into account).

Table 5.2 shows the experiment results. The gains achieved by our method range from 0 – 9 percent, with an average of 3.1 percent. The gains are larger for applications in which input parameter values significantly influence the achieved coverage (e.g. if it is important which mouse button or keys were pressed), while they are smaller (or almost non-existent) in other types of applications. Our method is able to achieve these improvements because we rely on symbolic execution, and are able to generate values of event parameters that change the control-flow of the application.

In the experiment description, Artzi et al., mention that generating

Table 5.2: Comparison of the experiment results between Artemis and our approach (Firecrow)

Application	Artemis	Firecrow	Gain
3dModeller	74%	82%	8%
ajaxtabscontent	89%	91%	2%
ball_pool	90%	99%	9%
dragable-boxes	62%	63%	1%
dynamicArticles	82%	85%	3%
fractalViewer	75%	77%	2%
homeostasis	63%	63%	0%
htmlEdit	63%	63%	0%
pacman	44%	47%	3%
Average	71.3%	82.6%	3.1%

tests for a single application does not take more than 2 minutes on an average PC (but they do not present concrete numbers). In our case (Table 5.3), the time required to generate tests for a single application is, on average, 12 minutes. This happens because the backbone of their approach is the WebKit[1] browser engine, while we use our own JavaScript interpreter (Chapter 8), written in JavaScript, which is not optimized for speed. We do this, because our approach uses tracking of symbolic expressions and the dependency graph to generate scenarios, and this information is obtained with the custom JavaScript interpreter. However, the techniques and algorithms used in our approach could be adapted to WebKit, Firefox, or Chrome, if these browsers would support tracking of symbolic expressions and the generation of dependency graphs. This would considerably reduce the time required to generate web application scenarios. This is part of our future work.

5.3.3 Evaluating prioritization functions

For the third experiment, the goal was to compare the effectiveness of scenario prioritization functions (Chapter 5.2.2), with respect to the coverage they are able to achieve. For the application suite, we have used the applications from the first two experiments. The experiment results are show in Table 5.3.

[1] `http://www.webkit.org/`

Table 5.3: Experiment results for comparing prioritization functions by maximum achieved coverage; 100 analyzed scenarios; maximum coverage in bold

	EventLength	Fifo	Rand	Cov	Cust
Application	Maximum statement coverage				
3dModeller	**82%**	**82%**	**82%**	**82%**	**82%**
ajaxtabscontent	**91%**	**91%**	78%	89%	**91%**
ball_pool	98%	98%	**99%**	**99%**	98%
dragable-boxes	**63%**	**63%**	61%	59%	**63%**
dynamicArticles	**85%**	**85%**	**85%**	**85%**	**85%**
fractal_viewer	75%	75%	75%	**77%**	75%
homeostasis	**63%**	**63%**	**63%**	**63%**	**63%**
pacman	**47%**	**47%**	45%	45%	**47%**
htmlEdit	**63%**	**63%**	**63%**	**63%**	**63%**
3dMaker	83%	83%	82%	53%	**84%**
angelJump	69%	69%	69%	**71%**	69%
minesweeper	**94%**	**94%**	93%	**94%**	**94%**
prism3D	**91%**	**91%**	78%	82%	81%
rentingAgency	97%	97%	74%	98%	**100%**
snake	96%	**99%**	80%	90%	90%
snowpar	**99%**	**99%**	42%	**99%**	87%
tinySlider	**76%**	**76%**	**76%**	**76%**	**76%**
floatwar	**67%**	64%	64%	66%	64%
Type	Summary				
Avg coverage	80%	80%	73%	77%	78%
Avg #scenarios	11	11	9	10	10
Avg #events	18	17	17	20	18
Avg time (min)	12	12	18	9	13

In this experiment, on average, the maximum coverage is achieved with the *Event Length* and *Fifo* prioritization functions. They are followed by the *Custom* prioritization function, the *Coverage* function, and finally in the end, the *Random* function. For each prioritization function, we also present the average number of kept scenarios (Avg #scenarios), the average total number of events in kept scenarios (Avg #events), and

the average time in minutes that was required to generate the scenarios[2]. However, in order to generally claim that one prioritization function is better than the other, experiments on a larger suite of web applications would have to be performed. We consider this part of our future work.

5.3.4 Generating Feature Scenarios – a case study

Consider the example application shown in Figure 5.2 that enables the user to: *i)* toggle between different types of accommodation (by using the select menu marked with 1, or by pressing keyboard keys: e.g. A – Apartments, or H – hotels), *ii)* to select map locations (marked with 2) with mouse clicks which will change the information and photos displayed in the photos section (marked with 3); *iii)* to toggle between different photos (marked with 3) by clicking on buttons, or by pressing keyboard buttons (e.g. 1 for the first photo, 2 for the second photo); *iv)* to toggle between different county map zoom levels (marked with 4) by clicking on the county map; *v)* to automatically cycle between different event information (marked with 5).

Figure 5.2: Case study application

[2]Intel Xeon 3.7 Ghz, 16GB RAM

The example application has three distinct high-level features: *i)* selecting the map location and viewing its information (sections marked with 1, 2, and 3); *ii)* toggling between different county map zoom levels (marked with 4); and *iii)* viewing event information (marked with 5). Even in the case of these relatively simple features, specifying scenarios with high coverage is a time-consuming activity that requires in-depth knowledge of application behavior and the understanding of the underlying implementation. For example, a developer who wants to specify a scenario that exercises the complete behavior of the first feature has to be aware of different ways the location can be selected (by mouse clicking on the location point in the map, by changing the type of displayed locations through the select box, or by pressing keyboard keys), and of different ways the photos (marked with 3) can be toggled (either with mouse clicks on different buttons, or with keyboard presses).

Table 5.4: A case study of generating feature scenarios. S_G – generated scenarios, S_F – scenarios after filtering

Feature	$\lvert S_G \rvert$	$\lvert S_F \rvert$	Gen. events	User events
#1	25	12	12	12
#2	25	1	2	2
#3	25	1	1	1

We have initialized the process for each of the features, with the results shown in Table 5.4. For each feature, the process was able to achieve full coverage (in general this does not have to be the case), and it was successful in generating scenarios that target specific UI controls. The table shows how many scenarios the process generated in order to achieve full coverage (column $|S_G|$), how many scenarios were kept after the filtering process ($|S_F|$), and how many events in total the filtered scenarios have (Gen. events). The table also shows the minimum number of events, we were able to find, to achieve full coverage. In this application, the process was able to generate feature scenarios which in total have the minimal number of events we were able to determine by studying the application code. In general, since scenarios can be picked randomly from the set of generated scenarios, the generated sequences of events in all analyzed scenarios are not necessarily minimal.

5.4 Conclusion

Scenarios that execute application features with high coverage are used in many software engineering activities, such as testing or reuse. Manually specifying these scenarios is a time-consuming activity, and automation could bring considerable benefits. In this chapter, we have presented an automatic method for generating scenarios. The method works by systematically exploring the event and value space of the application. In order to create high-coverage scenarios we utilize techniques such as symbolic execution and dependency tracking. In order to reduce the number of generated scenarios, we analyze the relationships between the scenarios and features, and remove all non-related scenarios. We also filter scenarios based on their coverage.

The method was evaluated with four experiments. In the first experiment we compared the coverage achieved with the generated scenarios, with the scenario of simply loading the web page. The experiment has shown that, on average, a 40 percent increase in coverage can be achieved. For the second experiment, we have compared our method with a similar method for automatic testing (Artemis). The experiment has shown that, for certain kinds of applications, where coverage significantly depends on values of input parameters, our method is able to achieve better coverage, while in other types of applications, our method achieves results comparable to Artemis. For the third experiment, we have compared different scenario prioritization techniques. However, the experiment results are inconclusive, and while two of the prioritization functions achieve better results than the rest, the experiment would have to be repeated on a larger suite of web applications. In general, as part of our future work, we plan to perform similar experiments on a larger suite of web applications. Finally, we studied how the method was able to generate feature scenarios on a case study application and, in this experiment, the method was able to generate scenarios that cause the manifestation of particular features.

Chapter 6

Identifying Code of Individual Features

From the user's perspective, a client-side application offers a number of features that are relatively easy to distinguish. However, the same can not be said for their implementation details. A feature is implemented by a subset of the application's code and resources, and identifying the exact subset is a challenging task: code responsible for the desired feature is often intermixed with code irrelevant from the perspective of the feature, and there is no trivial mapping between the source code and the application displayed in the browser. The ability to exactly identify the code and resources of a particular feature is vital for performing reuse. In addition, a wide range of software engineering activities such as code understanding, debugging, and maintenance can be facilitated. In this chapter, we describe a method for automatic identification and extraction of code and resources that implement a particular feature.

6.1 Feature manifestations

Client-side applications act as user interfaces to server-side applications, and their two primary functions are: *i)* to communicate with the user through the UI of the application, and *ii)* to communicate with the server by exchanging messages. Each application offers a number of features (Figure 6.1). Since client-side applications are UI applications,

each feature is triggered by a sequence of user actions, i.e. a scenario, and a triggered feature manifests as a sequence of: *i)* UI modifications to the structure of the page, and/or *ii)* Server-side communications. These structural changes and server-side communications represent the feature behavior in a particular scenario and we refer to them as *Feature Manifestations*.

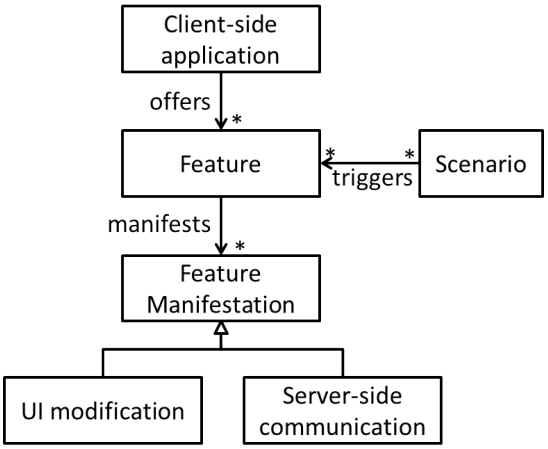


Figure 6.1: Feature manifestations

For example, Listing 6.1 shows a simple application with a feature that manifests when a user clicks on the part of the page structure defined with the *div* element in line 11. The feature manifests with one UI modification (line 15 – changing the background color of the *container* element), one server-side communication (line 18 – sending a synchronous message to the server), and one UI modification (line 19 – append text content to the *container* element).

```
1  <html>
2  <head>
3   <style>
4    #container {
5     width: 100px; height: 100px;
6     background-color: blue;
7    }
8   </style>
9  </head>
10 <body>
```

```
11   <div id="container"></div>
12   <script>
13    var container = document.getElementById("container");
14    container.onclick = function() {
15     container.style.backgroundColor = "red";
16     var httpRequest = new XMLHttpRequest();
17     httpRequest.open("GET", "serverSide.php", false);
18     httpRequest.send();
19     container.textContent += httpRequest.responseText;
20    };
21   </script>
22  </body>
23  </html>
```

Listing 6.1: An example of feature manifestations

As can be seen from the example in Listing 6.1, a feature manifestation matches an evaluation of a JavaScript expression executed when demonstrating a scenario, an evaluation that either modifies the structure of the page, or communicates with the server. Feature manifestations capture the essence of a feature in a particular scenario. One of the key insights that we use in this process is: in order to identify the code that implements a feature in a scenario, we have to identify code responsible for each feature manifestation.

6.2 Overview of the Identification process

A feature manifests when a user performs a certain scenario, and feature manifestations can only be determined dynamically. For this reason we base the approach on the dynamic analysis of application execution while feature scenarios are exercised. Scenarios are an integral part of our approach, and in the current process they have to be set up either manually by the user, or automatically generated with a scenario generation technique (Chapter 5).

In order to identify the implementation of a certain feature, we have to track dependencies between different parts of the application. For this reason, we use the *Client-side Dependency Graph* (Chapter 4), as the main artifact in the process. The overall *Feature Identification* process is shown in Figure 6.2. From the perspective of the whole reuse process (Chapter 3), the *Feature Identification* is the first step, and Figure 6.2 details the steps of the *Feature Identification* activity from Figure 3.2.

The *Feature Identification* consists of two phases: *Interpretation* and *Graph Marking*.

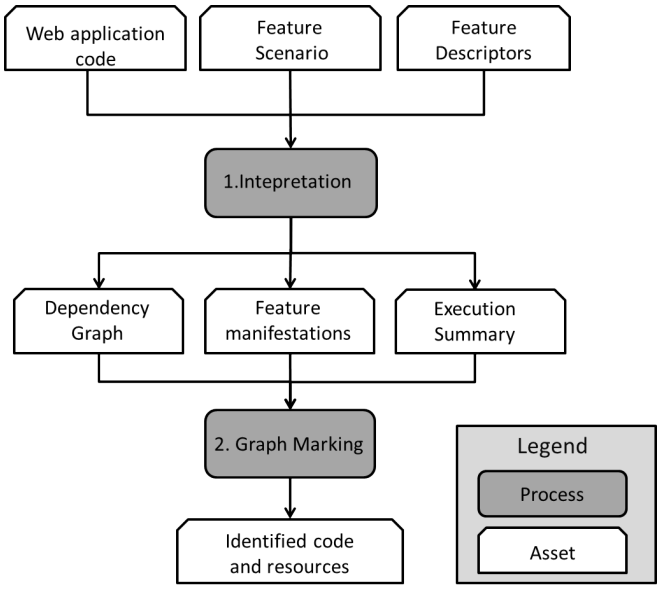


Figure 6.2: Identifying code and resources of a feature based on a scenario

Phase 1 – *Interpretation* – receives as input the whole web application code, a feature scenario that causes the manifestation of the desired feature, and a set of feature descriptors (i.e. CSS selectors[1] or XPath expressions[2]) that specify HTML elements which define parts of the page structure where the feature manifests. The goal of this phase is to build the client-side dependency graph, identify all feature manifestation points and gather dynamic information (*Execution Summary*) necessary for the accurate identification and extraction of feature code. The process interprets the whole web application with the scenario as a guideline. During the interpretation, as code expressions are evaluated, the dependency graph is created (Chapter 4), and when a point in the application

[1]http://www.w3.org/TR/CSS2/selector.html
[2]http://www.w3.org/TR/xpath/

execution is reached (i.e. a code expression is evaluated) that represents a *feature manifestation*, that point is stored.

Phase 2 – *Graph marking* – marks all code and resources that directly or indirectly contribute to the demonstrated feature, by traversing the dependency graph for every HTML node in the specified structure and for every feature manifestation. In essence, the graph marking phase performs dynamic program slicing (Section 2.6.2) with the HTML nodes of the specified structure and feature manifestations as slicing criteria. Usually there are multiple such slicing criteria, and in essence, the feature code is actually a union of code slices obtained for each slicing criterion. However, it is a well known fact that unions of dynamic slices do not necessarily reproduce application behavior for each slicing criteria [28]. For this reason, in order to compute correct unions, we gather additional dynamic data in the interpretation phase (*Execution Summary*).

Once the correct union of slices is computed, we generate code and download resources from the marked nodes. This action, in essence, extracts a subset of the original application still able to reproduce the scenario. In other words, the implementation of a feature, for this particular scenario, is identified and extracted.

6.2.1 Example

In the following sections we will illustrate the identification process with a running example shown in Listing 6.2.

```
1  <html>
2   <head>
3     <style>
4     .fav{background-image: url("fS.png");}
5     .noFav{background-image: url("nS.png");}
6     #star { width: 32px; height: 32px;}
7    </style>
8   </head>
9   <body>
10   <div class="imageRaterContainer">
11    <img alt="Image" src="atom.png"/><br/>
12    <div id="star" class="noFav"></div>
13   </div>
14   <div id="notif"></div>
15 <script>
16  var star=document.getElementById("star")
17  var notif=document.getElementById("notif")
18  star.onclick = function () {
19   var dec = star.className == "noFav" ? "fav" : "noFav";
```

```
20    star.className = dec;
21    var req = new XMLHttpRequest();
22    req.open("GET", "d.php?d="+dec, false);
23    req.send();
24    notif.textContent = req.responseText;
25  };
26  </script>
27  </body>
28  </html>
```

Listing 6.2: Example application

This very simple web application has two features (application UI is shown in Figure 6.3): *i)* it allows the user to mark an image as a favorite and sends that decision to the server, and *ii)* displays the message returned from the server (note that they could as well be considered as a single feature, but in this example, for the sake of presentation, we will consider them as separate). Both features are triggered by a scenario in which the user, by clicking on the star, toggles the image as a favorite. On each click, a request is sent to the server with the information about the state of the star.

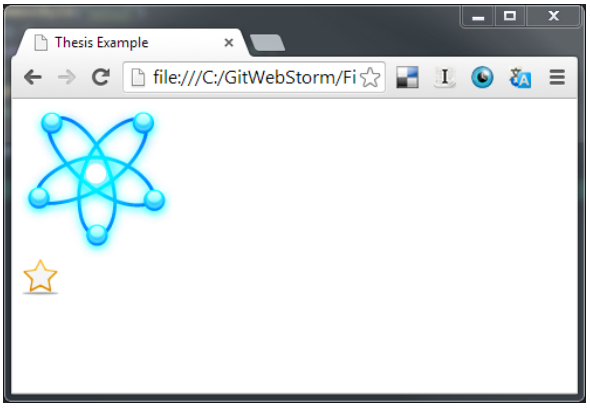


Figure 6.3: The UI of the application from Listing 6.2

The UI of the application is composed of two containers: the first (*imageRaterContainer*, line 10) is used as a container for the image element and the star element, and defines the structure related to the first feature; and the second (*notif*, line 14) is used for displaying status mes-

sages returned from the server, and defines the structure related to the second feature.

From the application's behavior point of view there are three crucial JavaScript expressions in Listing 6.2: lines 20, 23, and 24; lines that directly modify the DOM of the page (lines 20 and 24), or communicate with the server-side (line 23). From the feature point of view: lines 20 and 23 contribute to the behavior of the first feature, and line 24 to the behavior of the second feature. Our approach is, in essence, dealing with the identification of such feature manifestation expressions, determining whether or not they are important from the perspective of the selected feature, and then performing dynamic program slicing with those expressions as slicing criteria.

6.3 Interpretation

The first phase of the identification process is interpretation (Algorithm 7). As input, this phase receives the web application code, the recorded event trace (*featureScenario*), and the selectors specifying parts of the page structure where the feature manifests (*featureDescriptors*). The goal of this phase is to create the dependency graph, identify all feature manifestation points, and gather data necessary for the computation of correct slice unions. For this reason, the algorithm declares three global variables: *dGraph* which stores the dependency graph, *fManfs* for storing feature manifestations, and *exeLog* for logging code expressions that can cause problems when performing slice unions. The graph construction algorithm is already described in Chapter 4, Algorithm 1. The feature manifestation point detection and dynamic information gathering is done together with the graph construction process, while JavaScript code expressions are being evaluated (function *OnExpressionEvaluation*).

When detecting feature manifestation points, the main idea is to identify JavaScript code expressions that modify the target parts of the page structure, or that communicate with the server-side application. For this reason, on each evaluation of a JavaScript code expression, the function *OnExpressionEvaluation* is called with the j-vertex (*jVrtx*) matching the currently evaluated code expression and the evaluation result (*evalRes*). If the currently evaluated code expression is modifying the DOM of the page (line 4) and the modified DOM HTML nodes are parts of the targeted page structure (line 6) then the j-vertex causing the modification

Algorithm 7 Interpretation(*code*, *featureScenario*, *featureDescriptors*)

```
1:  dGraph ← buildGraph(code, eventTrace)
2:  fManfs ← []; exeLog ← []
3:  function OnExpressionEvaluation(jVrtx, evalRes)
4:      if isModifyingDOM(evalRes) then
5:          modifNds ← getModifNodes(evalRes)
6:          if match(modifNds, featureDescriptors) then
7:              push(fManfs, point(jVrtx, getLastDep(jVrtx), 'UI'))
8:          end if
9:      else if isEstablishingServerSideComm(evalRes) then
10:         push(fManfs, point(jVrtx, getLastDep(jVrtx), 'COM'))
11:     end if
12:     if canThrowException(jVrtx) then
13:         push(exeLog, point(jVrtx, getLastDep(jVrtx)))
14:     end if
15: end function
```

and the last dependency created from the j-vertex are stored as a feature manifestation point. For server-side communication, we consider that it is a part of a feature if it is, in any way, dependent on HTML elements that are parts of the targeted page structure. Since in the interpretation phase the dependencies are not yet followed, each server-side communication is treated as a potential feature manifestation point (lines 9–10, Algorithm 7), but with a flag that marks it as such ('COM').

```
15 <script>
16  var star=document.getElementById("star")
17  ...
18  star.onclick = function () {
19  var dec = star.className == "novFav" ? "fav" : "noFav";
20  star.className = dec;
21  var req = new XMLHttpRequest();
22  req.open("GET", "d.php?d="+dec, false);
23  req.send();
24  ...
```

Listing 6.3: Code Excerpt from Listing 6.2

Example. Consider the evaluation of an assignment expression in line 18, Listing 6.3, which assigns a function to a property of an object. Since the *star* identifier refers to an HTML node that is part of the targeted structure, the j-vertex matching the assignment expression and

the last dependency from that j-vertex (to the j-vertex matching the *star.onclick* expression) are stored as a feature manifestation point. A similar process is repeated when evaluating the assignment expression in line 20. The next interesting expression evaluation occurs in line 23, where an HttpRequest is sent. According to Algorithm 7, lines 9–10, the currently evaluated node, along with its last dependency will be stored as a potential feature manifestation point.

The algorithm continues with adding the information to the *exeLog* (line 13, Algorithm 7). For the reasons that will be described in the following section, we log all evaluated expressions that can potentially cause exceptions.

6.4 Problems with slice unions

Our ultimate goal is to use the feature identification method to reuse feature code into another application. This requires that we are able to obtain a subset of the source code that, for a given scenario, behaves in the same way as the whole application. Since a feature manifests through a sequence of feature manifestation points, we have to identify the subset of the application's code that influences each feature manifestation point. A straightforward approach for identifying the code of the entire feature would then be to just perform a union of all code expressions that influence at least one feature manifestation point. However, this approach is unsound.

Consider the example in Listing 6.4, where the goal is to extract the code of a feature that manifests on the parts of the web page structure with identifiers *container1* and *container2*.

```
1  ab:  var c1=document.getElementById("c1");
2
3  abc: function Cont(afImplement) {
4  c:     if(afImplement)
5  c:       var afterImplement = afImplement;
6         else
7           afterImplement = function(){};
8
9  abc:   this.init = function() {
10 ab:      c1.textContent += "1"; /* feature manifestation*/
11 c:       afterImplement();
12        };
13     }
14
```

```
15  a:    var o1 = new Cont();
16  a:    o1.init();
17
18  c:    var c2=document.getElementById("c2");
19  bc:   var o2 = new Cont(function(){
20  c:      c2.textContent+="2" /*feature manifestation*/
21        });
22  bc:   o2.init();
```

Listing 6.4: Example illustrating the problem of merging results for three feature manifestation points. Markings a, b, and c denote the feature manifestation point that has caused the line inclusion.

The example in Listing 6.4 has 3 feature manifestation points: *a*) line 10 invoked by the call expression in line 16, *b)* line 10 invoked by the call expression in line 22, and *c)* line 20. From the perspective of the feature manifestation point a the necessary JavaScript code lines are: 1, 3, 9, 10, 15, 16; from the perspective of feature manifestation point b: 1, 3, 9, 10, 19, 22; and from the perspective of the feature manifestation point c: 3, 4, 5, 9, 11, 18, 19, 20, 22. For each feature manifestation point this is the minimum amount of code necessary to replicate it. If we perform a simple union of the identified code lines, we end up with the whole JavaScript source code except for lines 6 and 7. However, this is not correct: the execution of the *init* function (lines 9–12) caused by the call expression in line 16 (o1.init()) now contains an error that will stop application execution – the *afterImplement* identifier evaluates to *null* instead of a function, due to not including line 7 (line 11 is included in the whole code because of feature manifestation c, and is unnecessary from the perspective of feature manifestation points a and b).

In general, consider two feature manifestation points a_1 and a_2, where it was identified that the necessary control-flow for a_1 is a sequence of expressions $\langle \ldots, e_a, e_c, \ldots \rangle$, and the control flow of a_2 is a sequence of expressions $\langle \ldots, e_a, e_b, e_c, \ldots \rangle$. When we construct a slice union, the control-flow for a_1 now becomes $\langle \ldots, e_a, e_b, e_c, \ldots \rangle$. Since e_b was not included on behalf of a_1, none of its dependencies at this point of execution were traversed and there is no guarantee that e_b will not cause any problems (e.g. throw a null exception because its initialization was not included). For this reason, since our end goal is to enable automatic feature reuse, we will follow the dependencies of e_b leading to the execution of a_1, at the expense of including additional code, because having executable feature code is more important than having the minimal amount of code. To do this, in Algorithm 7, line 13, we have to collect the trace

of all evaluated member expressions and call expressions (*exeLog*), i.e. expressions which when evaluated to null can change the control-flow (e.g by throwing an exception). Notice how this does not have any influence on the result of the evaluation of a_1 since e_b has already been determined as unimportant from the perspective of a_1.

6.5 Graph Marking

So far, in the interpretation phase, we have identified points in the execution where the behavior is manifested, but to identify all relevant code, we have to track all direct and indirect dependencies of those points. This part of the process is handled by the graph marking phase, which is composed of two steps: *i)* marking of feature code and *ii)* handling slice union problems.

6.5.1 Marking Feature Code

The first step of the graph marking phase is the marking of feature code. As is described in Algorithm 8, the dependency graph is traversed for all feature manifestation points (lines 2–12) and for all h-vertices (lines 13–19) that match, or are contained within, parts of the page structure specified by the selectors (*fDescriptors*). There are two different kinds of feature manifestation points (UI modifications and server-side communications), and the process treats them differently. If the feature manifestation point is communicating with the server-side, then the graph is traversed, for this point, only if there is a dependency from the feature manifestation point to the part of the structure where the feature manifests (lines 6, 7). And if the feature manifestation point is modifying parts of the page structure where the feature manifests, then the dependencies of that feature manifestation are always marked (line 10).

The *markGraph* function (Algorithm 9) describes the process of graph traversal with the goal of marking code nodes that influence the selected vertices. The key point is the selection of the dependencies that will be followed (*getPriorDepends* function). In the interpretation phase, all dependencies have been labeled with the identifiers of the evaluation position, and the *getPriorDepends* selects all previous non-traversed dependencies according to the evaluation position.

Algorithm 8 Marking Feature Code

```
1: function markFeatureCode(dGraph, fManfs, fDescriptors)
2:     for all fManf in fManfs do
3:         mVertex ← getVertex(fManf)
4:         d ← getDependency(fManf)
5:         if isServerSideCommunication(fManf) then
6:             if depsOnImprtntNd(mVertex, d, fDescriptors) then
7:                 markGraph(mVertex, d)
8:             end if
9:         else
10:            markGraph(mVertex, d)
11:        end if
12:    end for
13:    for all hVertex in getHVertices(dGraph) do
14:        if matches(getHNode(hVertex), fDescriptors) then
15:            for all d in getDependencies(hVertex) do
16:                markGraph(hVertex, d)
17:            end for
18:        end if
19:    end for
20: end function
```

Algorithm 9 Marking Graph vertices

```
1: function markGraph(vertex, dep)
2:     markAsIncluded(vertex)
3:     for all currDep in getPriorDependencies(vertex, dep) do
4:         markAsTraversed(currDep)
5:         markGraph(getTargetNode(currDep), currDep)
6:     end for
7: end function
```

Complexity. Let $G = \langle V, A \rangle$ be a dependency graph built in the interpretation phase; where V is a set of vertices and A a set of edges; and let s be a sequence of evaluated expressions in a scenario. The execution of the algorithm depends on the two for loops. For the first loop, the length of *fManfs* is upper bound by $|s|$ – there can not be more feature manifestations than evaluated expressions, however even though it is technically possible that all evaluated expressions are feature manifestations, it is often the case that $|fManfs| \ll |s|$. Every execution of the *depsOnImprtntNd* function can at most go through the whole graph (ev-

ery arc can be traversed at most once), and the number of executions is upper bound by $|A|$. The second loop is executed for each HTML node in the parts of the page structure where the feature manifests, so the number of iterations is upper bound by $|V|$. The *markGraph* function, which is called in each loop, can at most (across all invocations) visit all edges of the graph and thus has an upper bound of $|A|$. So the upper bound of the graph marking algorithm is: $O(|s||A|^2 + |V||A|)$.

Example. For the example in Listing 6.2, two feature manifestation points were identified: one for the execution of the assignment expression in line 20 (*assExpr@20: star.className = dec;*) that modifies the part of the UI, and one for the call expression in line 23 (*req.send();*), labeled as an server-side communication feature point. First, the *assExpr@20* is marked as important, and its dependencies traversed: the *memberExpr@20* with a dependency to *varDecl@16* is also marked as important, along with the node matching the initialization call expression (*document.getElementById('star') – callExpr@16*) where the current value of the identifier was set. Since the call expression is dependent on the *div@12*, it is also marked as important. This also causes the marking of h-nodes: *div@10*, *body@9*, *html@1* due to structural dependencies, c-nodes: *.noFav@5*, *#star@6* (which causes the marking of *style@3* and *head@2*). Since *div@12* is also dependent on *assExpr@18* all of its dependencies are also included. Similarly, for the node matching the right hand side of the assignment expression in line 20 (*identifier@20*), all the dependencies are also traversed and marked. Next, the second feature manifestation point is processed. Since it is an server-side communication feature manifestation point, first its dependencies are followed in order to determine if it is in any way dependent on any important part of the UI. Since it is indirectly dependent on *div@12*, the graph is traversed, and all expressions in lines 21–23 are marked as included. The algorithm then goes through all h-nodes that define the selected parts of the web page structure, and traverses their dependencies. In this example, this does not include any more expressions, since everything important was already included in previous traversals.

```
1  <html>
2   <head>
3     <style>
4     .fav{background-image: url("fS.png");}
5     .noFav{background-image: url("nS.png");}
6     #star { width: 32px; height: 32px;}
7    </style>
```

```
8   </head>
9   <body>
10   <div class="imageRaterContainer">
11    <img alt="Image" src="atom.png"/><br/>
12    <div id="star" class="noFav">Note</div>
13   </div>
14
15 <script>
16  var star=document.getElementById("star");
17
18  star.onclick = function () {
19    var dec = star.className == "noFav" ? "fav" : null;
20    star.className = dec;
21    var req = new XMLHttpRequest();
22    req.open("GET", "d.php?d="+dec, false);
23    req.send();
24
25 };
26 </script>
27 </body>
28 </html>
```

Listing 6.5: Example application extracted code

With this, the process has identified all code expressions responsible for the implementation of a feature during the given scenario. In essence, the process has identified that the first behavior (selecting the image as favorite) depends on the execution of the assignment expression in line 20, and the second behavior (sending the decision to the server) depends on the call expression in line 23. By traversing the dependencies of those two expressions the process identifies code responsible for the whole feature. By generating code from the nodes marked as important we get the code shown in Listing 6.5.

Compared to the code from Listing 6.2 the identification process has identified the HTML element defined in line 14, the variable declaration in line 17, and the assignment expression in line 24 as code that is not necessary for the target feature. Notice how the second expression in the conditional expression in line 19 (“noFav”) is replaced by null, simply because it was not executed in the demonstrated scenario. In order to remedy this, the developer would have to change the scenario by demonstrating another click on the star element (causing the execution of the second expression).

6.5.2 Fixing Slice Union problems

After the code of each feature manifestation point has been identified, in order to extract the feature code in a stand-alone fashion, we have to create a union of all code expressions that are used by at least one feature manifestation point. As is explained in Section 6.4, this can lead to a number of problems. In order to fix these problems we present Algorithm 10.

Algorithm 10 Fixing Slice Union Problems

```
1:  function FIXSLICEUNIONPROBLEMS(exeLog)
2:      do
3:          hasIncludedNewVertices ← false
4:          for all item in exeLog do
5:              jVrtx ← getJVertex(item)
6:              if isIncluded(jVrtx) and anyCntxtDepTrvrsd(item) then
7:                  deps ← getUntraversedDepsFromContext(jVrtx, item)
8:                  numNewInclVrtxs ← markAllDeps(jVrtx, deps)
9:                  if numNewInclVrtxs ≠ 0 then
10:                     hasIncludedNewVertices ← true
11:                 end if
12:             end if
13:         end for
14:     while hasFoundNewVertices
15: end function
```

The main algorithm loop is executed as long as new vertices that will be included into the final feature code are found. In each iteration (lines 4–13, Algorithm 10), the algorithm goes through all trace items (*item*) in the execution log (*exeLog*, created in Algorithm 7). If the trace item's j-vertex is included as part of feature code in previous graph markings, then the algorithm checks if at least one dependency from the same context in which the trace item was executed has been traversed (line 6). If it has been, then in the final feature code, the matching code expression will be executed in this execution context. For this reason, we have to follow the untraversed dependencies (line 7) of the j-vertex that exist in the current context (line 8). The function *markAllDeps* calls the *markGraph* function on the *jVrtx* for each such dependency, with an addition that it also counts the number of newly included vertices.

Example. Consider the code in Listing 6.6. As was mentioned in Section 6.4, the union code does not include line 7 where the variable *afterImplement* is initialized. When executing the code leading to feature manifestation *a*, this causes a null exception for the call expression in line 11 (included due to the feature manifestation point *c*). Algorithm 10 fixes this problem by going through the *exeLog*. When visiting the call expression in line 11 called by the call expression in line 16, the algorithm computes that the call expression in line 11 is included and that there are dependencies created in this context that were traversed. This means that the dependencies of the call expression in line 11, for this context, also have to be traversed. By following these untraversed dependencies, the assignment expression in line 7 is included, and the slice union problem is fixed.

```
/*03*/abc: function Cont(afImplement) {
/*04*/c:      if(afImplement)
/*05*/c:        var afterImplement = afImplement;
/*06*/        else
/*07*/          afterImplement = function(){};
/*08*/
/*09*/abc:    this.init = function() {
/*10*/ab:       c1.textContent += "1";
/*11*/c:        afterImplement();
/*12*/        };
/*13*/      }
/*14*/
/*15*/a:    var o1 = new Cont();
/*16*/a:    o1.init();
```

Listing 6.6: Excerpt from Listing 6.4

6.6 Evaluation

We have performed the evaluation of the feature identification process with two goals: *i)* to show that the process is able to identify code that implements a feature manifested by a scenario, and *ii)* to compare the gains that can be achieved by using our method with a baseline obtained by profiling code. We consider the identification process successful if by extracting the identified code into a stand-alone web application we get the same functional and visual results when executing the scenario with the extracted code as we do with the original code. We compare the results obtained with our method with the results obtained by profiling

code. Profiling is a straightforward extraction approach – the idea is to keep lines executed in a scenario, while maintaining syntactical correctness. The code extracted in this way is still capable of replicating the scenario.

We have performed three sets of experiments: *i)* extracting client-side library features, *ii)* extracting client-side web application features, and *iii)* page optimization. In all cases, scenarios are specified as tests. We consider the feature identification process successful if the tests can be successfully executed both in the original application, and in the new application composed out of the extracted code.

All applications, and the tests describing the scenarios can be downloaded from `www.fesb.hr/~jomaras/download/FIdEvaluation.zip`.

6.6.1 Extracting Library Features

In the first experiment – *extracting library features* – our goal was to validate the extraction process against a set of externally defined behaviors. For this purpose, we decided to use several widely used JavaScript libraries, which all come with unit-tests specified by their developers. While it is true that features do not need to correspond to unit-tests, and that the purpose of the tests is to reveal errors and not necessarily specify features, in this experiment we consider unit-tests as externally defined behavior specification that we use to gain information about whether the extraction process is correct. The extraction process is successful if the extracted code is able to pass the same unit tests as the original code.

The experiments were performed on six open-source JavaScript libraries: Gauss[3] – a library for statistics, analytics, and sets; Sylvester[4] – a vector and matrix library; Prototype[5], Underscore[6], MooTools[7], and jQuery[8] – widely used general purpose libraries.

The experiment was successful, and in all 469 cases the extracted code was able to pass the tests. Table 6.1 shows the summary of the experiment results. For each library, we present the range (from minimum to maximum), the average, and the median of the following metrics: lines of

[3] https://github.com/stackd/gauss
[4] https://github.com/jcoglan/sylvester
[5] http://prototypejs.org/
[6] http://underscorejs.org/
[7] http://mootools.net/
[8] http://jquery.com/

Table 6.1: Library extraction experiment results summary. LOC (Lines of Code), Exe (Number of evaluated expressions in thousands), Time in sec, #FM (Number of feature manifestations), o (original), p (profiled), and s (sliced). The number before each library name presents the number of tests.

	#33 Gauss: 678 LOC			#84 Sylvester: 1756 LOC		
	Range	**Avg**	**Med**	**Range**	**Avg**	**Med**
oLOC	690-699	692.2	692	1767-1953	1794.2	1788.5
pLOC	79-167	114.9	109	337-702	430.1	405
sLOC	29-86	56.0	56	33-450	141.87	116
pExe	0.25-231	26.0	12.5	4.2-400	16.4	6.3
sExe	0.15-230	18.2	3	0.1-369.5	11.2	2.5
Time	0.34-43.7	4.6	1.8	1.7-553	104.0	2.2
#FM	1-9	2.1	1	1-30	4.6	3
	#79 underscore: 1258 LOC			#85 prototype: 7036 LOC		
oLOC	1268-1493	1285.1	1276.7	7049-7117	7062	7057
pLOC	232-553	274.9	273	1631-1827	1681.1	1664
sLOC	163-262	199.86	198	20-498	117.27	89
pExe	7-19.5	8.1	7.5	48-74.5	50.3	50
sExe	5.4-18.6	6.5	5.8	0.02-26.4	2.0	0.5
Time	1.9-5.3	2.1	2	11.6-53.2	18.3	16.9
#FM	1-23	5.0	4	1-47	8.0	6
	#35 mooTools: 5977 LOC			#153 jQuery: 9790 LOC		
oLOC	5988-6118	6003.4	6004	10,503-10,266	10,079	10,071
pLOC	1564-1724	1610.0	1617	1807-2978	2203.4	2145.5
sLOC	10-550	257.4	210	18-1834	943	873.5
pExe	150.2-168.5	152.6	152.8	33-225.7	52.8	41.6
sExe	0.22-90	35.3	16.9	0.01-150.5	27.1	20.8
Time	23.3-84.8	55.5	48.4	11.6-86.5	18.8	15
#FM	1-34	3.4	2	1-604	32.2	17
	#469 Total					
oLOC	690-10,266	6003.4	6004			
pLOC	79-2978	1272.5	1645			
sLOC	10-1628	326.2	196.5			
pExe	0.2-396.6	43.8	37			
sExe	0.01-369.5	16.3	6			
Time	0.3-553	16.1	13.1			
#FM	1-604	14.0	5			

code (LOC), number of evaluated expressions in thousands (Exe), slicing time in seconds (Time), and the number of encountered slicing criteria (SC). For LOC we present the original number of code lines ($oLOC$), profiled lines of code ($pLOC$), and sliced lines of code ($sLOC$). When presenting the number of evaluated expressions, we show how many expressions where evaluated when executing the test in the context of the original code ($pExe$) and in the context of the sliced code ($sExe$).

As shown in Table 6.1, the average LOC of a single test with the library included varies from 690 for the smallest, Gauss library, to 10,266 for the biggest, jQuery library. During the execution of each test, the interpreter on average visits around 22 percent of library code (Figure 6.4).

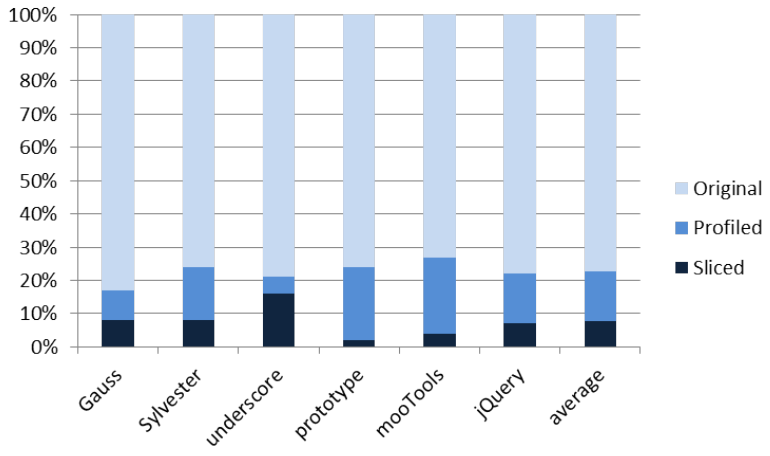


Figure 6.4: Average difference in original, profiled, and sliced lines of code, for the six test libraries

Out of the profiled code, the feature identification process identifies as important around 36 percent of the code. So, when compared to profiling, our method achieves around 63 percent of savings in terms of code lines (Figure 6.5).

When executing the tests on the sliced code, the experiments have shown that the behavior that passes the test can be reproduced with, on average, 60 percent of evaluated expressions. This saving represents performance gains (Figure 6.6).

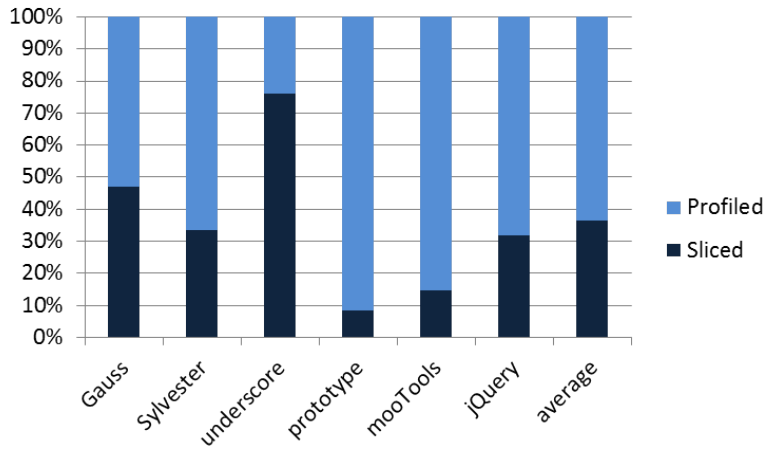


Figure 6.5: Average ratio between profiled and sliced lines of code

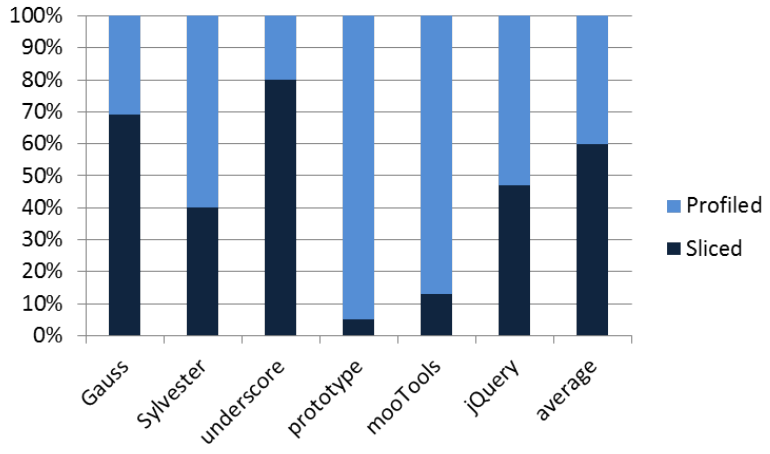


Figure 6.6: Average ratio between the number of evaluated code expressions in the profiled code and sliced code

Slice Unions

As can be seen from Table 6.1, the average number of feature manifestations (slicing criteria) per test is 14. This means that problems with slice unions can occur. In our experiments, the slice union problems occur in one test of the Sylvester library, and in 17 tests of the jQuery library. The comparison of the average code sizes for these tests is shown in Figure 6.7. On average, in order to resolve the problems of slice unions, the process has included additional 17.5 percent code.

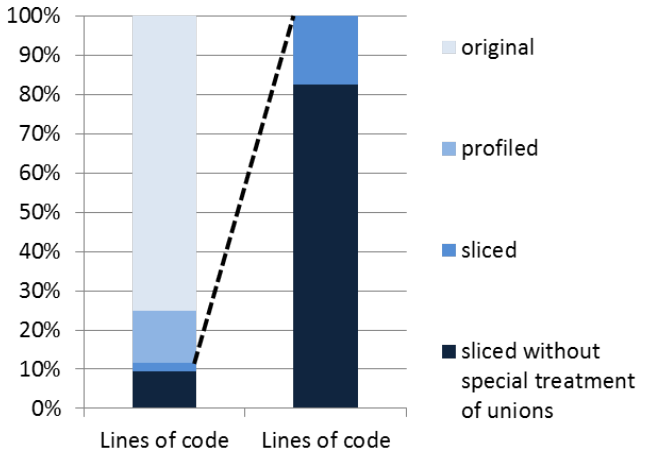


Figure 6.7: Average difference between the sliced without slice unions, sliced, profiled, and original code, for the tests failing without taking into account slice unions

6.6.2 Extracting Features

For the second experiment – *extracting features* – our goal was to show that the process is capable of extracting features from standard web applications, features implemented with HTML, CSS, and JavaScript code. The experiment was performed on ten medium-sized web applications (Table 6.2), which were selected because they have multiple, easily-identifiable features. Each test application has at least two features whose code was identified and extracted by the feature identifica-

tion process. For each test application, we have manually identified its features, the parts of the structure where the features manifests, and have specified the scenarios that capture the feature behavior.

Table 6.2: Web applications used for feature extraction. LOC (Lines of Code)

PageID	Page name	LOC	#Features
1	ds22	21309	4
2	salleedesign	12394	3
3	mtcdc	13957	3
4	mailboxing	12171	3
5	dunked	12727	3
6	wordpress-virtue	24569	3
7	hipstamatic	12073	2
8	makalu	11998	2
9	nitrografix	12755	2
10	kennymeyers	11931	2

Every scenario that causes a manifestation of a particular feature is represented as a Selenium[9] test. Selenium is a navigation scripting and testing utility that can be used to automate web applications for testing purposes. A developer specifies a series of actions and defines UI properties that have to be satisfied in order for the test to be successful. We consider that a feature is successfully extracted if the same tests can be successfully executed both in the original application and in the application composed out of the extracted code.

For the ten web applications, we have set up 27 experiments, whose results are shown in Table 6.3. The experiment was successful and in all cases the extracted code was able to successfully pass the tests. All applications, their test cases, and experiment results can be downloaded from `www.fesb.hr/~jomaras/download/FIdEvaluation.zip`.

On average, during the execution of each scenario the control-flow went through 30 percent of the application code, and out of that visited code, the feature identification process calculated that around 56 percent

[9]http://docs.seleniumhq.org/

Table 6.3: Experimental results for feature extraction. ID (Page and Feature), p (Profiled), s (Sliced), o (Original), LOC (Lines of Code), Time (slicing time in seconds)

ID	pLOC	sLOC	pLoc/oLoc	sLOC/pLOC	Time
1-1	5408	3552	25%	66%	100
1-2	5471	3606	26%	66%	136
1-3	5552	3629	26%	65%	355
1-4	5412	3495	25%	65%	101
2-1	3159	1716	25%	54%	36
2-2	2921	1144	24%	39%	20
2-3	3156	1826	25%	58%	26
3-1	3477	1658	25%	48%	13
3-2	4544	2710	33%	60%	83
3-3	4770	3070	34%	64%	96
4-1	3909	2539	32%	65%	27
4-2	3815	2530	31%	66%	28
4-3	4090	2688	34%	66%	53
5-1	4417	2826	35%	54%	145
5-2	4501	2293	35%	51%	99
5-3	4406	2303	35%	52%	125
6-1	9072	4947	37%	55%	148
6-2	9080	4744	37%	52%	129
6-3	9640	5042	39%	52%	141
7-1	2645	1264	22%	48%	28
7-2	2662	1283	22%	48%	30
8-1	3253	1746	27%	54%	35
8-2	3251	1697	27%	52%	32
9-1	4549	3160	36%	69%	84
9-2	4486	3149	35%	70%	216
10-1	2388	62	20%	3%	25
10-2	4365	3066	37%	70%	57
Average	4607.3	2657.2	30%	56%	87.7
Median	4406	2688	31%	58%	83

of executed code lines is necessary for the implementation of the target feature.

6.6.3 Page optimization

For the third experiment – *page optimization* – the goal was to show that the process is capable of identifying code of all features offered by the application. In the experiments the process identifies and removes code that does not contribute to any behavior. In order to do this, we have to know all application behaviors. For this reason, we have chosen 8 demo web applications that describe their behavior, and 2 standard web applications where it was easy to identify all application behaviors. Similar to the feature extraction experiment, based on the application behaviors, we have defined Selenium tests, and we consider that the extraction is successful if the extracted code is able to pass the predefined tests the original application has passed.

Table 6.4: Experimental results for page optimization. o (Original), p (Profiled), s (Sliced), LOC (Lines of Code), Exe (Executions), LT (Loading Time)

Page	oLOC	pLOC	sLOC	pExe	sExe	LT	sLT
codaBubble	9611	2838	1383	39598	19488	204	79
fncyChckbx	117	117	108	1491	1235	36	36
hmnTypst	9480	2017	605	31629	12016	204	34
idtPride	10621	3981	2537	114283	58479	285	165
password	149	149	149	7047	7047	10	10
tinyslider	260	254	248	20704	20690	27	27
tabs	9514	2520	1234	36599	19525	200	65
fourandthree	10564	3664	2393	30523	15513	247	200
suckerFish	9663	2623	1352	75952	36152	205	75
jSlideshow	9859	3273	1944	101472	66917	232	116
average	6984	2144	1195	45930	25706	165	80

Tables 6.4 and 6.5 show the selected pages and all data gathered during the experiment. The experiment was successfull, i.e. for all test applications, the extracted code was able to pass the Selenium tests. The table shows that the optimized page generates 38%–100% executions (savings from 0%–62%), resulting in 0%–83% gains (1 - s/p LT) in page

loading time. The savings are greatest in applications that use client-side libraries, while they are almost non-existent in small demo applications (where there is no dead code). All applications, their test cases, and experiment results can be downloaded from: `www.fesb.hr/~jomaras/download/FIdEvaluation.zip`.

Table 6.5: Comparison between original (o) code, profiled (p) code, and sliced (s) code. LOC (Lines of Code), Exe (Executions), LT (Loading Time), Time (Feature location time in seconds)

Page	p/o LOC	s/p LOC	s/p Exe	s/p LT	Time
codaBubble	30%	49%	49%	39%	27
fncyChckbx	100%	92%	83%	100%	3
humanTypist	21%	30%	38%	17%	23
idtPride	37%	64%	51%	58%	74
password	100%	100%	100%	100%	3
tinyslider	98%	98%	100%	100%	135
tabs	26%	49%	53%	32%	25
fourandthree	35%	65%	51%	81%	27
suckerFish	27%	52%	48%	37%	82
jSlideshow	33%	59%	66%	50%	52
average	50.7%	65.8%	63.9%	61.4%	45

It is important to note that the goal of the evaluation was to show that the method is capable of identifying code responsible for a behavior, and not to determine how much unnecessary code is usually included in web applications. However, the results indicate that web applications contain more code than is actually needed for their behavior, and that considerable savings could be achieved by applying this extraction method.

6.6.4 Threats to validity

There are several issues that might occur when attempting to generalize the experiment results. One concern is whether the selected applications are representative of real-world web applications. We tried to tackle this concern by performing experiments on a wide range of applications: from JavaScript libraries ranging from 690 to 10,000 lines of code, all the way to full web pages built from around 25,000 lines of code, that use different

wide-spread JavaScript libraries.

Another important threat to validity is whether or not our method is capable of extracting all of the code that implements a feature. Since our method is based on dynamic analysis of web application code in a particular scenario, we are aware that the quality of the scenarios is vital for the correct identification of feature code. This is why we are not claiming that our method is capable of identifying the full code of a feature, but the code of the feature manifested by the specified scenarios.

6.7 Conclusion

In this chapter, we have presented a method for identifying the implementation details of features that manifest on particular parts of the web page structure, for a certain scenario. During application execution, the process constructs a dependency graph and identifies the points in the execution where the feature manifests (*feature manifestations*). The implementation details of a feature are then identified by traversing the dependency graph for all feature manifestations. We have defined algorithms for finding feature manifestations and marking feature code.

The main advantages of the approach are: *i)* it does not require any formal specification of the feature (something that is rarely done in web application development) and the user can specify the desired feature behavior; and *ii)* it enables dynamic tracking of code dependencies (something that can not be accurately done statically for a language as dynamic as JavaScript). The limitations of the approach are: *i)* that it is primarily suited for functional features with observable behaviors (non-functional features, such as security or maintainability do not have determinable feature manifestations), and *ii)* the accuracy and the completeness of the captured feature is dependent on the quality of the scenarios.

We have evaluated the approach by performing three sets of experiments on a range of web applications, and have reached two conclusions: *i)* the method can correctly identify stand-alone behaviors by analyzing web application event traces, and *ii)* considerable savings in terms of number of executions, page loading time, and code size can be achieved while still being able to reproduce the demonstrated behavior.

Chapter 7

Integrating Features

Once the code of the target feature has been identified, in order to achieve reuse, we often have to integrate the feature code into the code of an already existing application. Merging two code bases can lead to a number of problems that have to be detected and fixed. In this chapter, we present an automatic feature integration process. We identify problems that can occur when integrating code from one application into another application; and present a set of algorithms that detect and resolve those problems, perform the actual code merging and verify that the integration is performed successfully.

7.1 Overview

In this section, we define the goal and successfulness criteria for feature integration, and we present an overview of the whole process.

7.1.1 Goal

Let A and B be two client-side web applications, each defined with its HTML code, CSS code, JavaScript code, and resources – $\langle H_A, C_A, J_A, R_A \rangle$ for application A and $\langle H_B, C_B, J_B, R_B \rangle$ for application B. Let f_a be a feature from A that manifests when a scenario s_a is performed, and that is implemented by a subset of A's code and resources $\langle h_a, c_a, j_a, r_a \rangle$, identified by the *Feature Identification* process (Chapter 6).

The goal of the integration process is to create a new application B' that offers both the feature f_a from A and the features F_B from B. We do this by integrating the code and resources of f_a into application B.

We consider that the integration process is successful if, in the final B' application, the scenario s_a causing the manifestation of f_a can be repeated with the same presentational and behavioral characteristics as in A, and all scenarios S_B of B with the same presentational and behavioral characteristics as in B. This implies that there should not be any feature "spilling" – the feature f_a, in the context of B', should not operate on parts of application originating from B (nor should features from B operate on parts of the application originating from A). With regard to the behavior, this means that JavaScript code j_a, when included in $J_{B'}$, should not interact with H_B, C_B, or R_B, nor should J_B interact with h_a, c_a, or r_a. For the preservation of presentation, CSS rules c_a should not be applied to H_B (nor C_B to h_a).

7.1.2 Process Overview

Once the *Feature Identification* process (Chapter 6) has been performed, i.e. an execution summary has been gathered, the dependency graph constructed, and the graph vertices responsible for the feature implementation identified, we have to integrate the identified feature code with the code of the target application. Since introducing code of one application into another application can lead to a number of different problems, these problems have to be detected and fixed. For these reasons, the process of feature integration is composed of two mandatory phases (Figure 7.1): *i)* Conflict Resolution and *ii)* Merging; and one optional phase: *Verification*. In the context of the whole reuse process shown in Figure 3.2, Figure 7.1 presents the details of the *Feature Integration* activity.

As input, the process receives the *Feature Execution Summary* (E_A) and the *Feature Dependency Graph* (G_A); the *Application Execution Summary* (E_B) and the *Application Dependency Graph* (G_B); the *Reuse Position* that specifies where the feature will be reused; the scenario that causes the manifestation of the feature (s_a) and the scenarios that capture the behavior of the application (S_B). The feature execution summary and graph are obtained with the *Feature Identification* process (Chapter 6), and the application execution summary and the application dependency graph are obtained in the *Application Analysis* phase (Chapter 3).

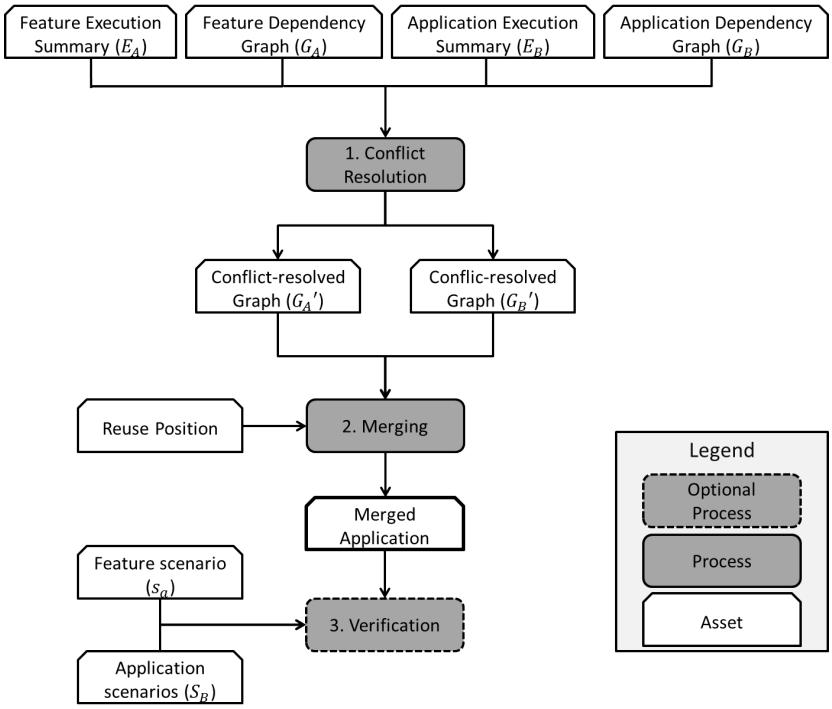


Figure 7.1: The process of extracting and reusing features

Conflict Resolution

The purpose of the integration process is to merge the identified feature code into application B. Since merging two code bases can lead to a number of errors, potential problems have to be detected and resolved. Moreover, these problems can occur both statically and dynamically, and the conflict detection has to take this into account. For this reason, the conflict resolution phase (*1. Conflict Resolution*) is based on the analysis of the execution summaries of both applications. Once the conflicting positions are identified, by using the dependency graphs of both applications, the conflicts are resolved, and the process moves to the next phase – merging.

Merging

In order to achieve reuse, the identified and conflict-resolved feature code $\langle h_{a'}, c_{a'}, j_{a'}, r_{a'} \rangle$ is merged with the conflict-resolved code of application B (*2. Merging*), and a new application B' – $\langle H_{B'}, C_{B'}, J_{B'}, R_{B'} \rangle$ is created. The merging is performed by appending the head HTML nodes from f_a to the head HTML node from B, and the body nodes from f_a to the body node of B. Once this is done, the HTML nodes that define the structure of the feature $h_{a'}$ are moved to the target reuse position. This move can also cause certain kinds of problems, which are detected and fixed.

Verfication

In the final, optional, step of the process (*3. Verification*), the goal is to check that the behavior of the resulting application is the same as the behavior in the originating applications. When executing a certain scenario in the context of the final B' application, code constructs causing the observable behaviors should only influence the nodes originating from the same application (e.g. j_a should only modify h_a).

7.1.3 Running Example

Consider the two applications: A, which has a feature f_a of toggling between image sources on image clicks (Listing 7.1 shows the feature code, identified by the feature identification process), and B (Listing 7.2) with a feature of changing image sources on mouse over and a feature of displaying how many times the images were changed (the source code of application A is colored in red and of application B in blue, to make the origin of code more clear, when the result is presented later).

```
1  <html>
2   <head>
3    <style>
4     img{border: solid; }
5     .mI{ width: 200px; }
6    </style>
7    <script>
8     SRCS = ["img/C.jpg", "img/D.jpg"];
9     Array.prototype.next=function(c){
10     var i = this.indexOf(c)
11     var next = 0
12     if(i>=0 && i<this.length-1)
```

```
13       next = i+1
14      return this[next]
15     }
16     iProto=HTMLImageElement.prototype
17     iProto.toggleSrc = function(s){
18      this.src=s.next(this.src)
19     }
20     window.onload = function(){
21      var im = document.querySelectorAll('img')
22      for(var i=0; i<im.length; i++)
23      im[i].onclick = function(){
24       this.toggleSrc(SRCS)
25      }
26     }
27    </script>
28   </head>
29   <body>
30    <div id="imCont">
31     <img class="mI" src="img/C.jpg"/>
32     <img class="mI" src="img/D.jpg"/>
33    </div>
34   </body>
35  </html>
```

Listing 7.1: Example feature f_a

```
1   <html>
2    <head>
3     <style>
4      img{border:dashed;}
5      .mI{width:300px;}
6     </style>
7     <script>
8      history=[], doc=document;
9      SRCS = ["img/P.jpg", "img/T.jpg"]
10     proto = HTMLImageElement.prototype
11     proto.toggleSrc=function(srcs, c){
12      this.src=this.src.indexOf(srcs[0])==-1?srcs[0]:srcs[1]
13      history.push(this.src)
14      var summ = {}
15      for(var i in history){
16       var item = history[i]
17       if(!summ[item]) summ[item] = 0
18        summ[item]++
19      }
20      c.textContent=JSON.stringify(summ)
21     };
22     window.onload = function(){
23     var im=doc.querySelectorAll("img")
```

```
24      var inf=doc.querySelector("#info")
25      for(var i=0; i<im.length; i++)
26       im[i].onmouseover = function(){
27        this.toggleSrc(SRCS, inf)
28       }
29      }
30     </script>
31    </head>
32    <body>
33     <div id="imCont">
34      <img class="mI" src="img/P.jpg"/>
35      <img class="mI" src="img/T.jpg"/>
36      <div id="info"></div>
37     </div>
38    </body>
39   </html>
```

Listing 7.2: Example Application B

If a naive merge would be performed by simply merging the head and body child nodes from A to B, several problems would occur:

1. the *img* and *.mI* CSS selectors from A (@4, 5, A) would override the *img* and *.mI* CSS selectors from B (@4, 5, B);
2. the *SRCS* array @8, A would override the *SRCS* array @9, B;
3. the extension of the Array prototype in A (@9, A) would cause problems in the for-in loop in B (@15, B), because the for-in loop would also iterate over the *next* property from the Array prototype;
4. the extension of the *HTMLImageElement* prototype (@17, A) would override the *HTMLImageElement* prototype extension @11, B;
5. the function set as an *onload* handler in A (@20, A) would override the *onload* assignment in B (@22, B);
6. both applications query the DOM of the page for all image elements (@21, A and @23, B), and since the structure of the page has changed (from the perspective of each application, two new images have been added) both queries would return more elements than in the original applications (images from B would respond to clicks, and images from A would respond to mouse hovering);
7. two nodes with the same *imCont* identifier would exist (@30, A and @33, B).

These two applications, even though they use some frowned-upon language features (e.g. extending the prototypes of built-in objects), were chosen to illustrate problems that can occur when integrating code of two different code bases. A non-naive, automatic feature integration approach has to take into account all conflicts that can occur when merging the code and resources of two different web applications. In the following section we specify different types of possible conflicts.

7.2 Conflict Types

Merging the code and resources of f_a into B creates a new page whose DOM is different from the DOM expected by the code of each application. This can create a number of problems that are complicated by the fact that the web application code is heavily interdependent and that any change can influence a number of different places. Moreover, because JavaScript is highly dynamic, both the positions on which the problems arise, and the positions to where they are propagated can not be accurately determined statically.

Table 7.1: Types of conflicts in client-side Web applications

Type	**Error source**	**Potential errors introduced**
DOM	HTML node naming attribute	CSS rules applied to different HTML nodes (visual layout changed)
		Different values of JavaScript expressions accessing node visual properties
		Different values of DOM query expressions
JavaScript	Global variables	Naming conflicts
	Built-in object extensions	Naming conflicts in extended objects; Errors when iterating over object properties
	Event-handling properties	Event-handler overriding
	HTTP Requests	Cross-site HTTP request errors
Resource	File names	Resource overriding

Overall, there are three broad types of conflicts: DOM conflicts, JavaScript conflicts, and resource conflicts (Table 7.1). Throughout this section, we will in more detail discuss each of the conflict types, the sources of the errors and their effects on the behavior and presentation of the web application.

7.2.1 DOM conflicts

From the DOM perspective, the merging of HTML code can lead to conflicts in naming attributes of HTML nodes (class, id, and name). Since HTML is an error tolerant language, this won't lead to any problems in the DOM itself. However, node naming attributes are referenced in CSS and JavaScript code, and the main problem with DOM conflicts is that they propagate to CSS and JavaScript code.

Conflicts that propagate to CSS – CSS rules are applied to HTML nodes based on CSS selectors, and if CSS conflicts occur, CSS rules designed to target HTML nodes of one application could, in the final application, be applied to HTML nodes of the other application. This can lead to changes in the visual layout of the page.

Conflicts that propagate to JavaScript – JavaScript code interacts with the DOM and accesses HTML nodes by using queries similar to CSS selectors. This means that if there are DOM conflicts, JavaScript expressions that query the DOM of the page can return different results in the context of the final application than in the original application. Also, if DOM conflicts propagate to CSS code, and CSS rules from one application are applied to HTML nodes from the other application, the results of evaluating certain code expressions that access the element's visual properties could be changed. These conflicts change the internal state of the application, which can result with changes in the intended behavior of the application.

DOM conflicts can occur both statically, by being directly present in the HTML code of the applications, and dynamically with evaluation of JavaScript expressions (e.g. the evaluation of an expression changes an attribute value to a conflicting one).

7.2.2 JavaScript conflicts

Alongside conflicts that propagate from the DOM, JavaScript code can introduce a number of errors, caused by the use of different types of

global variables, or messages exchanged with the server.

In JavaScript there are different types of global variables, and from the perspective of conflict-handling they can be divided into three groups: *i) Standard global variables*, defined by declaring variables in the global scope, or by extending the global window object, which can cause naming conflicts; *ii) Built-in object extensions*, defined by extending built-in objects (e.g. Object, String, Array prototypes, the Math object) which can lead to naming conflicts within the extended objects, and errors can be introduced when iterating over object properties; and *iii) Event-handling variables*, created by registering event handlers (e.g. onload, onmousemove properties of the global window object), which can lead to problems with property overriding.

During the life-cycle of a web application, the client-side can exchange a number of messages with the server-side. Due to security reasons, a client-side application is not allowed to exchange messages with a server-side application hosted on another domain. If the code responsible for the communication is transfered to another server, all attempts to communicate with the server-side will fail, and errors will be introduced into the application.

7.2.3 Resource conflicts

Conflicts can also occur between resources (images, fonts, videos, files, etc.) if there exist resources with the same identifiers in both applications. These types of conflicts can propagate to HTML, CSS, and JavaScript code, and can lead to problems in the visual layout and presentation of the application.

7.3 Resolving Conflicts

The process of resolving conflicts is composed of two steps: *i)* resolving conflicts that arise due to changes in the DOM structures of both applications, and *ii)* resolving problems that happen when combining two JavaScript code bases. Since conflicts can occur both statically and dynamically, all possible conflicts can not be accurately detected with static analysis, and conflict resolutions performed with simple string renamings, without taking into consideration the semantics of the changed expressions, can only resolve a subset of possible problems (and even

then, we can not be sure if they are applied to correct expressions). For this reason, the conflict detection process is based on the analysis of execution summaries and client-side dependency graphs. Once the conflicting positions are located, by following the dependencies in the dependency graphs, it is possible to make accurate modifications of the correct code constructs.

7.3.1 Resolving DOM conflicts

When resolving DOM conflicts, the first step is to identify all static and dynamic code positions that can cause conflicts when a new page is created by merging the DOMs of two applications. Once these positions are identified, conflicted HTML attributes are renamed, new HTML attributes are added, and CSS rules and JavaScript DOM queries are modified in order to localize them in a way that they only interact with nodes from their respective applications. Algorithm 11 describes the process of handling both the DOM conflicts and their propagation to CSS and JavaScript. As input the algorithm receives the dependency graph G_A and execution summary E_A of A for scenario s_a, and the dependency graph G_B and the execution summary E_B of B for scenarios S_B.

Algorithm 11 resolveDOM(G_A, E_A, G_B, E_B)

```
1:  attrConflicts ← getHtmlAttrsConflicts(E_A, E_B)
2:  resourceConflicts ← getResourceConflicts(G_A)
3:  for all item : concat(attrConflicts, resourceConflicts) do
4:      new ← genName(item, E_A, E_B)
5:      for all pos : getUsagePos(item, E_A) do
6:          if isInHtml(pos) or isInCss(pos) then
7:              replaceVal(pos, item, new)
8:          else if isInJs(pos) then
9:              replaceDomStrLit(pos, item, new, G_A)
10:         end if
11:     end for
12: end for
13: expandNodes(genName('f', E_A, E_B), G_A, E_A)
14: expandNodes(genName('b', E_A, E_B), G_B, E_B)
```

The algorithm finds all conflicting HTML node name attributes and all conflicting resources (lines 1, 2), and for each found item a new,

non-conflicting name is generated. An item can be used in a number of different positions: in HTML code as node attributes, in CSS code as selectors or key values, and in JavaScript code as assignment or call expressions (e.g. assignment expressions that modify node attributes, or DOM querying call expressions). If the usage position is in HTML or CSS code then the old value in the feature code is simply replaced with the new, non-conflicting value. If the usage position is in JavaScript code, the process traverses the dependency graph and attempts to find the string literal that matches the old value and replace it with the new value (if the string literal can not be found, e.g. is constructed by concatenating strings, a comment notifying that a conflict was not resolved is added to the access position).

Example. There are two conflicts caused by attribute namings (Listing 7.3): the id *imCont* (@30, *A* and @33, *B*); and class attributes *mI* (@31, 32 *A* and @34, 35 *B*), which are resolved by renaming access positions in application *A*: *i)* attribute *imCont* is used only in HTML code, so *imCont* (@30, *A*) is replaced with *r_imCont*; *ii) mI* is used both in HTML code (@31, 32 *A*) and CSS code (@5, *A*), and is in both places replaced with *r_mI*.

```
.................   A   .................
05 A:.mI{ width: 200px; }
30 A:<div id="imCont">
31 A:  <img class="mI" src="img/C.jpg"/>
32 A:  <img class="mI" src="img/D.jpg"/>
33 A:</div>
.................   B   .................
33 B:<div id="imCont">
34 B:  <img class="mI" src="img/C.jpg"/>
35 B:  <img class="mI" src="img/D.jpg"/>
36 B:</div>
.................   B'   .................
B':.r_mI{ width: 200px; }
B':<div id="r_imCont" o=r>
B':  <img class="r_mI" src="img/C.jpg"/>
B':  <img class="r_mI" src="img/D.jpg"/>
B':</div>
```

Listing 7.3: Excerpts from applications *A* and *B* and the final result when resolving attribute conflicts

The process continues in line 13, Algorithm 11, by handling selector conflicts in CSS and JavaScript code. The goal is to make the selectors more specific by limiting them only to parts of the DOM that match the

originating application. Non-conflicting names are generated with calls to the *genName* function and are added as attributes to enable differentiation between nodes originating from different applications. Type selectors are expanded so they target only nodes they have targeted in the originating applications (both for CSS selectors, and JavaScript DOM queries).

Example. There are two selector caused conflicting expressions (Listing 7.4): CSS selector *img* (@4, *A* and *B*), and *doc.querySelectorAll('img')* (@21, *A* and @23, *B*). Failing to resolve the CSS selectors would cause the wrong styles to be applied to images from application *B*, and failing to resolve the *querySelectorAll* call would result in more elements returned by the query in the context of the final application than in the context of each application. For this reason, the body descendants in application *A* (@29, *A*) are expanded with an additional attribute (e.g. attribute *origin* with value *reuse*, represented by $o = r$). This is followed with the expansion of CSS selectors: for *A*, the *img* selector is replaced with *img[o=r]* (@4, *A*); and for *B* (@4, *B*) with *img:not([o=r])* (the CSS rule in application *B* becomes: img:not([o=r]) { border: dashed;}, meaning apply dashed border to *img* elements that do not have the *o* attribute with a value of *r*). With this change, instead of styles that are applied to HTML nodes from the other application, we have obtained CSS rules that target only the HTML nodes from their respective applications. Finally, DOM querying conflicts in JavaScript are resolved by replacing the query arguments with the same values as in the CSS replacement.

```
.................   A   .................
04 A: img { border: solid; }
05 A: .mI { width: 200px; }
21 A: var im = document.querySelectorAll('img')
29 A:<body>
30 A:  <div id="imCont">
31 A:    <img class="mI" src="img/C.jpg"/>
32 A:    <img class="mI" src="img/D.jpg"/>
33 A:  </div>
34 A:</body>
.................   B   .................
04 B: img { border: dashed; }
05 B: .mI { width: 300px; }
23 B: var im = document.querySelectorAll("img")
36 B:</div>
.................   B'   .................
B':img[o=r] { border: solid; }
```

```
B':.mI { width: 200px; }
B':img:not([o=r]) { border: dashed; }
B':.mI:not([o=r]) { width: 300px; }
B': var im = document.querySelectorAll('img[o=r]')
B': var im = document.querySelectorAll('img:not([o=r])')
B':  <div id="imCont" o=r>
B':    <img class="mI" src="img/C.jpg" o=r/>
B':    <img class="mI" src="img/D.jpg" o=r/>
B':  </div>
```

Listing 7.4: Excerpts from applications A and B and the final result when resolving selector conflicts

7.3.2 Resolving JavaScript conflicts

When resolving JavaScript conflicts, the goal is to detect and resolve JavaScript problems that arise due to *i)* global variable naming conflicts, *ii)* the modifications of the globally accessible objects that can change the behavior of additionally included code, and because of the *iii)* possible migration of source code to another server. Since conflicts can occur both dynamically and statically, as input the Algorithm 12 receives the dependency graphs and execution summaries from both applications.

The first step of the algorithm is finding conflicts related to standard global variables (lines 1–4): for each conflicted identifier, a new non-conflicting name is generated, and all usage positions of that identifier in the feature code are replaced by traversing the dependency graph (a warning is added, if this is not possible, e.g. the property name is created by string manipulations).

Next, the algorithm resolves conflicts that arise by extending built-in objects (lines 5–11). For each object extension in the feature code (line 5), the algorithm traverses all code positions in application B that iterate over the properties of extended objects and adds a statement that will skip the iteration over the properties extended by the feature. The algorithm proceeds by checking if there are any conflicts with the object extensions done in application B (lines 7–9), and if there are, the property names in the feature code are replaced with non-conflicting names. The process goes similarly for application B (lines 12–14) with the exception that there is no need for handling naming conflicts (since they have already been handled). Next, the event handler conflicts are resolved (lines 15–20) by inserting code that creates an event-handler-tracker object that keeps track of all registered handlers, replacing conflicting code

Algorithm 12 resolveJs(G_A, E_A, G_B, E_B)

```
 1: for all conflict : getStandardGlobalCnflicts(E_A, E_B) do
 2:     decl ← getDeclaration(conflict, G_A)
 3:     renameIdDependencies(decl, genName(conflict, E_A, E_B))
 4: end for
 5: for all objExt : getBuiltInObjExtensions(E_A) do
 6:     addSkipIterToPropIters(getPropIters(objExt, E_B))
 7:     if hasConflicts(objExt, E_B) then
 8:         decl ← getDeclaration(objExt, G_A)
 9:         renameIdDependencies(decl, genName(objExt, E_A, E_B))
10:     end if
11: end for
12: for all objExt : getBuiltInObjExtensions(E_B) do
13:     addSkipIterToPropIters(getPropIters(objExt, E_A))
14: end for
15: conflicts ← getConflictedHandlers(E_A, E_B)
16: if nonEmpty(conflicts) then
17:     addInitConflictHandlerObjectAsTopNode(G_B)
18:     expandWithConflictHandlerCode(conflicts, G_B)
19:     addHandlerInvokerCodeAsLastBodyNode(G_B)
20: end if
21: for all httpRequest : getServerCommunications(E_A) do
22:     logRequestInfo(httpRequest)
23: end for
```

expressions in both applications with code that reroutes the handler registration and deregistration to the event-handler-tracker, and inserting code that invokes the necessary handlers.

Finally, the problems related to server-side requests from the feature code are addressed (lines 21–23). Since these problems can not be resolved on the client-side, during the execution of the feature we gather information about each request and notify the developer, who can then manually migrate the server-side code to the new server, or can enable cross-site HTTP requests on the original server.

Example. There is one standard conflicting global variable (Listing 7.5): *SRCS* (@8, A and B), two related to extending built-in objects: *next* (@9, A) and *toggleSrc* (@17, A and @11, B), and one related to registering event-handlers: *onload* (@20, A and @22,B). According

to Algorithm 12, the global variable *SRCS* in *A* will be replaced with *r_SRCS*. Next, conflicts caused by extending built-in objects are resolved: *i)* the property *next* defined on the array prototype (@9, *A* is handled by adding a skip iteration statement to all iterations over the extended objects in application *B* (the for-in loop @15, *B*); *ii)* the property *toggleSrc* on the *HTMLImageElement* (@17, *A*) is resolved by traversing property dependencies in the feature, and replacing the identifiers with the newly generated name. Finally, the *window.onload* properties are resolved by inserting event-handler-tracker object code, and rewriting the event property access code.

```
.................    A    .................
08 A: SRCS = ["img/C.jpg", "img/D.jpg"]
09 A: Array.prototype.next = function(){...}
17 A: iProto.toggleSrc = function(s){...}
20 A:window.onload = function(){...}
24 A:this.toggleSrc(SRCS)
.................    B    .................
08 B: SRCS = ["img/P.jpg", "img/T.jpg"]
11 B: proto.toggleSrc = function (srcs, c) {...}
15 B: for(var i in history){
16 B:   var item = history[i];
22 B: window.onload = function(){...}
.................    B'   .................
B':var HNDLR = {aApp:{}, bApp:{}}
B':R_SRCS = ["img/C.jpg", "img/D.jpg"]
B':iProto.r_toggleSrc = function(s){...}
B':HNDLR.aApp.onload = function(){...}

B':this.r_toggleSrc(R_SRCS)

B': for(var i in history){
B':   if(i == 'next') continue;
B':   var item = history[i];

B': HNDLR.bApp.onload = function(){...}

B': window.onload = function() {
B':  HNDLR.aApp.onload();
B':  HNDLR.bApp.onload();
B':}
```

Listing 7.5: Excerpts from applications *A* and *B* and the final result when resolving JavaScript conflicts

7.4 Merging code

Once all conflicts have been detected and resolved, the process moves to the second phase – merging the code of the feature and the application. Algorithm 13 describes the steps necessary to merge the code of the feature with the target application.

Algorithm 13 merge(G_A, G_B, *rSelector*)

```
1: for all hChild : getHeadChildren(G_A) do
2:     appendToHeadNode(hChild, G_B)
3: end for
4: for all bChild : getBodyChildren(G_A) do
5:     appendToBodyNode(bChild, G_B)
6: end for
7: fNds ← getFeatureNodes(G_A)
8: for all pos : getStructuralSelectorsPositions(fNds, G_A) do
9:     replaceSelector(pos, generateNewQuery(pos, rSelector))
10: end for
11: moveNodes(rSelector, fNds, G_B)
12: scripts ← getFeatureScriptsAffectedPos(fSelector, rSelector, G_B)
13: updatePosition(rSelector, scripts, fNds, G_B)
```

The main idea of the algorithm is to perform the merge of the head and body nodes of each application, and then to move the feature nodes to the designated position, without introducing errors. The algorithm works by first taking the head children and the body children of the feature graph from application A and appending them to the head and the body node of application B. Next, the HTML nodes that define the feature are selected from the graph with the goal of moving them to a new position defined by the *rSelector* selector (*Reuse Position*, Figure 7.1). Some CSS selectors that apply styles to feature nodes, or JavaScript DOM queries, can depend structurally on the position of the feature nodes in the page hierarchy, which can introduce errors when the feature nodes are moved. For this reason, similar fixes as in Algorithm 11 have to be applied (lines 8–10, new selectors are created to replace possibly conflicting ones). Also, due to DOM queries, when moving feature nodes it is necessary to maintain the relative position of the feature script nodes towards the feature HTML nodes (line 11), because the source code can

have implicit dependencies towards the position in the DOM.

Example. The goal is to append the feature defined by the div node with the *imCont* id from A to the div node with the id *imCont* from B. First, the nodes style and script (starting @3 and @7, A) are appended to the head node of B, and the div node (@30, A) is appended to the body node @32, B. Next, the feature node (@30, A) is moved to the div node @33, B. This does not cause any errors, nor does it change the relative position of any feature script nodes, and the integration process is finished. Listing 7.6 shows the final result of the process (`a` labels lines from A, `b` from B, and `n` new lines that do not come from either A or B; `m` denotes that the line has been modified).

```
1b: <html>
2b:  <head>
3n:  <script>
4n:  var HNDLR={aApp:{}, bApp:{}}
5n:  </script>
6b:  <style>
7bm: img:not([o=r]) {border:dashed;}
8bm: .mI:not([o=r]) { width:  300px; }
9b:  </style>
10b: <script>
11b: SRCS=['img/P.jpg', 'img/T.jpg'],
12b: history=[], doc = document;
13b: proto=HTMLImageElement.prototype
14b: proto.toggleSrc=function(srcs,c){
15b:  this.src=this.src.indexOf(srcs[0]) == -1 ? srcs[0] : srcs[1]
16b:  history.push(this.src)
17b:  summ = {}
18b:  for(var i in history){
19n:   if(i == 'next') continue;
20b:   var item = history[i];
21b:   if(!summ[item]) summ[item] = 0
22b:   summ[item]++
23b:  }
24b:  c.textContent=JSON.stringify(summ)
25b: }
26bm: HNDLR.bApp.onload=function() {
27bm:   var im=doc.querySelectorAll("img:not(o=r)")
28b:    var inf=doc.querySelector("#info")
29b:    for(var i=0; i<im.length; i++)
30b:     im[i].onmouseover = function(){
31b:       this.toggleSrc(SRCS, inf)
32b:    }
33b:  }
34b: </script>
35a: <style>
```

```
36am: img[o=r] { border:  solid; }
37am: .r_mI[o=r] { width:  200px; }
38a:  </style>
39a:  <script>
40am: R_SRCS=['img/C.jpg','img/D.jpg']
41a:  Array.prototype.next=function(c){
42a:   var i = this.indexOf(c)
43a:   var next = 0
44a:   if(i>=0 && i<this.length-1)
45a:    next = i+1
46a:   return this[next]
47a:  }
48a:  iProto=HTMLImageElement.prototype
49am: iProto.r_toggleSrc=function(s){
50a:   this.src=s.next(this.getAttribute('src'))
51a:  }
52am: HNDLR.aApp.onload = function(){
53am:  var im = document.querySelectorAll('img[o=r]')
54a:   for(var i=0; i<im.length; i++)
55a:    im[i].onclick = function(){
56am:      this.r_toggleSrc(R_SRCS)
57a:    }
58a:   }
59a:  </script>
60b: </head>
61b: <body>
62bm: <div id='imCont'>
63b:  <img class="mI" src="img/P.jpg"/>
64b:  <img class="mI" src="img/T.jpg"/>
65b:  <id id="info"></div>
66am:  <div id='r_imCont' o=r>
67am:  <img class=r_mI src=img/C.jpg o=r>
68am:  <img class=r_mI src=img/D.jpg o=r>
69a:   </div>
70b:  </div>
71n:  <script>
72n:  window.onload=function(){
73n:   HNDLR.aApp.onload()
74n:   HNDLR.bApp.onload()
75n:  }
76n:  </script>
77b:  </body>
78b:</html>
```

Listing 7.6: Reuse Result

7.5 Verification

Once the feature code has been integrated into the code of the target application, the final step is to verify the correctness of the integration. As stated in the beginning of the chapter: the feature f_a should not operate on parts of the application originating from B, nor should features from B operate on parts of application originating from A. Since client-side web applications are UI applications, their behavior is observable as modifications to the structure and presentation of the page. For this reason, in order to verify the correctness of integration, for a particular scenario, we check all observable behaviors in B', and study both the modified HTML nodes and the code constructs causing the modification. The observable behavior is correct if the modified nodes and the code constructs are originating from the same application.

The process of verification is shown in Algorithm 14, and is executed for the scenario causing the manifestation of the feature s_a, and for all scenarios S_B capturing the behavior of application B. As input, the algorithm receives the source code of the final B' application, and the current scenario that is being verified.

Algorithm 14 verify(*code*, *s*)

```
1:  s' ← updateScenario(s)
2:  exeInfo ← executeScenario(code, s')
3:  for all ob ← getObservableBehaviors(exeInfo) do
4:      hNodes ← getAffectedConstructs(ob)
5:      constructs ← getAffectingConstructs(ob)
6:      if notFromSameApp(hNodes, constructs) then
7:          reportWarning(hNodes, constructs)
8:      end if
9:      if communicatesWithUnavailableServer(constructs) then
10:         reportWarning(constructs)
11:     end if
12: end for
```

For each scenario, the algorithm first updates the scenario info in order to compensate for the changes done in the conflict resolution phase (so that the events are executed on correct elements). Next, the scenario is executed in the context of the final B' application (line 2). The HTML nodes (line4) and code constructs (line 5) involved in each observable

behavior encountered while executing the scenario are studied. In order for the reusable behavior to be correct, both the HTML nodes and the code constructs causing the observable behaviors have to originate from the same application (lines 6–8). The algorithm also reports a warning if the application is trying to establish an HTTP connection with an unavailable server (lines 9–11).

Example. In the example applications let scenario $s_a = [\langle click, .mI \rangle]$ (click on an element specified by a CSS selector ".mI") and let $S_B = [\langle onmouseover, .mI \rangle]$ (move mouse over the element specified by a selector ".mI"). There are seven feature manifestations in f_a for s_a: four from the application of CSS rules to image elements (Listing 7.7): CSS@4 → img@31, CSS@5 → img@31, CSS@4 → img@32, CSS@5 → img@32; the registration of the *onclick* handler for both image elements @23; and the modification of the "src" attribute of the first image by executing the JavaScript assignment expression @18. Similarly, application B for the scenarios S_B has eight observable behaviors: four by applying the CSS rules @4, 5, B to nodes @34, 35, two by setting the *onmouseover* property of two image elements @26; one by modifying the "src" attribute of the first image with a JavaScript expression @12, and one by changing the value of the "textContent" property of the element @36 with a JavaScript expression @20.

```
.............   A   .............
04A:   img{border: solid; }
05A:   .mI{ width: 200px; }
18A:    this.src=s.next(this.src)
23A:    im[i].onclick = function(){...}
31A:    <img class="mI" src="img/C.jpg"/>
32A:    <img class="mI" src="img/D.jpg"/>
.............   B   .............
04B:   img{border:dashed;}
05B:   .mI{width:300px;}
12B:   this.src=this.src.indexOf(srcs[0])==-1?srcs[0]:srcs[1]
20B:    c.textContent=JSON.stringify(summ)
26B:    im[i].onmouseover = function(){...}
34B:   <img class="mI" src="img/P.jpg"/>
35B:   <img class="mI" src="img/T.jpg"/>
36B:   <div id="info"></div>
```

Listing 7.7: Excerpts from applications A and B showing code relevant for feature manifestations

Exercising the scenario s_a in the context of the resulting B' application (Listing 7.6) results with eight observable behaviors related to the

application of CSS rules to image elements (CSS@7 → img@63, CSS@8 → img@63, CSS@7 → img@64, CSS@8 → img@64, CSS@36 → img@63, CSS@37 → img@63, CSS@36 → img@64, CSS@37 → img@64); two observable behaviors by setting the *onmouseover* property @30 on the image elements @63, 64; two observable behaviors by setting the *onclick* property @55 on the image elements @67, 68; and one by modifying the "src" attribute @50 of the image element @63.

When verifying the correctness of the behavior of s_a, first we update the scenario info into $s_a = \langle click, .r_mI \rangle$ (because the class attribute was renamed), and then we check all observable behaviors. In this case all observable behaviors in B' occur correctly, every HTML node is modified by either a CSS rule or a JavaScript expression originating from the same application. The process continues similarly for S_B.

7.6 Experiments

The evaluation of the approach is based on six case study applications divided into three groups, and in each group we have one A application and one B application (Table 7.3). In the first group, we have applications 1 and 2 that were developed without third-party JavaScript libraries; in the second group applications 3 and 4 that use the most wide-spread third-party JavaScript library – jQuery; and in the third group applications 5 and 6 that were developed with the second most-wide spread JavaScript library – MooTools[1]. With these six applications we have created a set of experiments with a goal to test whether our method is capable of performing automatic feature code integration in different situations (e.g. is the process able to include a feature developed with the jQuery library into the application developed with the MooTools library, and vice versa), and in total, we investigate the 9 combinations. Based on the feature scenarios generated by the scenario generation method (Chapter 5) we have specified Selenium tests[2] that test the correctness of the features in the final application. The case study applications, their tests, and the results are available at: `www.fesb.hr/~jomaras/download/reuseCaseStudies.zip`.

Table 7.3 shows the experiment results. For each case, we present the total number of lines of code in the application from which a feature

[1] `w3techs.com/technologies/history_overview/javascript_library/all`
[2] `http://docs.seleniumhq.org/`

Table 7.2: Case study web applications. Lines of code (LOC)

#	Group	Application	LOC	Feature LOC
1	1	TinySlider	316	242
2	1	Fancy buttons	180	N/A
3	2	PrIDE	10,554	1083
4	2	Mailboxing	12,031	N/A
5	3	Tab panels	5,112	1253
6	3	Login panel	8,813	N/A

Table 7.3: Experiment results: HTML modifications (H), CSS modifications (C), JavaScript modifications (J) ; Time – process execution time in seconds

#	A	B	H (A;B)	C (A;B)	J (A;B;B')	Time
1	1	2	22;0	2;5	1;1;2	15
2	1	4	23;0	3;7	0;0;0	55
3	1	6	23;0	3;2	0;0;0	42
4	3	2	30;0	1;5	0;0;0	14
5	3	4	39;0	14;7	9;0;0	54
6	3	6	29;0	1;2	9;0;0	41
7	5	2	22;0	5;5	0;0;0	50
8	5	4	22;0	5;7	4;12;0	90
9	5	6	22;0	5;2	310;0;0	77

was extracted (A-LOC), lines of code of the application where the feature will be integrated into (B-LOC), total LOC of the feature extracted with the feature identification process (F-LOC), number of changes done to the HTML code (H), CSS code (C), and JavaScript code (J) that were performed by the process to resolve conflicts; and the total running time of the process[3]. All experiments have been performed with the Firecrow[4] tool, further described in Chapter 8, which implements the algorithms described in the paper.

In all cases the method was able to introduce a feature from one application into another. However, in order to achieve this, some modifications of the application source code were necessary. As can be seen

[3]Phantom Js 1.9, Intel Xeon 3.7 GHz, 16 GB RAM

[4]`https://github.com/jomaras/Firecrow`

from the Table 7.3 the majority of these modifications was concerned with resolving HTML naming attributes conflicts, and conflicts that arose due to overriding CSS styles. As far as JavaScript conflicts go, there are three interesting types of cases: Case 1 – both applications use the *onload* property of the *window* object to schedule the execution of a function after the page has been loaded; in order to prevent overriding, event-handling conflict code was inserted into both applications, and conflict-free handler caller code was added to the final *B'* application;and Cases 5, 6, 8 – where conflicts in the library methods have been found and fixed; and Case 9 which has a much larger number of JavaScript modifications, compared to other cases. Case 9 deals with reusing a feature from an application developed with the MooTools library into an application also developed with the MooTools library. The MooTools library creates a large number of global JavaScript variables, and makes considerable modifications to the built-in objects. For this reason, when performing feature integration, a large number of conflicts is detected and handled. From a practical perspective, it does not make much sense to integrate parts of the library, obtained from feature code, with the full library. However, in this experiment, our goal was to test the feature integration process, and this case shows that the process is able to accurately handle a large number of conflicts.

7.7 Conclusion

This chapter describes how to achieve automatic integration of features from one client-side web application into an already existing client-side application, in order to achieve feature reuse. We have specified what exactly feature integration is, and when can it be considered successful. Naturally, when attempting to introduce code from one application into another application a number of conflicts can occur – we have identified different types of conflicts and have developed algorithms capable of detecting and resolving them. Once the conflicts are resolved, in order to achieve reuse, we have defined an algorithm that merges the code of the feature with the target application. We have also defined an algorithm that verifies the correctness of the process. In the end, by testing the method on a suite of non-trivial applications, we have shown that the method is capable of identifying and handling conflicts, and performing feature integration.

Chapter 8

Firecrow tool

In order to support the process of of automatic feature reuse, we have developed a prototype tool – Firecrow[1]. In this chapter, we describe different subsystems composing the tool, and discuss their roles in supporting the various steps of the process.

8.1 Tool organization

The Firecrow tool is composed out of five subsystems: *i) DoppelBrowser*, a JavaScript library that processes and interprets web application code, creates a dependency graph, and is capable of creating execution summaries for a particular execution of a client-side web application; *ii) Feature Locator*, a JavaScript library that traverses the dependency graph, analyzes the execution summary, and identifies the code implementing a certain feature; *iii) Scenario Generator*, an application for automatic generation of scenarios; *iv) Feature Integrator*, a JavaScript library that locates and fixes potential feature integration errors, and performs the merging of the feature code and target application code. The functionalities provided by these subsystems can be used from a *Firefox plugin*.

Figure 8.1 shows a screenshot of the tool, when used as a plugin to the Firefox's Developer Tools[2]. Mark A shows the structure of the feature chosen for reuse, and mark B the tool.

[1] https://github.com/jomaras/Firecrow
[2] https://developer.mozilla.org/en/docs/Tools

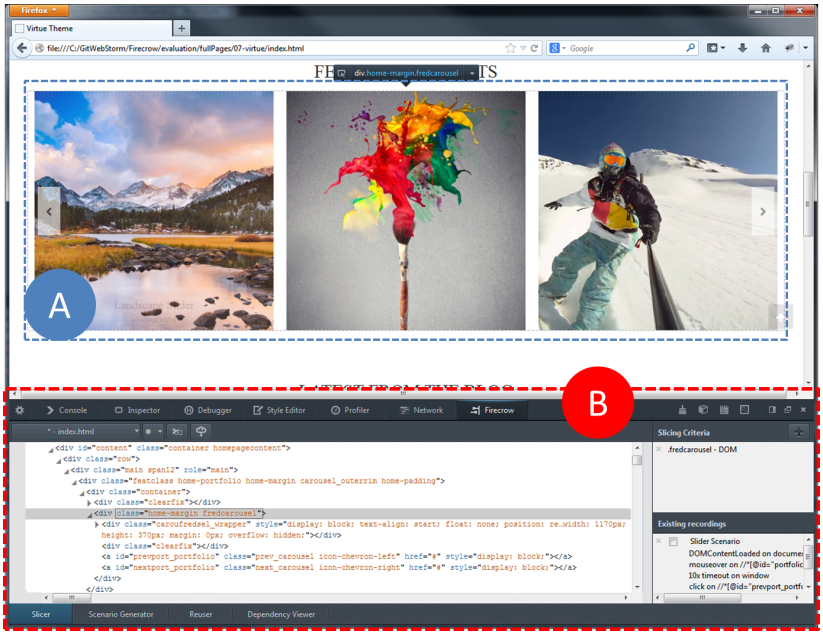


Figure 8.1: A screenshot of the Firecrow tool used within the Firefox browser; A – structure of the feature chosen for extraction; B – Firecrow within Firefox's Web Developer Tools

8.1.1 DoppelBrowser

DoppelBrowser is a JavaScript library that evaluates web application code according to standard rules of client-side web application evaluation. It includes a newly developed JavaScript interpreter that, besides evaluating JavaScript code, also keeps track of relationships between variable values and code expressions from where the values originate from. This additional interpreter feature enables us to construct a client-side dependency graph (Chapter 4) that accurately captures the dependencies that exist in a client-side application. In addition, the library can be used to gather execution summaries that capture important run-time information used in the process of automatic testing, as well as in the reuse process itself – the *DoppelBrowser* library is an integral component in all steps of the automatic reuse process. The library is browser agnos-

tic, and we have used it to evaluate web application code from different browsers: Firefox, Chrome, Safari, and PhantomJs browsers.

8.1.2 Feature Locator

Feature Locator is a JavaScript library developed with the purpose of identifying code that implements a particular feature. It works by analyzing the dependency graph and the execution summary gathered by the *DoppelBrowser*, and performs the graph marking algorithms (Chapter 6.5). As the *DoppelBrowser* library, it can be run inside different browsers.

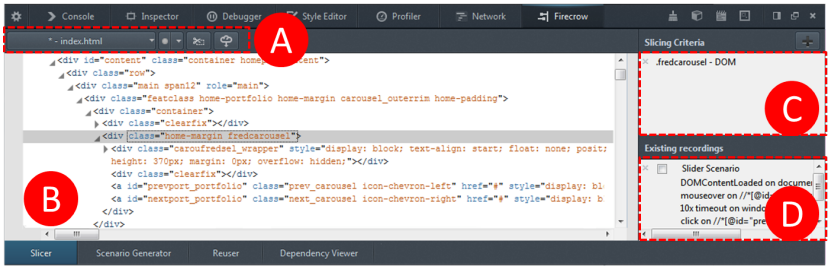


Figure 8.2: Feature Locator used as a Firefox plugin. A – action toolbar; B – DOM viewer; C – Slicing Criteria; D – Scenario description

Figure 8.2 shows the UI of the *Feature Locator* subsystem, when accessed through a Firefox plugin. Mark A indicates the toolbar that allows browsing though application source code, recording scenarios, and initiating the feature identification process; mark B shows the DOM viewer which enables easy specification of parts of web page structure where the feature manifests; mark C shows a container with slicing criteria (either points in the source code, or CSS selectors that point to certain parts of the page); mark D shows a list of all events for the recorded scenario.

8.1.3 Scenario Generator

Scenario Generator is a stand-alone application that makes use of the *DoppelBrowser* library. It implements the algorithms described in Chapter 5. The tool systematically explores the value and the event space of

a client-side web application and generates scenarios that cover the behaviors of a client-side web application.

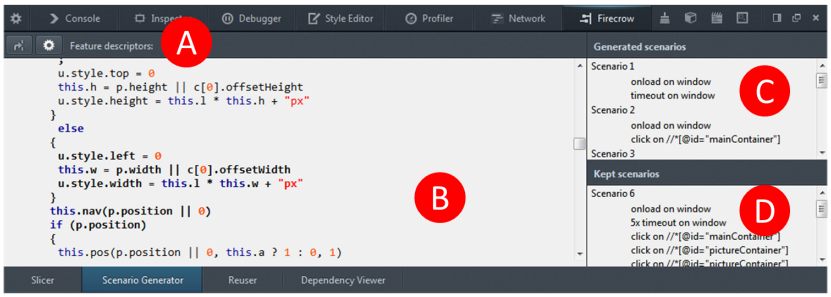


Figure 8.3: Scenario Generator used as a Firefox plugin. A – toolbar; B – web application code; C – Generated Scenarios; D – Kept scenario

Figure 8.3 shows the UI of the Scenario Generator subsystem, when accessed though a Firefox plugin. Mark A indicates the toolbar that is used to specify feature selectors, prioritization function, and start the process; mark B shows the source code of the application, where the bold text denotes parts of the source code that was executed by at least one scenario; mark C shows all scenarios that were generated in the process, while mark D shows the scenarios that were kept after the scenario filtering phase.

8.1.4 Feature Integrator

The *Feature Integrator* is a JavaScript library that provides the functionality of integrating feature code identified with the *Feature Locator* subsystem into an already existing application. The subsystem implements algorithms presented in Chapter 7.

Figure 8.4 shows the UI of the Feature Integrator subsystem, used as a Firefox plugin. Marks A, B, and C indicate the toolbar that allows the user to select the feature page and specify parts of the page structure where the feature manifests, and the page where the feature will be reused into, as well as the exact position in the page structure where the feature will be integrated; mark D shows the feature scenarios, and mark E the scenarios of the target application.

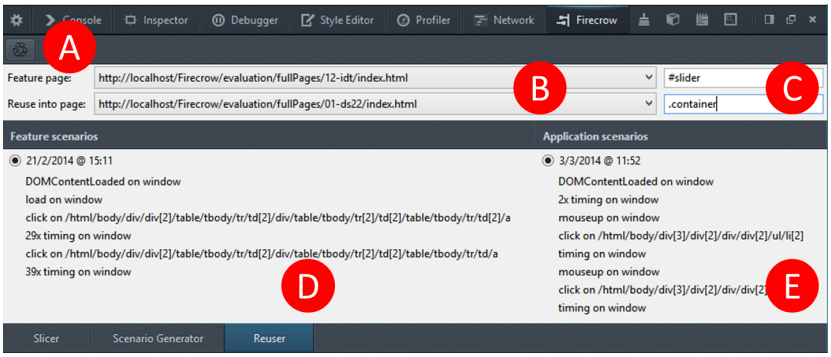


Figure 8.4: Feature Integrator used as a Firefox plugin. A – toolbar; B – Setting the feature and the target page; C – Specifying the parts of the page structure that define the feature and the position where the feature will be reused; D – Feature scenarios; E – application scenarios

8.2 Conclusion

In this chapter, we have given an overall description of the Firecrow tool, a tool that implements the processes and algorithms described throughout the thesis. The tool was developed in order to provide easy access to the functionalities of scenario generation, feature identification, and feature integration. Since the tool is developed as a JavaScript library, it can be adjusted to allow usage from browsers other than Firefox (e.g. Chrome, Opera). In addition, the tool is extensible, and some of the extensions that we are considering are debugging and program understanding.

Chapter 9

Related Work

Throughout this thesis, we have described the process of automatic feature reuse in client-side web application development. We have discussed different steps needed to support reuse: automatic scenario generation (Chapter 5); locating the implementation details of particular features and performing slicing in order to extract the feature code (Chapters 4 and 6); and integrating the extracted feature code into an already existing application (Chapter 7). In this chapter, we present work related to each of these sub-problems, as well as to the general problem of software reuse.

9.1 Software Reuse

Software reuse, as the process of creating new software systems from existing software artifacts, has long been advocated a way to reduce defect density and increase developer productivity. Overall, the approaches to software reuse can be divided into two groups [29]: *i)* preplanned reuse approaches, where software artifacts that will be reused are explicitly constructed with reuse in mind; and *ii)* pragmatic approaches, where reused software artifacts are not necessarily constructed for reuse.

Over the years, a number of approaches that support reuse have been developed: Hunter Gatherer [53], Internet Scrapbook [57], HTMLviewPad [58], and Web Mashups [44] in the web application domain; and G&P [31], Jigsaw [16], and CodeGenie [39] in the Java domain.

HunterGatherer [53], Internet Scrapbook [57], and HTMLviewPad [58] are similar approaches mostly related to clipping and reusing fragments of Web pages. They enable the creation of personalized pages that aggregate data or "information components" from different sources. Since these approaches were developed in 1990's and early 2000, when web page development was not so dynamic on the client-side, their usability in the current web development is quite limited. These approaches are mostly limited to reusing HTML elements such as text fragments and forms, and make no attempts to also include CSS and JavaScript.

Further work has lead to the development of Web Mashups [44], which are web applications that combine information and services from multiple sources on the web. The main advantage of Mashups is that they enable the creation of new applications by integrating services offered by third-party providers. The main difference between Mashups and our approach is that Mashups address reuse on a service-level, while we specifically target reuse on the code level.

In the more general domain of Java applications, G&P [31] is a reuse environment composed of two tools, Gilligan and Procrustes, that facilitates pragmatic reuse tasks. Gilligan allows the developer to investigate dependencies from a desired functionality and to construct a plan for its reuse, while Procrustes automatically extracts the relevant code from the originating system, transforms it to reduce the compilation errors and inserts it into the developer's system. In this domain there is also a tool called Jigsaw [16] which facilitates small-scale reuse of source code. The main difference between our approaches is that G&P and Jigsaw are approaches that statically analyze Java applications. While the ideas and end goals are similar, their methods can not be used in the highly dynamic, multi-paradigm environment of client-side web applications.

Recently, an increasing amount of open source code is being made available on the Internet, and a number of approaches that search the openly available code repositories have been developed. CodeGenie [39] presents a tool and an approach to code search and pragmatic reuse based on test cases. The test cases have two main purposes: *i)* they define the behavior of the desired functionality, and *ii)* they test the reuse result for suitability in the local context. First, the code search for the functionality described by the tests is performed, next the matching source code is sliced from the originating system, and finally the sliced code can be integrated into the target application. Compared to our approach, CodeGenie is focused on reusing auxiliary functions, and is not

designed to support reuse of higher-level, complex UI features. Similar to G&P, CodeGenie is designed for the domain of Java applications.

9.2 Automatic Testing of Web Applications

One of the contributions of this thesis is a method for automatic generation of scenarios. This is closely related to the discipline of automatic testing for web applications.

In [43], Mesbah et al. describe their approach for automatic testing. The method is based on a crawler [42] that infers a state-flow graph for all client-side user interface states. New states and transitions are created by executing existing event handlers, analyzing the structure of the application and determining if it is changed enough to warrant a new state. The crawling phase is directed either with randomly generated input values or with user-specified values. Various errors are detected (DOM validity, error messages, etc.) by analyzing possible client-side user interface states.

Saxena et al. [52] present a method and a tool – Kudzu. The approach explores the application's event space with GUI exploration (searches the space of all event sequences with a random exploration strategy), and the application's value space by using dynamic symbolic execution. In the process, they have developed a string constraint solver capable of taking into account the specifics of string constraints present in JavaScript programs.

Artemis [5] is an approach for feedback directed testing of JavaScript applications from which we have derived most insights when developing our approach. The approach is based on dynamic analysis of web application execution – the application execution is monitored and all event registrations logged. New test cases are created by extending already existing tests with event registrations and by generating variants of the event input parameters. For generating new event input parameters they use randomly chosen values, and constants collected during the dynamic execution. They also introduce prioritization functions which influence the order in which generated test cases are analyzed.

None of the introduced client-side web application testing approaches enable developers to target specific client-side features, nor do they provide filtering of generated scenarios in order to minimize the number of usage scenarios. Also, in order to improve coverage, we use

the systematic exploration of the application's event-space (similar to Artemis [5]) and combine it with symbolic execution (similar to Saxena's approach [52]). On top of this, we track application dependencies by the means of a dependency graph (Chapter 4), which enables us to accurately capture dependencies between different events, and to create event chains.

In the domain of testing server-side web applications, there exists the SWAT tool [3], which uses search-based testing. In their approach, random inputs to the web application are generated with additionally incorporated constant seeding (gathered by statically analyzing the source code), and by dynamically mining values from the execution. Although some parts of the approach could be adopted to fit the domain of client-side applications, their method is specifically developed to deal with constraints inherent in server-side applications.

9.3 Feature Location

In our approach, we dynamically analyze the application execution and the dependency graph to locate the code that implements a particular feature. In this section, we present some of the most important dynamic feature location techniques, as well as the use of dependency graphs for feature location.

Chen and Rajlich [13] present a version of a system dependence graph (SDG) [33] called an abstract system dependence graph (ASDG) is presented. The ASDG is used in a feature location computer-assisted search process. In each step of the search, one component (functions and global variables) is chosen for visit. All visited components and their neighbors constitute a search graph. Each visit to a component expands the search graph, and the process continues until all components implementing a feature are located.

One of the earlies dynamic feature location techniques, proposed by Wilde [63], was based on the idea of comparing execution traces obtained by exercising the feature of interest with those obtained while the feature was not exercised. The execution traces are obtained by executing sets of test cases that invoke application features. For each feature, the approach then groups components into sets according to how specific the components are to the feature.

Eisenbarth and Koschke [20] have developed a semiautomatic tech-

nique that reconstructs the mapping for features that are triggered by the user and exhibit an observable behavior. The method works by analyzing execution traces and static program dependence graphs whose nodes are methods, data fields, classes, etc. and whose edges are function calls, data access links and other types of relationships obtained by static analysis. Then they use formal concept analysis, where computation units are objects, scenarios attributes, and where relationships indicate if an object was visited during a scenario, to create a concept lattice, which is then manually analyzed by the analyst. Information about how specific the computational unit to a feature are then derived.

Compared to these methods, our method takes into account the fact that client-side web applications are UI applications where features manifest on certain parts of the web page structure. Then, by analyzing application executions in which the feature manifests and traversing the dependency graph, we can locate the code that implements the feature that manifests on a certain part of web page structure, for a particular scenario.

There are also two tools that facilitate the understanding of dynamic web page behavior: Script InSight [40] and FireCrystal [45]. Script InSight helps to relate the elements in the browser with the lower-level JavaScript syntax. It uses the information gathered during the script's execution to build a dynamic, context-sensitive, control-flow model that summarizes tracing information. FireCrystal facilitates the understanding of interactive behaviors in dynamic web pages by recording interactions and logging information about DOM changes, user input events, and JavaScript executions. After the recording phase, the user can use an execution time-line to see the code that is of interest for the particular behavior. Compared to our approach they make no attempts to track data dependencies between different code expressions, nor to identify individual features in the analyzed code.

9.4 Program Slicing

Our work is closely related to program slicing, defined by Weiser [62] as a method that, for a given subset of a program's behavior, reduces that program to a minimal form which still produces that behavior. In its original form, a program is sliced statically, for all possible program inputs. Static slicing can be difficult, and can lead to slices that are larger

than necessary, especially in the case of pointer usage. Further research has lead to development of dynamic slicing [2] in which a program slice is composed of statements that influence the value of a variable occurrence for specific program inputs – only the dependencies that occur in a specific execution of a program are studied.

Program slicing is usually based on some form of a dependency graph that captures dependencies between code constructs. Depending on the area of application, it can have different forms: a Flow Graph in Weiser's original form, a Program Dependence Graph (PDG) [34] representing both data and control dependencies for each evaluated expression, or a System Dependence Graph (SDG) [33] which extends the PDG to support procedure calls rather than only monolithic programs. The SDG has also been later expanded in order to support object-oriented programs [38]. None of these graphs is fully suitable to support a multi-language dynamic environment that is the client-side of the web application.

In the web domain, Tonella and Ricca [59] define web application slicing as a process which results in a portion of a web application which still exhibits the same behavior as the initial application in terms of information of interest to the user. They present a technique for web application slicing in the presence of dynamic code generation by building an SDG for server-side web applications. Even though the server-side and the client-side applications are parts of the same whole, they are based on different development paradigms, and cannot be treated equally. Nowadays, the client-side applications are highly dynamic, event-driven environments where the interplay of three different languages (HTML, JavaScript, and CSS) produces the end result displayed in the browser. For this reason, server-side analysis techniques, such as Tonella and Ricca's, can not be applied to client-side applications.

Chapter 10

Conclusion

In this thesis, we have described an automated approach to feature reuse in client-side web application development. We have specified a method composed of three distinct phases: *i)* automatic scenario generation, *ii)* identifying feature implementation details, and *iii)* integrating the feature code into an existing application, thereby achieving reuse. In this chapter, we summarize the results with respect to the research questions and contributions presented in Chapter 1, and we present possible future work.

10.1 Identification of feature implementation details

RQ1*: How can we identify the subset of the web application source code and resources that implement a particular feature?*

In order to answer this question, we have studied the state of the art and state of the practice of feature location and program slicing techniques. We have defined a client-side dependency graph (Chapter 4) that is capable of capturing the dependencies that exist in a client-side web application, and have specified an algorithm that creates the dependency graph by dynamically analyzing application execution.

We have also defined a process of identifying feature implementation details (Chapter 6). The process dynamically analyzes application execution and identifies points in the execution where the feature mani-

fests. Based on these feature manifestation points the feature code and resources are identified by traversing the dependency graph. In order to validate the correctness of the feature identification process, we have performed three sets of experiments (Section 6.6). In all cases, the scenarios that capture the behavior of a feature are specified as tests, and the identification process is considered successful if the extracted code is able to pass the same test as the original code. We have also compared our approach with profiling, which is a straightforward extraction approach where all executed expressions are treated as important. The experiments have shown that the process is able to successfully identify feature code (i.e. the identified and extracted code can pass the same tests as the original code), and that considerable savings, in terms of code lines and improved performance can be achieved.

10.2 Integration of feature code

RQ2: *How can we introduce the source code and resources of a feature into an already existing application, without breaking the functionality of neither the feature nor the target application?*

Once the code of the feature has been identified, in order to achieve reuse we have to integrate it into an existing application. Naively merging the code of the feature with the code of the target application can lead to a number of different conflicts that have to be detected and resolved. We have identified the types of conflicts that can occur, and have developed algorithms that detect and resolve those conflicts. After conflict resolution, in order to achieve reuse, we often have to integrate the code of the feature with the code of the target application. We have defined algorithms capable of integrating code, as well as verifying the correctness of that integration. The process has been tested on a suite of web applications, and the evaluation has shown that the process can introduce feature code into an already existing application.

10.3 Automatic Scenario Generation

RQ3: *How can we automatically generate scenarios that cause the manifestation of a client-side feature?*

Both the method for identifying implementation details of an individual feature and the method for integrating feature code rely on the

existence of scenarios that capture the behavior of either the feature or the whole application. Specifying these scenarios manually can be a difficult and error prone activity. For this reason, we have developed an automatic method for scenario generation. The method works by systematically exploring the event and value space of the application. For some applications this causes the exploration of a large number of scenarios. For this reason, our method also includes the scenario filtering phase, where scenarios that do not contribute to the overall application coverage are removed. We have evaluated the method based on four experiments, which have shown that the method is able to generate scenarios that target specific application features, and that an increased coverage can be achieved, when compared to other, similar approaches.

10.4 Future Work

In this section, we discuss possible extensions to the research presented in the thesis.

10.4.1 The client-side dependency graph

One contribution of this thesis is the client-side dependency graph, which we have used to establish dependencies between the execution of events in order to create scenarios that achieve high coverage, to identify the code of individual features, and to establish and resolve conflicts that occur when integrating feature code into an already existing application. These are only some of the usages for the dependency graph. Since the dependency graph accurately captures dependencies that exist in an application, it can also be used to facilitate code understanding, dependency analysis, and debugging. One possible direction in further research is to investigate some of these possibilities, especially in the area of facilitating the understanding of dynamic behaviors in client-side web applications.

10.4.2 Automatic Scenario Generation

In this thesis, we have defined a method for automatic generation of feature scenarios that creates high-coverage scenarios by systematically exploring the event and value space of the application. In our approach,

we consider all event types and all created event-chains as equally probable. One idea for possible improvement is to study both the event types and event chains that occur when users interact with the application, and then create heuristics that would increase the achieved code coverage.

We also plan to perform an empirical investigation to compare the scenarios generated by our method with the scenarios specified by application users and developers. This would help us to evaluate the quality of generated scenarios, and it would possibly provide further insights that could be used to improve our automatic scenario generation method.

10.4.3 Identifying Feature Code

The feature identification process, based on the feature manifestation points and the dependency graph, identifies the code of the target feature. Similar techniques could also be used to identify code based on some other criteria, e.g. by substituting feature manifestation points with error manifestation points, a technique for identifying code that leads to an error could be developed. For this reason, we plan to investigate the application of the feature identification process to debugging.

Apart from debugging, the fact that we are able to identify code of all application features could be used to facilitate a number of software engineering activities, for example, feature understanding or software measurement (e.g. metrics that capture application complexity or maintainability could be derived). We plan to study some of these possibilities.

During our experiments (Chapter 6) we have noticed that the test web applications contain more code than is actually needed by their behavior. Since significant savings, in terms of code size and increased performance, can be made by removing unnecessary code, one of our plans is to make a wide, empirical study to investigate do web applications, in general, include more code than actually needed.

10.4.4 Firecrow

The algorithms and processes used to implement automatic feature reuse are currently implemented by our tool – Firecrow (Chapter 8), which is a JavaScript library. While this offers benefits in terms of easy extensibility and the ability to mimic different browsers, it also suffers from performance problems. For this reason, as part of future work, we plan

to implement the developed processes and algorithms directly into open-source browsers (e.g. Firefox, Chrome). This would enable us to perform extensive experiments on the most complex currently available applications.

10.4.5 Extending the approach to server-side applications

Web applications are composed out of two distinct parts: the server-side, which is a usually a procedural application implementing data-access and business logic, and the client-side, an event-driven application implementing the UI. In this thesis, we have developed methods for automatic reuse on the client-side. However, since the client-side and server-side are parts of the same whole, in order to increase the usability of the developed methods, we plan to extend the approach to also include the server-side. In order to do this, we will have to make adjustments to the dependency graph, and extend the algorithms for automatic scenario generation, feature identification, and feature integration in a way that they also take into account the specifics of the server-side.

10.4.6 Extending the approach to other domains

Even though the processes described in this thesis were specifically developed to address the domain of client-side web applications, some of the ideas could also be adapted to fit other, similar domains. For example, other UI domains, such as the domain of mobile applications, or the domain of standard desktop applications, have certain underlying principles that make them similar to the client-side web application domain. For this reason, we think that one viable aspect of future work could be the modifications of the developed procedures to other domains, or even the development of a general framework for automatic feature reuse.

Bibliography

[1] IEEE standard for software and system test documentation. *IEEE Std. 829-2008*, 2008.

[2] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Notices*, volume 25, pages 246–256. ACM, 1990.

[3] Nadia Alshahwan and Mark Harman. Automated web application testing using search based software engineering. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 3–12. IEEE Computer Society, 2011.

[4] Saswat Anand, Edmund Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, et al. An orchestrated survey on automated software test case generation. *Journal of Systems and Software*, 2013.

[5] Shay Artzi, Julian Dolby, Simon H. Jensen, Anders. Møller, and Frank Tip. A framework for automated testing of Javascript web applications. In *Software Engineering, ICSE 2011, 33rd International Conference on*, pages 571–580. ACM, 2011.

[6] Thomas Ball. The concept of dynamic analysis. In *Software Engineering – ESEC/FSE 1999*, pages 216–234. Springer, 1999.

[7] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. How reuse influences productivity in object-oriented systems. *Communications of the ACM*, 39(10):104–116, 1996.

[8] Boris Beizer. *Software testing techniques*. Dreamtech Press, 2003.

[9] Ted J. Biggerstaff. The library scaling problem and the limits of concrete component reuse. In *Software Reuse: Advances in Software Reusability, 1994. Proceedings., Third International Conference on*, pages 102–109. IEEE, 1994.

[10] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering*, pages 482–498. IEEE Computer Society Press, 1993.

[11] Barry Boehm. Managing software productivity and reuse. *Computer*, 32(9):111–113, 1999.

[12] Frederick P. Brooks Jr. No silver bullet-essence and accidents of software engineering. *IEEE computer*, 20(4):10–19, 1987.

[13] Kunrong Chen and Václav Rajlich. Case study of feature location using dependence graph. In *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*, pages 241–247. IEEE, 2000.

[14] Thomas A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[15] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on*, 35(5):684–702, 2009.

[16] Rylan Cottrell, Robert J. Walker, and Jorg Denzinger. Jigsaw: a tool for the small-scale reuse of source code. In *ICSE*, pages 933–934. ACM, 2008.

[17] Ole-Johan Dahl and Kristen Nygaard. Simula: an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[18] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2011.

[19] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[20] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *Software Engineering, IEEE Transactions on*, 29(3):210–224, 2003.

[21] Michael Ellims, James Bridges, and Darrel C. Ince. The economics of unit testing. *Empirical Software Engineering*, 11(1):5–31, 2006.

[22] Richard K. Fjeldstad and William T. Hamlen. Application program maintenance study: Report to our respondents. *Proceedings Guide*, 48, 1983.

[23] William B. Frakes and Giancarlo Succi. An industrial study of reuse, quality, and productivity. *Journal of Systems and Software*, 57(2):99–106, 2001.

[24] John E. Gaffney Jr and R. D. Cruickshank. A general economics model of software reuse. In *Proceedings of the 14th international conference on Software engineering*, pages 327–337. ACM, 1992.

[25] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *Software, IEEE*, 12(6):17–26, 1995.

[26] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[27] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. Microsoft Research, 2008.

[28] Robert J. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2(1):33–53, 1995.

[29] Reid Holmes. *Pragmatic Software Reuse*. PhD thesis, University of Calgary, Canada, 2008.

[30] Reid Holmes and Robert J. Walker. Supporting the investigation and planning of pragmatic reuse tasks. In *Proceedings of the 29th international conference on Software Engineering*, pages 447–457. IEEE Computer Society, 2007.

[31] Reid Holmes and Robert J. Walker. Semi-Automating Pragmatic Reuse Tasks. Automated Software Engineering, pages 481–482. IEEE Computer Society, 2008.

[32] Reid Holmes and Robert J. Walker. Systematizing pragmatic software reuse. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(4):20, 2012.

[33] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[34] Susane Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, 1989.

[35] Narendra Jussien, Guillaume Rochart, and Xavier Lorca. The CHOCO constraint programming solver. In *CPAIOR'08 workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, 2008.

[36] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[37] Charles W. Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

[38] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *Software Engineering, 1996., Proceedings of the 18th International Conference on*, pages 495–505. IEEE, 1996.

[39] Otávio Augusto Lazzarini Lemos, Sushil Bajracharya, Joel Ossher, Paulo Cesar Masiero, and Cristina Lopes. Applying test-driven code search to the reuse of auxiliary functionality. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 476–482. ACM, 2009.

[40] Peng Li and Eric Wohlstadter. Script insight: Using models to explore Javascript code from the browser view. In *International Conference on Web Engineering*, pages 260–274, 2009.

[41] M. Douglas McIlroy, JM. Buxton, Peter Naur, and Brian Randell. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98. sn, 1968.

[42] Ali Mesbah, Engin Bozdag, and Arie van Deursen. Crawling ajax by inferring user interface state changes. In *Web Engineering, 2008. ICWE'08. Eighth International Conference on*, pages 122–134. IEEE, 2008.

[43] Ali Mesbah, Arie van Deursen, and Danny Roest. Invariant-based automatic testing of modern web applications. *Software Engineering, IEEE Transactions on*, 38(1):35–53, 2012.

[44] San Murugesan. Understanding web 2.0. *IT professional*, 9(4):34–41, 2007.

[45] Stephen Oney and Brad Myers. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 105–108. IEEE Computer Society, 2009.

[46] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *ACM Sigplan Notices*, volume 19, pages 177–184. ACM, 1984.

[47] David Lorge Parnas. On the design and development of program families. *Software Engineering, IEEE Transactions on*, (1):1–9, 1976.

[48] Denys Poshyvanyk, Y. G. Guéhéneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *Software Engineering, IEEE Transactions on*, 33(6):420–432, 2007.

[49] Rubén Prieto-Díaz. Status report: Software reusability. *Software, IEEE*, 10(3):61–66, 1993.

[50] Václav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 271–278. IEEE, 2002.

[51] Mary Beth Rosson and John M Carroll. The reuse of uses in Smalltalk programming. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 3(3):219–253, 1996.

[52] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for Javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 513–528. IEEE, 2010.

[53] M.C. Schraefel, Yuxiang Zhu, David Modjeska, Daniel Wigdor, and Shengdong Zhao. Hunter Gatherer: Interaction Support for the Creation and Management of Within-Web-Page Collections. World Wide Web, pages 172–181, 2002.

[54] Koushik Sen, Darko Marinov, and Gul Agha. *CUTE: a concolic unit testing engine for C*, volume 30. ACM, 2005.

[55] Thomas A. Standish. An essay on software reuse. *Software Engineering, IEEE Transactions on*, (5):494–497, 1984.

[56] Giancarlo Succi, Luigi Benedicenti, and Tullio Vernazza. Analysis of the effects of software reuse on customer satisfaction in an RPG environment. *Software Engineering, IEEE Transactions on*, 27(5):473–479, 2001.

[57] Atsushi Sugiura and Yoshiyuki Koseki. Internet scrapbook: creating personalized world wide web pages. Human Computer Interaction, pages 343–344. ACM, 1997.

[58] Yuzuru Tanaka, Kimihito Ito, and Jun Fujima. Meme Media for Clipping and Combining Web Resources. *World Wide Web*, 9:117–142, 2006.

[59] Paolo Tonella and Filippo Ricca. Web Application Slicing in Presence of Dynamic Code Generation. *Automated Software Engineering*, 12(2):259–288, 2005.

[60] Will Tracz. Where does reuse start? *ACM SIGSOFT Software Engineering Notes*, 15(2):42–46, 1990.

[61] Jilles Van Gurp and Jan Bosch. Design erosion: problems and causes. *Journal of systems and software*, 61(2):105–119, 2002.

[62] Mark Weiser. Program slicing. *Software Engineering, IEEE Transactions on*, (4):352–357, 1984.

[63] Norman Wilde and Michael C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.