ABSTRACT

Title of dissertation: A FRAMEWORK FOR DETECTING AND DIAGNOSING CONFIGURATION FAULTS IN WEB APPLICATIONS

Cyntrica N. Eaton, Doctor of Philosophy, 2007

Dissertation directed by: Professor Atif Memon
Department of Computer Science

Software portability is a key concern when target operational environments are highly configurable; variations in configuration settings can significantly impact software correctness. While portability is key for a wide range of software types, it is a significant challenge in web application development. The client configuration used to navigate and interact with web content is known to be an important factor in the subsequent quality of deployed web applications. With the widespread use of diverse, heterogeneous web client configurations, the results of web application deployment can vary unpredictably among users. Given existing approaches and limited development resources, attempting to develop web applications that are viewable, functional, and portable for the vast web configuration space is a significant undertaking. As a result, faults that only surface in precise configurations, termed *configuration faults*, have the potential to escape detection until web applications are fielded.

This dissertation presents an automated, model-based framework that uses static analysis to detect and diagnose web configuration faults. This approach overcomes the limitations of current techniques by featuring an extensible model of the configuration space that enables efficient portability analysis across the vast array of client environments. The basic idea behind this approach is that source code fragments (*i.e.*, HTML tags and CSS rules) embedded in web application source code adversely impact portability of web applications when they are unsupported in target client configurations; without proper support, the source code is either processed incorrectly or ignored, resulting in configuration faults. Using static analysis, configuration fault detection is performed by applying a model of the web application source against knowledge of support criteria; any unsupported source code detected is considered an index to potential configuration faults. In the effort to fully exploit this approach, improve practicality, and maximize fault detection efficiency, manual and automated approaches to knowledge acquisition have been implemented, variations of web application and client support knowledge models have been investigated, and visualization of configuration fault detection results has been explored. To optimize the automated acquisition of support knowledge, alternate learning strategies have been empirically investigated and provisions for capturing tag interaction have been integrated into the process.

A FRAMEWORK FOR DETECTING AND DIAGNOSING CONFIGURATION FAULTS IN WEB APPLICATIONS

by

Cyntrica N. Eaton

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2007

Acknowledgments

I have the utmost appreciation, respect, and admiration for my advisor, Dr. Atif Memon, and I am *truly* grateful that I had the opportunity to work with him. His patience and guidance were key in this experience and I can never thank him enough for his open door, open ear, and well placed pep talks.

I would like to take this opportunity to thank my preliminary and final examination committee members, Dr. Atif Memon, Dr. Rance Cleaveland, Dr. William Gasarch, Dr. Brian Hunt, and Dr. Vibha Sazawal for their feedback and suggestions. I appreciate the time and effort you devoted to reading my drafts and helping me improve my work.

Where would I be without my Mom, Cynthia Eaton, my Dad, Rickey Eaton, and my Maternal Grandmother, Blondell Hardie! You guys will never know the extent of my gratitude for raising me in a loving environment, providing me with all I need to thrive, and most importantly, for allowing me the freedom to plot my own path. Thanks for all of your support over the years.

A special thanks goes out to my sister-friends Chakeita Jackson, Tracey Taylor, Erika Thompson, and Irvinia Jackson. I truly love you all and I really appreciate the laughter, heart-to-hearts, and support throughout this journey and beyond! To Dr. Johnetta Davis, Dr. Angela Grant, Hattie Redd, and Tamara Washington, thanks for being excellent mentors and providing me with a blueprint.

To my STAND (Science and Technology: Addressing the Need for Diversity) Family, Joelle Davis Carter, Alice Bishop, and Tamara Singleton, you are an amazing group of women and I feel honored and blessed to work with you. To my remaining Math SPIRAL (Summer Program in Research and Learning) Family, Dr. Marshall Cohen and Dr. Leon Woods, I've enjoyed working with each of you for the last two summers and I appreciate the encouragement.

I have to thank my uber-talented sisters in research, Jaymie Strecker, Penelope Brooks, and Xun Yuan for reading my drafts, sitting through practice talks, and giving me useful feedback. I've enjoyed working with each of you and I wish you all the best in the future!

In closing, I want to say that there is not enough room to list everyone who has touched my life and impacted me in a positive way. For everyone who prayed for me, encouraged me, and supported my endeavors, please accept my sincerest gratitude.

Table of Contents

List of Tables vii

List of Figures viii

List of Abbreviations 1

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Research Approach . . . 5
- 1.3 Framework Design Considerations . . . 6
- 1.4 Challenges in Attaining and Applying Source Support Knowledge . . . 7
- 1.5 Thesis Contributions . . . 10
- 1.6 Dissertation Structure . . . 11

2 Background and Related Work 12
- 2.1 Web Applications and the Browser Wars . . . 12
  - 2.1.1 HTML . . . 13
  - 2.1.2 CSS . . . 14
  - 2.1.3 The Browser Wars . . . 15
  - 2.1.4 Definitions . . . 16
- 2.2 Related Work . . . 19
  - 2.2.1 Web Portability Analysis . . . 20
    - 2.2.1.1 Manual and Automated Execution-based Approach . 20
    - 2.2.1.2 Lookup-based Approach . . . 24
    - 2.2.1.3 Source Code Standardization Approach . . . 25
  - 2.2.2 Web Testing . . . 26
  - 2.2.3 Portability . . . 27
  - 2.2.4 Fault Isolation . . . 28
  - 2.2.5 Machine Learning in Software Fault Detection . . . 31

3 General Framework Architecture 33
- 3.1 Framework Overview . . . 34
- 3.2 Manual and Automated Execution-based Approach . . . 35
- 3.3 HTML Lookup Techniques . . . 36
- 3.4 Source Code Standardization . . . 37

4 Initial Implementation 38
- 4.1 General Framework Instantiation . . . 38
- 4.2 Inductive Model . . . 41
  - 4.2.1 Modeling Client Configurations . . . 41
  - 4.2.2 Modeling the Association Vector . . . 43
  - 4.2.3 Algorithm to Generate/Update the Inductive Model . . . 47
  - 4.2.4 Algorithm to Use the Inductive Model . . . 49

4.3 Empirical Study 50
4.3.1 Infrastructure 51
4.3.2 Empirical Method 51
4.3.2.1 Research Questions and Evaluation Strategy 51
4.3.2.2 Independent and Dependent Variables 54
4.3.2.3 Experimental Procedure 54
4.3.2.4 Step 1: Client Configuration Selection 56
4.3.2.5 Step 2: Training Set Selection 56
4.3.2.6 Step 3: Tag Extraction/Abstraction 59
4.3.2.7 Step 4: Defining the Gold Standard 60
4.3.2.8 Steps 5: Tag Classification and Evaluation 61
4.3.3 Threats to Experimental Validity 61
4.3.3.1 Internal Validity 61
4.3.3.2 External Validity 62
4.3.4 Results and Discussion 63
4.4 Summary 68

5 Current Framework Implementation 69
5.1 Current Framework Design 72
5.1.1 Knowledge Base 73
5.1.1.1 Support Criterion Structure 74
5.1.1.2 Knowledge Consolidation 76
5.1.2 `updateKB()` 78
5.1.2.1 Manual Update 79
5.1.2.2 Automated Update 80
5.1.2.3 Information Solicitation 81
5.1.3 `processURL()` 84
5.1.4 `query()` 84
5.1.5 `generateReport()` 85
5.2 Machine Learning Knowledge Base Updates 87
5.2.1 Data Retrieval 88
5.2.2 Web Application Model 90
5.2.3 Learning Strategies 93
5.2.4 Data Storage 95
5.3 Research Questions and Metrics 96
5.3.1 Research Questions 97
5.3.2 Configuration Subject and Data 98
5.3.3 Evaluation Metrics 99
5.3.3.1 Actual Support 100
5.3.3.2 Predicted Support 100
5.3.3.3 Accuracy 101
5.4 Study Design, Results, and Discussion 102
5.4.1 Q1 Study: The effect of web application model, strategy, and training set size on learning accuracy 103
5.4.1.1 Experimental Procedure 103

5.4.1.2 Results 104
5.4.2 Q2 Study: How does the web application model affect analysis costs in terms of tags/rules evaluated and the time needed for analysis? 105
5.4.3 Q3 Study: The effect of training set imbalance on false positives. 105
5.4.4 Q4 Study: The impact of CSS inclusion during the learning process 106
5.4.5 Q5 Study: The impact of Tag Interaction during the learning process 106
5.4.6 Threats to Experimental Validity 107
5.4.6.1 Internal Validity 107
5.4.6.2 External Validity 108

6 Conclusions and Future Work 108

Bibliography 126

List of Tables

4.1 ϕ Values for All Tags in the Example of Figure 4.2. 47

4.2 Configuration Point Details. 52

4.3 Part of the Negative Instance Set of the Initial Web Application Pool. 58

4.4 Evaluation Results. 66

5.1 Contingency table illustrating the four possible states of tag/category co-occurence 95

List of Figures

1.1 When rendered in (a) Internet Explorer 6.0 and (b) Netscape 4.8, both on Windows XP, the Scrabble Home Page is significantly different. 2

1.2 A Web Application Created in Word 97 Executed Differently in Different Client Configurations. 10

2.1 Sample HTML/CSS code and the corresponding web page. 13

3.1 General framework architecture for detecting configuration-specific faults in web applications. 34

4.1 An Example of a Client Configuration Space. 43

4.2 Set of Web Applications Classified as Positive or Negative. 46

4.3 The `updateVector()` Algorithm. 48

4.4 The `queryData()` Algorithm. 50

4.5 False Positive Rate with Respect to Training Set Size. 65

4.6 Examples of Configuration-Specific Errors Found in Our Study. . . . 117

4.7 Mozilla is More Forgiving than Netscape when Tags are Misproperly Placed in Source Documents. 118

5.1 Instantiation of the general framework in the current Approach . . . 118

5.2 A generic representation of the knowledge base. 119

5.3 A practical example of support violation offsets. 119

5.4 Snapshot of the Knowledge Base after a manual update. 119

5.5 Positive and negative web applications in an arbitrary client configuration 120

5.6 Snapshot of the knowledge base after an automated update. 120

5.7 Snapshot of the knowledge base after information solicitation. 121

5.8 The retrieval of data, implemented by `processURL()`, begins once the user submits a URL. From there, the corresponding web page is fetched and, based on the hyperlinks observed, a crawler collects each of the web pages that are a part of the site. Once the source code is retrieved, a vector model of the web application is created. 121

5.9 An overview of `query()` 122

5.10 Visualization of compliance analysis results 122

5.11 The interaction matrix 122

5.12 Accuracy Values Defined 123

5.13 The affect of learning strategy, training set size, and web application model on learning accuracy. The graph shown in (a) corresponds with the L1 learning strategy; (b) corresponds with L2. 123

5.14 The affect of web application model on time needed for analysis and the number of tags/rules analyzed. The graph shown in (a) shows the time needed(b) shows the number of tags analyzed. 124

5.15 The affect of training set imbalance on false positive rate. The graph shown in (a) shows what results with an extra negative example (b) shows the results with an extra positive training example. 125

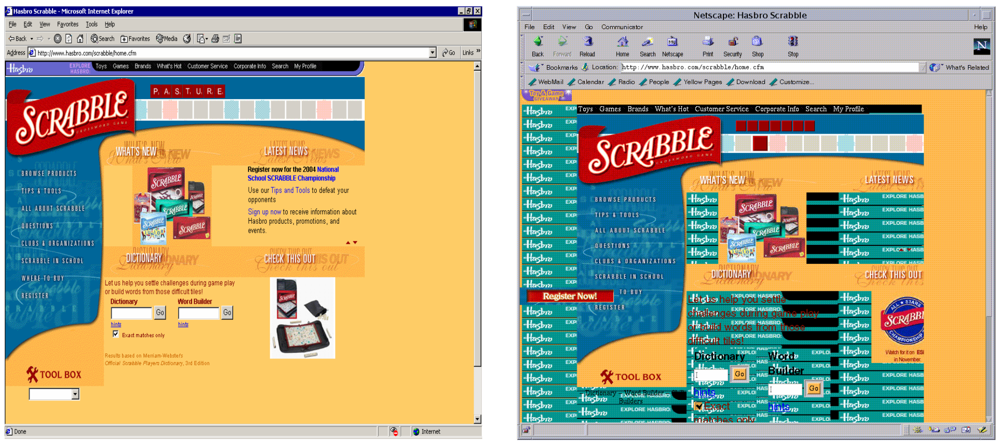
Chapter 1

Introduction

1.1 Motivation

Establishing a high level of confidence in the quality of an implementation is essential in software development. Though the process of detecting and correcting faults in an implemented software system is inherently difficult[21], software quality assurance (QA) becomes increasingly complex when faults only surface in precise configurations. In such cases, the number, nature, and interconnection of constituent parts [45] that define the configuration can significantly impact software quality. To adequately reduce the number of faults in the delivered product, developers must evaluate the overall correctness of the implementation in addition to how that correctness is affected by variation in configurations.

The problem of detecting configuration faults has a trivial solution if the space (or set) of target configurations is manageably small; namely, evaluating the implementation in every possible configuration. Yet, as the size and variability of the configuration space grows, developers are faced with a fundamental QA trade-off between comprehensive configuration space coverage and limitations in development resources [51]. Access to each prospective configuration or the time necessary to apply an exhaustive, brute force assessment strategy is highly unlikely under realistic development conditions. Without an effective technique for assessing soft-


Hasbro Scrabble - Microsoft Internet Explorer
Netscape: Hasbro Scrabble
SCRABBLE
WHAT'S NEW
LATEST NEWS
DICTIONARY
CHECK THIS OUT
TOOL BOX

(a) (b)

Figure 1.1: When rendered in (a) Internet Explorer 6.0 and (b) Netscape 4.8, both on Windows XP, the Scrabble Home Page is significantly different.

ware portability across the configuration space, quality could degrade as software is ported and faults have the potential to remain latent until they are encountered by users in the field. As a result, correcting configuration faults is a crucial step in establishing portability for a highly varied configuration space.

While configuration faults affect portability for a wide range of software types, they are a particular challenge in web application development. Defined as software accessed via a web browser over a network [50], web applications have become one of the most widely used class of software to date and critical components of the global information infrastructure [18]. Given that there are several different browsers (*e.g.*, Microsoft Internet Explorer (IE), Netscape, AOL Browser, Opera, Mozilla, Safari for Mac OS X, Konqueror for Linux, Amaya, Lynx, Camino, Java-based browsers, WebTV), each with different versions (*e.g.*, IE 4.0, IE 5.0, IE 6.0, Netscape 4.0), a number of operating systems on which to run them (*e.g.*, Windows, Power Macintosh), and dozens of settings (*e.g.*, browser view, security options, script enabling/disabling) client configurations used to launch and interact with web applications are highly varied. Though expanded variation and flexibility in web access options allows for more customized web user experiences, subsequent differences in configurations present a serious challenge for web developers to ensure universal quality. Characterized as the *software configuration explosion problem* [34], this high degree of flexibility translates into a wide space of potential web client configurations and complicates the QA effort by requiring that web developers not only ensure that the systems they have developed are correct, but that correctness persists as software is ported. Failure to evaluate web application portability across the configuration space can result in instances where a web page renders correctly in some client configurations and incorrectly in others (Figure 1.1).

In practice, one of the more popular approaches to web application portability analysis involves a qualitative comparison between expected and actual execution. The idea behind this technique is to identify a subspace of popular client configurations and to launch the web application in each. While developers using this strategy get first-hand exposure to configuration faults, this approach is weakened by *limited scope* (because analysis focuses on a small number of target client environments) and *non-diagnostic results* (because only the occurrence of an error, not the cause of the error, is detected). In an effort to address the challenges of web configuration fault detection and the weaknesses of existing web portability analysis approaches, the goal of this research is to enable automated detection and diagnosis of web configuration faults across a large configuration space in a manner that is comprehensive, yet efficient. The basic idea behind this approach is that source code fragments (*i.e.*, Hypertext Markup Language (HTML) tags and Cascading Style Sheet (CSS) rules) embedded in web application source code adversely impact portability of web applications when they are unsupported in target client configurations; without proper support, the source code is either processed incorrectly or ignored, resulting in configuration faults. Using static analysis, configuration fault detection is performed by applying a model of the web application source against knowledge of support criteria; any unsupported source code detected is considered an index to potential configuration faults. In the effort to fully exploit this approach, improve practicality, and maximize fault detection efficiency, manual and automated approaches to acquisition of source code support knowledge have been implemented, variations of web application and client support knowledge models have been investigated, and visualization of configuration fault detection results has been explored. To optimize the automated acquisition of support knowledge, alternate machine learning strategies have been empirically investigated and provisions for capturing tag interaction have been integrated into the process. In the immediate sections that follow, this chapter continues with an overview of the research approach, insight into design considerations for practical implementation, a discussion of research contributions, and finally, closes with an outline of the dissertation structure.

1.2 Research Approach

In web application development, HTML tags and CSS rules are the core languages used. As building blocks of web applications, HTML and CSS directives indicate how an application should be rendered and how users should be able to interact with various web application widgets. When web applications are launched, browsers parse the source code and use it as a basis for rendering and functionality. The ability of a configuration to process these statements correctly provides a critical link between what the web application should be able to do, as outlined in source code, and what it actually does once it has been deployed; a client configuration capable of processing a given tag/rule properly is said to *support* it. Asymmetric support for source code across the configuration space greatly complicates development of web applications that are portable. Given this concept of asymmetry and the perspective that the functional and aesthetic properties of web applications are a function of the underlying source code, it is very difficult for web developers to know which configurations will support their specification, embodied by the source code elements, and which ones will not. In light of these factors, the problem of evaluating the portability of web applications across varied configurations can effectively be recast as identifying known patterns of unsupported source code; this idea lies at the base of the web portability analysis approach utilized in this research.

In the example shown in Figure 1.1 for instance, the tag `<div style = background-image(hasbro.jpg)>` is not supported in client configurations in which Netscape 4.8 is the browser. Because the tag was not supported, it was processed improperly and the image was erroneously repeated throughout; the result was a confounded web page display and diminished usability. The approach used in this work considers <`div style = background-image(hasbro.jpg)`> to be an index to configuration faults in client environments that have Netscape 4.8 as the browser and uses static analysis to detect similar issues in source code inclusion.

1.3 Framework Design Considerations

To adequately assess web application portability, an ideal approach would achieve a high level of configuration coverage in an efficient manner, accurately detect configuration faults, and effectively present results to enable quick correction of discovered faults. Since the research approach uses knowledge of source code support in various configurations as a basis for analysis, it is also important to thoroughly and accurately represent source code support knowledge and have the proper analysis techniques to exploit this knowledge [25]. In response to these requirements, the diagnostic knowledge-based framework developed is automated, employs static analysis, and maintains an extensible model of source code support. Being model- and static analysis-based, the framework uses web application source and a model of code support in varied configurations to facilitate analysis; these factors enable prediction of behavior/rendering faults without executing the WA in a given configuration (thus eliminating the need for direct physical access to target configurations during quality analysis) and, ultimately, provide the basis for efficient coverage of the configuration space. By automating the analysis process, the effort of the *Portability Evaluator*[1] has been reduced to merely submitting a Uniform Resource Locator (URL), or web address, to initiate analysis; automation is especially important in web application development because of short development cycles. The extensible nature of the source code support model allows portability assessments for newer configurations as they are developed and evolved; this is an important attribute in web portability analysis given that client configuration options continually expand as new browsers are developed and newer versions are released. The diagnostic capability of the framework addresses the issue of detecting fault causes quickly by explicitly isolating unsupported source code fragments. Aptly addresses, these solution requirements contribute to a practical, efficient framework that support the goals of this work and provide the basis for continuing research. These factors combined help to define a knowledge-based system that leverages a static, model-based approach to discovering unsupported web application source code in an implementation and gains knowledge of support through varied means.

1.4 Challenges in Attaining and Applying Source Support Knowledge

Given the fundamental attributes of the research approach outlined in this thesis, detection of configuration faults will only be as thorough as the knowledge of tag/rule support criteria across the configuration space. Several challenges to

[1]The *Portability Evaluator* is the person on a web application development team responsible for conducting portability analysis. In some instances, the developer and the portability evaluator may be one in the same. The distinction was made here for clarity.

gaining knowledge of tag support rules threaten the ability to acquire accurate, comprehensive compliance information. Firstly, relying on browser documentation is problematic largely because it can be inaccurate and incomplete. Determining the tags/rules that are accurately accounted for is cumbersome. To ensure that source code fragments are actually supported, a page containing the tag and a description of how the tag should behave or render would need to be launched in the corresponding environment and, like the execution-based strategy, a comparison between the actual and expected effects would be necessary. Much like the execution-based approach, a major problem here is the conflict between the need to evaluate support for the code in each target configuration and the constraints imposed by limited time and limited access to client configurations.

A second, more independent source of information comes from websites that list tag support information [2]. These sources generally feature only a subset of potential environments and a subset of HTML/CSS directives; furthermore, the support rules featured are not guaranteed to be accurate. These factors combined directly conflict with the need for complete, comprehensive support data.

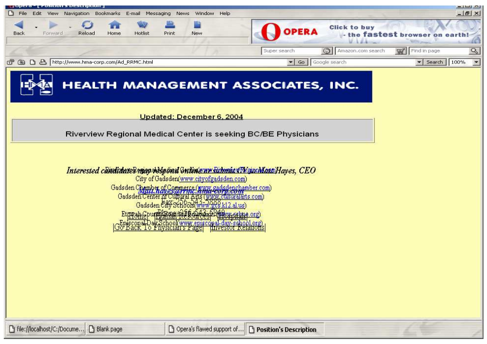
An initial approach to the problem of asymmetric tag/element support was the introduction of coding standards by the World Wide Web Consortium (W3C). In theory, browsers are supposed to follow W3C standards. Yet, even when an attempt is made to fully comply with a given standard, separate implementations of the same standard could differ somewhat in how the HTML/CSS directive is handled causing a resulting difference in functionality or presentation [37]. Furthermore, while some

[2]The Advanced HTML Reference is an example (`http://www.blooberry.com/indexdot/html/`)

browsers claim to be standards compliant, there is evidence that most of them are not, i.e., some tags deemed standard by the W3C remain unsupported or are supported improperly [12]. In short, using standards will often mean that web pages will be accessible by more users, though developers still have to do some work to ensure web application portability[37]. Subsequently, a standards-oriented solution to evaluating tag support is inadequate.

Even with complete, accurate knowledge of environment-specific unsupported HTML/CSS, the use of this information would remain an issue. The main challenges here lie in the qualifications of web application developers and basic human ability.

First, there is an expansive set of HTML tags/CSS elements that web developers could use to create their pages; recognizing the support available for each in the large space of possible client environment configurations is far from intuitive. Although traditional software developers had to be relatively familiar with a language before being confident enough to create and distribute products publicly, a growing number of authoring tools providing a What You See Is What You Get (WYSIWYG) environment that allows developers to create web pages without being familiar with HTML. Users can create a document in Microsoft Word, as an example, and save the document into HTML form [43]; the corresponding HTML is then automatically generated. Yet, the results, can be highly illegible for users depending upon the client configuration used to launch the web application. Given the influence of tag support on end-user accessibility, a deficient grasp of the specific HTML elements incorporated in a web page and a lack of knowledge of how those elements function in various web browsing environments can have a negative effect


File Edit View Navigation Bookmarks E-mail Messaging News Window Help
Back Forward Reload Home Hotlist Print New
OPERA Click to buy - the fastest browser on earth!
Super search
Amazon.com search
Find in page
Go
Google search
Search
100%
file://localhost/C:/Docume...
Blank page
Opera's flawed support of...
Position's Description

http://www.hma-corp.com/Ad_RRMC.html

HEALTH MANAGEMENT ASSOCIATES, INC.

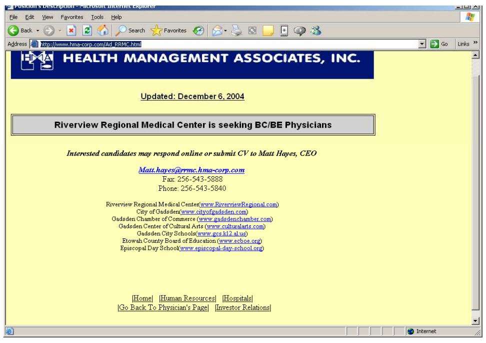
Updated: December 6, 2004

Riverview Regional Medical Center is seeking BC/BE Physicians

Interested candidates may respond online or submit CV to Matt Hayes, CEO

City of Gadsden(www.cityofgadsden.com)
Gadsden Chamber of Commerce (www.gadsdenchamber.com)
Gadsden Center of Cultural Arts (www.culturalarts.com)
Gadsden City Schools(www.gcs.k12.al.us)
Etowah County Board of Education (www.ecboe.org)
Episcopal Day School(www.episcopal-day-school.org)
|Go Back To Physician's Page| |Investor Relations|

Opera 6.0



http://www.hma-corp.com/Ad_RRMC.html

HEALTH MANAGEMENT ASSOCIATES, INC.

Updated: December 6, 2004

Riverview Regional Medical Center is seeking BC/BE Physicians

Interested candidates may respond online or submit CV to Matt Hayes, CEO

Matt.hayes@rrmc.hma-corp.com
Fax: 256-543-5888
Phone: 256-543-5840

Riverview Regional Medical Center(www.RiverviewRegional.com)
City of Gadsden(www.cityofgadsden.com)
Gadsden Chamber of Commerce (www.gadsdenchamber.com)
Gadsden Center of Cultural Arts (www.culturalarts.com)
Gadsden City Schools(www.gcs.k12.al.us)
Etowah County Board of Education (www.ecboe.org)
Episcopal Day School(www.episcopal-day-school.org)

|Home| |Human Resources| |Hospitals|
|Go Back To Physician's Page| |Investor Relations|

Internet Explorer 6.0

`http://www.hma-corp.com/Ad_RRMC.html`

Figure 1.2: A Web Application Created in Word 97 Executed Differently in Different Client Configurations.

on WA portability (Figure 1.2). Experienced developers would need an efficient way of ensuring that the tags incorporated in WA source is supported across varied environments.

Additionally, tag interaction can be a significant factor in web page accessibility as well. A strategy that merely focuses on the occurrence of unsupported tags in source HTML may suffer from an illusion of false positives in instances where an unsupported tag is recognized yet no consideration is given to the complementary tags that provide back-up in unsupportive environments.

1.5 Thesis Contributions

The primary research contribution of this dissertation is a web portability analysis framework with supporting models and algorithms for efficient analysis across a vast configuration space. The specific focus of this work is to automate detection and diagnosis of web configuration faults across a wide configuration space. The key idea of the approach is to use knowledge of HTML tags and CSS rules as the basis of analysis and to integrate automated and manual means for accumulating this knowledge. The research presented in this thesis overcomes the limitations of existing tools and techniques by making the following contributions:

- A framework that utilizes models of web applications and client configuration to detect, diagnose, and support correction of configuration faults;
- A formal, inductively generated model of web client configurations;
- A model of web applications that adequately supports knowledge acquisition of unsupported HTML/CSS directives in varied environments;
- Results of experiments that demonstrate the impact of web configuration faults in practice and feasibility of applying an automated approach to support knowledge acquisition, and
- A basis for accumulating a comprehensive, accurate source of HTML/CSS configuration support criteria.

1.6 Dissertation Structure

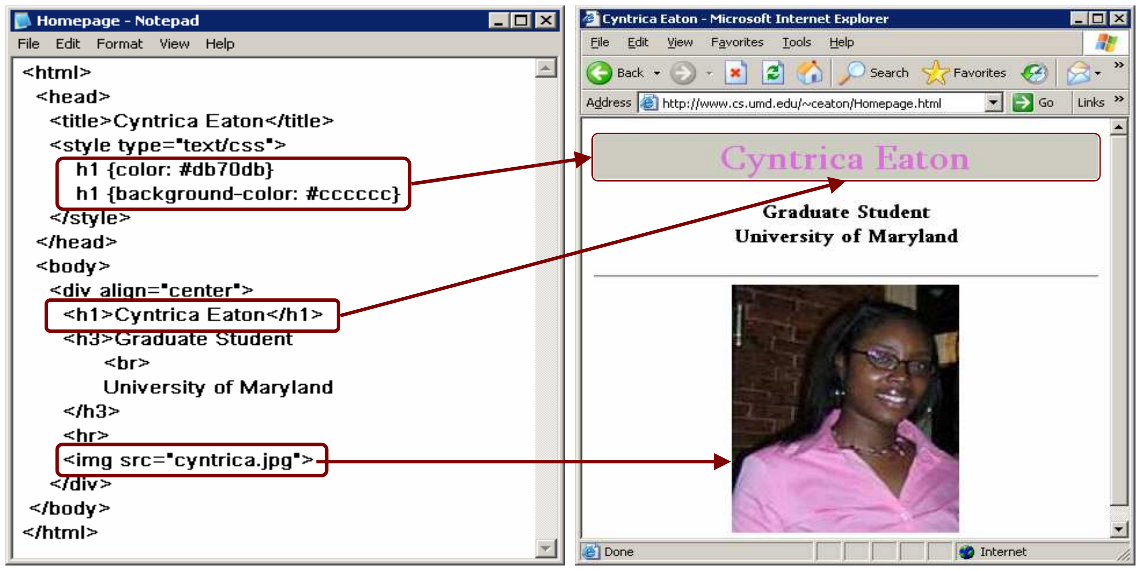
In the effort to present the main ideas of this work, survey the state of the art, and provide a more detailed discussion of research contributions, this dissertation has been divided into six major sections and is organized as follows: Chapter 2 provides an overview of background information and related work. Chapter 3 introduces a general framework for web portability analysis and provides a characterization of current techniques (discussed in Chapter 2) in terms of a general framework. Chapter 4 reviews the initial framework implementation along with an overview of the models and metrics used and their evaluation. Chapter 5 discusses the current implementation in the context of the general framework along with the evaluation of varied automated acquisition techniques. Chapter 6 concludes with a summary of lessons learned and a discussion of future work.

Chapter 2

Background and Related Work

2.1 Web Applications and the Browser Wars

The World Wide Web Consortium (W3C), the main international standards organization [49], lists accessibility as the first of its long term-objectives for the web. Driven by an aim to make the full potential of the web available to all, the W3C's push for universal access primarily focuses on the development and implementation of technologies that account for vast differences in culture, languages, education, ability, material resources, access devices, and physical limitations of users on all continents (`www.w3.org/Consortium`). The work presented in this thesis contributes to this effort because it focuses on differences in access devices (*e.g.*, client configurations), and how they must be accounted for during web development to ensure universal access for web users. Recall, the portability analysis approach implemented in this work uses knowledge of source code support in varied client configurations to predict faulty behavior/rendering of web applications. The sections that follow provide more insight into the core source code languages used in web development, namely Hypertext Markup Language (HTML) and Cascading Style Sheets (CSS); briefly reviews why the aftermath of the *Browser Wars* make knowledge of source code support a valid basis for discovery and diagnosis of web configuration faults; and formalizes notions of tag/rule support and the need for complete knowledge bases during analysis.


Homepage - Notepad
File Edit Format View Help
Cyntrica Eaton - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://www.cs.umd.edu/~ceaton/Homepage.html
Go Links
Cyntrica Eaton
Graduate Student
University of Maryland
Done
Internet

```
<html>
  <head>
    <title>Cyntrica Eaton</title>
    <style type="text/css">
      h1 {color: #db70db}
      h1 {background-color: #cccccc}
    </style>
  </head>
  <body>
    <div align="center">
      <h1>Cyntrica Eaton</h1>
      <h3>Graduate Student
          <br>
          University of Maryland
      </h3>
      <hr>
      <img src="cyntrica.jpg">
    </div>
  </body>
</html>
```

Figure 2.1: Sample HTML/CSS code and the corresponding web page.

2.1.1 HTML

Originally developed as a simple, primitive language for information exchange on virtually any platform [27], **HTML** has evolved into an elaborate, varied basis for developing engaging web applications. In general, HTML tags are used to modify the appearance of text, link widgets to scripts, incorporate objects, and define the internal structure of web documents. Tags are distinguished in web application source code by opening and closing angled brackets (*i.e.*, < and >). In Figure 2.1, for example, `<h1>` is an HTML tag placed at the beginning and end of the text `Cyntrica` and thus modifies how it renders when the web application is deployed. For even greater control over the effects of a tag, web developers can specify attributes and attribute values. In the example shown (Figure 2.1), the <img> tag is modified by the attribute `src` and further defined by the attribute value `cyntrica.jpg`; combined this tag, attribute, and attribute value indicate the relative placement of the image in the web application and the file location of the bit-mapped image to be displayed.

Given the examples presented above, it is rather straightforward that HTML tags are a class of building blocks of web applications. More specifically, they provide directives that indicate how an application should be executed, where page objects should be placed, and how users should be able to interact with various widgets. Subsequently, when support for a given tag is non-existent or insufficient in a given client configuration, the associated directive is improperly processed, and faults have the potential to surface in the corresponding configuration. As a result, HTML tags are important correctness predictors during portability analysis.

2.1.2 CSS

CSS notation was originally introduced by the W3C to promote a cleaner separation between document structure and appearance than HTML and provide developers greater control over web page appearance [43, 1]. CSS rules are distinguished in source code by opening and closing braces (*i.e.*, { and }) and consist of selectors, rules, and values. Selectors specify the HTML tag(s) to which the rule applies, and declarations specify the stylistic effect. The declaration is a set of rule/value pairs. Broadly speaking, rules either specify relative position or display attributes. In Figure 2.1 for example, `h1` is the selector in both of the circled CSS rules. In the first instance, `color` is the rule and `#db70db` is the value. This rule indicates that any text within tag <`h1`> should be the `color #db70db = pink`. Much like HTML, CSS are effective configuration fault indices because, if they are unrecognized in a given environment, the corresponding effect will not render in the corresponding web page triggering a configuration fault.

2.1.3 The Browser Wars

The problem of asymmetric tag/rule support and, ultimately, configuration faults, is largely a residual effect of the 90's era *Browser Wars*. During that time, the popularity of the web was increasing quite rapidly; several browser developers (with the major players being Internet Explorer and Netscape Navigator) incorporated extra, proprietary HTML tags that rendered improperly in other environments. Prior to these extensions, the original version of HTML only defined primitive layouts. Responding to the demand for a richer selection of markup elements (*e.g.*, nested lists, images), browser developers sought to gain competitive advantage by supporting advanced functionality. To introduce more sophisticated elements to designers and enable them to create pages with more complex designs, browsers were deliberately developed with features available in one and not the other. The end result for users, as we experience today, is variability in HTML/CSS support that surfaces in improperly rendered media, incorrect display of formatting, forms that are not seamlessly linked to their scripts, and other faulty behavior and appearance.

2.1.4 Definitions

Lack of universal support for source code elements and an inability to process source code properly in particular client configurations are a direct result of the Browser Wars and the underlying basis for configuration faults in web applications. Having introduced HTML tags, CSS rules, and the concept of the Browser Wars, this section formalizes the concept of tag/rule support and makes the case for the importance of thorough support criteria knowledge.

Definition 1: HTML and CSS Tag Support

When support for a given source code fragment (tag/rule) is known to be non-existent or insufficient, the associated code is improperly processed and faults potentially surface for users in the associated configuration. Let C denote the universal set of client configurations. Consider T to collectively be the universal set of all possible HTML document source tags and CSS rules. Consider that:[1]

$$\forall c_j \in C\ \exists I = \{i_1, i_2, ..., i_{|I|}\}\ s.t.\ (I \subseteq T) \wedge \neg supports(c_j, I)$$

That is, each client configuration supports only a portion of the universal tag/rule

[1]In the definition that follows, the term *support*(x, y) indicates that x supports y.

space; for all possible client configurations, there is a set of HTML tags and CSS rules that are unsupported in the given environment. Consequently, the tags/rules in I could be considered indices to configuration faults when they are included in the source code of a web page w is launched in c.

Example 1:

The tag <`marquee`>, though a part of the universal tag set T, was implemented by Internet Explorer developers and is not properly processed in client configurations that feature early versions of Netscape as the browser. As a result, the corresponding client configurations will not process the tag properly and the intended functionality will be lost.

Definition 2: Necessity for Complete Definitions of I

Adequate detection of unsupported tags in I is largely dependent upon the accuracy and completeness of its description. Detection of faulty program properties will only be as strong as the knowledge of such properties. Consider the following:

$$\forall t_i \in T \ s.t. \ t_i \notin I \wedge \neg supports(c_j, t_i)$$

This essentially states that, the definition of I must be complete to ensure accurate analysis results. If, for instance, a tag, t_i is actually unsupported in c_j but it is not included in the description of I, the resulting static analysis will be flawed. If knowledge base is incomplete, only a subset of bugs is recognized. Although a page containing t_i should be flagged as possibly having a configuration fault, the analysis will not return the proper result. Static analysis provides an adequate foundation for configuration fault detection and diagnosis, but it can be weakened by the quality and completeness of support criteria used during analysis.

Example 2:

Consider a tag, <`blink`>, that is unsupported when the browser in the client configuration is Internet Explorer. If <`blink`> was included in the source code and the tag was not included in I during analysis, the accuracy of the resulting report would be compromised. Developers would be subject to latent failures and false confidence in web page, and ultimately web application portability.

Definition 3: Tag Interaction

As mentioned in the previous definition, completeness of bug patterns is important for evaluation. Tag interaction, specifically support violation offset, is another phenomenon that must be modeled in order to get accurate results. Consider that:

$$\forall (t_a \wedge t_b) \in T s.t. supports(c_j, (t_a \wedge t_b)) \wedge \neg supports(c_j, t_a)$$

This means that the tag t_a is not supported in configuration c_j and thus, its inclusion in source code could have negative effects. Yet, a web page that contains both t_a and t_b behaves/renders properly upon deployment. In this context, t_b offsets the lack of support for t_c in c_j.

It is important to account for support violation offsets during portability analysis; a strategy that merely looks for unsupported tags in source HTML produces an illusion of false positives[2] when no consideration is given to tag offsets.

Example 3:

A common example of support violation offset is the Javascript versioning tag. More specifically, Internet Explorer 4.0 does not recognize Javascript 1.3. However,if a web page had both <`script language=javascript1.2`> and <`script language=javascript1.3`>, the page would work properly because the latter could be offset by the prior.

2.2 Related Work

The goal of the work presented in this thesis is to effectively detect configuration faults and ensure a consistent level of quality for users as they launch and interact with web applications. This section outlines related work that addresses portability analysis (for the web and in general), web quality analysis, and fault localization. An overview of machine learning applications in software engineering concludes this section.[3]

[2]In this context, a false positive is a web page that passes analysis but contains tags that are unsupported in a given configuration.

[3]This area of research is related to this dissertation because machine learning tactics are employed to assist in the correct, thorough definition of I and avoid the problems formalized in Definitions 2 and 3.

2.2.1 Web Portability Analysis

QA strategies for evaluating web portability include launching web applications in varied configurations [3, 4, 8, 9, 54], looking for unsupported HTML in source code [6, 16], and attempting to transform code into a form that is universally supported [11]. This section outlines existing approaches along with their limitations and tools that implement them.

2.2.1.1 Manual and Automated Execution-based Approach

Execution-based approaches to web portability analysis primarily involve launching web applications in target configurations and verifying correctness based on the qualitative comparison between expected and observed results. In the brute-force application of this approach, web application deployment and analysis are both carried out manually. One factor that makes this approach particularly problematic is the requirement that web applications be physically loaded in order to preform quality assurance. Though exhaustive coverage of the configuration space would allow thorough portability analysis, physical access to each possible class of environments is extremely difficult; as a result, there is a notable conflict between the need to test each potential client configuration and the constraints imposed by limited development resources. Even with access to each possible configuration, the time and effort required to effectively asses web pages using this strategy can also impede the depth of the web application evaluated. Because this strategy can be weakened by client configuration availability and limited time, this technique is highly ineffective and impractical for web developers interested in establishing portability across a vast, richly defined configuration space.

While the brute-force strategy evaluates web application portability post-implementation, Berghel presented a manual execution-based approach [3, 4] designed for pre-implementation use. The basic idea outlined in [3, 4] is to launch a suite of test web pages, called *Web Test Patterns*, and observe the results to become familiar with HTML support criteria in varied configurations. Each web test pattern in the suite incorporates several HTML tags and descriptions of the impact they would have if processed correctly. This approach allows web developers to derive a cognitive model of HTML support criteria across various configurations. As developers synthesize web application code during development, the idea is that they will use this model to drive decisions regarding which HTML tags to include in the implementation. Much like the brute-force strategy, the effectiveness of Web Test Patterns is mainly restricted by resource limitations. In addition, Berghel's approach only allows users to develop a mental model of tag support criteria; effective application of this model can be severely flawed in practice given the expansive set of HTML tags that can be included in source code and the intricacy of support criteria. Retaining this information and attempting to use this strategy effectively is clearly time-, cognition-, and resource-intensive.

To minimize the effort and, ultimately, the cost of analysis using execution-based approaches, researchers have proposed reducing the space of test configurations through combinatorial testing approaches. In particular, Xu et al. [54] propose applying single- factor and pair-wise coverage criteria to systematically reduce the space of distinct configurations evaluated during quality assurance. This process applies sampling heuristics to select the minimal set of client configurations that must be assessed to establish confidence in the entire configuration space. While this approach can make subsequent analysis more cost-effective in terms of resources and effort, it can also create false confidence in analysis results when the set of test configurations does not accurately represent the entire space.

Commercial tools designed to make execution-based approaches more cost effective mainly focus on automating the launch of web applications in varied configurations to mitigate the necessity for in-house access to configurations during quality assurance [8, 9]. Such tools accomplish this goal by launching a web application on the behalf of web developers in a subset of configurations and capturing a screenshot of the rendered result; the image is then returned to the developer who analyzes web application correctness by manually examining the screen-shots and relying on visual cues (*i.e.*, misrendered pages) to discover errors. If visual cues signal an error, the developer must employ additional methods, such as manually examining the application code, to identify fault causes. Since the result of this analysis only provides visual evidence of an error and no indication as to why the error occurred it is *non-diagnostic.* In addition, this approach only detects rendered faults, or faults that are evident based on visual inspection; detection of behavioral faults is infeasible since a single snapshot cannot capture such defects. Consider for instance that some browsers allow quick access to page elements through the use of *hot keys* or key sequences that activate particular widgets in an interface. Though configurations that feature Opera as the browser do not support this functionality, this lack of support is not conveyed in the screen shot.

Problems with this approach include a non-diagnostic presentation of results since users would only have evidence of the fault from visual cues; identifying the factors that contribute to the anomalous behavior requires more work and effort. In addition, usefulness of fault detection results are generally constrained by the small set of client configurations used during analysis and the dimensions of the screen capture. In other words, this approach only detects rendered faults, or faults that are evident solely based on visual inspection. Faults triggered by user action or those that fall out of the range of the screenshot will remain undetected. Subsequently, this approach is only relevant to portability analysis when aesthetic, visually-detectable faults are of interest. Consider, for instance, that some browsers allow quick access to page elements through the use of *hot keys* or key sequences which activate access. None of the Opera browser versions support this feature, yet, launching web applications that have this feature in Opera and capturing a screenshot does not provide insight into the existence of such faults.

In general, execution-based approaches are deficient because of limited configuration coverage, lack of diagnostic ability, limited applicability of results or some combination of these. As a result, practical implementation of execution-based strategies generally involves configuration sampling. Such issues give rise to an incomplete, resource-intensive analysis of the web application that does not provide an adequate basis for establishing confidence in web application portability. The approach outlined in this thesis by-passes execution based analysis altogether and applies a static analysis approach, the crux of which facilitates more efficient analysis alone. The static, model-based analysis applied reduces the need for configuration access and simultaneously provides a more fertile basis for fault diagnosis. The model-based aspect of this work enables each configuration to be represented during analysis reducing the threat of inaccurate equivalence assumptions.

2.2.1.2 Lookup-based Approach

Look-up based approaches, like Doctor HTML [16] and Bobby [6], detect configuration faults by maintaining an account of unsupported HTML tags in a predefined subset of web configurations and essentially *looking* for them in source code. Results of analysis are returned as a list of the unsupported tags found in a given web application source code and the configurations with support violations.

One problem of this approach is captured nicely by Figure 1.1. In this example, quality is clearly diminished for Netscape users of the Scrabble website, however, Doctor HTML did not include this particular support violation in the analysis report. This factor drives home the point that analysis will only be as thorough as the knowledge of configuration support criteria. In instances when the incomplete or inaccurate support criteria is used, configuration faults will continue to remain latent after analysis. Since the tool approach is proprietary, it is unclear whether this oversight resulted from a lack of CSS rule analysis or, despite inclusion of CSS rule support knowledge, the corresponding criteria was simply missing from the checklist. In either the case, the work presented in this thesis improves upon the *lookup-based approach* by using a more inclusive model during analysis, integrating diverse knowledge acquisition strategies to improve analysis accuracy, and incorporating an extensible knowledge base. In terms of the more inclusive model, CSS rule support knowledge has been incorporated; support criteria data is accumulated from diverse sources in the effort to build an accurate, thorough support knowledge; and the extensible knowledge base model allows support criteria to continually evolve.

2.2.1.3 Source Code Standardization Approach

Though Chen and Shen [11] do not precisely focus on web configuration fault detection, correcting web portability threats is a key aspect of their work and is highly applicable to the domain of web portability analysis. In their research, Chen and Shen base their approach on the assumption that web source code standards, as defined by the W3C [48], provide the most effective basis for developing web applications that are portable. The crux of their technique is to transform the source code of a web application into a standardized form in which all non-standard code fragments are eliminated from the code and the appearance of an original implementation is preserved. One problem with this approach stems from the fact that, as noted by Phillips [37], even if browsers fully comply with published standards, the code may still be processed differently since standards do not address every detail of implementation; in addition, there are instances in which browsers claim to be standards-compliant yet some tags deemed standard by the W3C are unsupported or supported improperly [12]. In some instances, web developers only get acquainted with the parts of the standards that work in most browsers through experience [37];

subsequently, developers may still have to employ a variant of the execution-based approach to assess source code support in client configurations. In contrast, one goal of the work in this thesis is to derive detailed knowledge of source code support to facilitate analysis as opposed to relying on standards that may or may not be fully incorporated across the configuration space.

2.2.2 Web Testing

The research presented in this dissertation is one facet of a general endeavor to support and improve the quality of user experiences on the web. Given increased interest in the quality and reliability of web applications, several researchers have proposed and developed web QA techniques. The majority of the research effort found in the literature has been concentrated in applying traditional QA measures to web-based software. In particular, several tools and techniques have been developed to assess the functionality and performance of web applications including general frameworks [53, 42, 40, 52] and test case generation strategies [2], to the application of traditional white-box testing techniques [47, 39], object-oriented based strategies [28], and statistical testing approaches [24]. Although the pursuit of web quality is a unifying factor between the cited work and the research presented in this thesis, none of the fore mentioned approaches concentrates on how to detect configuration faults before web applications are fielded. In terms of the thesis approach, the main concern is the challenges presented when software configurations are untested and the environment does not support HTML/CSS directives.

2.2.3 Portability

Software portability research considers how software correctness can be compromised depending on the configuration used to deploy it. Work addressing software portability is relatively scarce in recent literature though the problem of detecting configuration faults with limited resources continues to loom. In one example of work in this area, Mooney [35] cites portability as a desirable attribute for a wide majority of software products and proposes guidelines for ensuring probability in each phase of software development. Bishop [5] discusses how portability affects graphical user interface (GUI) design and proposes an approach that uses XML to encode software specifications and an engine capable of generating environment-specific event handlers based on the XML. Koltashev [26] addresses the issue of portability in a mission critical context - communication satellites. In that domain, the satellites are functionally equivalent, yet the computing hardware varies significantly between them; this essentially provides the basis for a need to port on-board software to various computing platforms. Koltashev [26] proposes architectural stratification and interface standardization as a means of ensuring portability for satellite software. Cohen et. al [13] look at the effects of the configuration used during testing on code and fault coverage noting that for individual test cases and certain types of faults, configuration matters.

In addressing the difficulty of setting up and maintaining large machine test banks during portability analysis, a commercial tool, IBM's VMWare [23], was designed to mitigate the effects of limited physical access to test configurations. The basis of this tool is to simplify configuration management and eliminate the need for access to physical machines to detect configuration faults. This software enables the use of virtual test environments that simulate multiple operating systems and software applications running concurrently in virtual machines on a single Intel-based computer.

The Skoll system, developed by Memon et al. [34], is based on the idea that evaluating software quality under varied usage conditions can be greatly improved by increasing user participation in quality analysis. They accomplish this goal with a global-oriented QA process that assigns users specific test cases to run in their configurations and accumulates results to derive a model for failure prediction. One QA task implemented in Skoll was to determine which specific options caused software failures to manifest; The authors call this process *fault characterization.*

2.2.4 Fault Isolation

In short, fault isolation is an effort to identify source code statements that cause anomalous behavior[41]. This field of research is highly applicable to the work presented in this thesis largely because the endeavor to automated web configuration fault isolation is largely an attempt to isolated unsupported tags/rules using source code analysis. As mentioned prior, tags/rules have the potential to trigger anomalous behavior/appearance when they are unsupported in a given configuration. While this dissertation focuses on faults that are activated in precise configurations, this section surveys a more general collection of fault isolation/localization research.

Several techniques have been proposed to address fault localization in more traditional systems. Li et al. [29], for instance, consider the impact of improperly copy/pasted code on software errors. Hangal et al. [20] present DIDUCE, a tool that isolates the root of errors based on system invariants. Engler et al. [19] look for contradictions in code constructs, provides an alert when contradictions are detected, and allows users an opportunity to determine which construct is incorrect; once a contradiction is identified, a template rule, or bug pattern, is devised to identify other code that may be the root of similar errors. Zeller et al. [56] presents an approach to fault isolation called the delta debugging algorithm. The idea behind Zeller et al.'s research is to isolate the difference between a passing test case and a failing one by simplifying a set of failing test cases to a minimal subset that still produces the failure. Fault-causing attributes of failing test cases are isolated by incrementally eliminating attributes that appear to be irrelevant to the failure.

Another body of work uses violation of *implicit coding rules* as a means of fault isolation. Implicit coding rules have been defined as key, generally undocumented rules that affect the correctness of a software system when violated. Lack of documentation is particularly problematic because developers who initially worked on the project may be aware of them yet, as new developers join projects or legacy programmers forget them, defects are introduced. Matsumura et al. [33] use manually generated bug code patterns to identify violations of *implicit* coding rules. The idea is to have an expert maintainer investigate bug reports, identify the code fragments that caused the bug, and derive a template for a code pattern that can be used to identify similar faults. Li and Zhou [30] address implicit coding rules as well, yet their work incorporates a data mining approach to automatically extract rules and employ a technique to automatically detect subsequent violations.

Sterling and Olsson introduce the concept of program chipping [44], a technique that uses program source code to generate a parse tree, eliminates or changes nodes of the parse tree, and compiles/runs the code associated with the modified tree to identify code features that contribute to faulty behavior. This technique automatically removes of chips away parts of a program so that the part that contributes to some symptomatic output becomes more apparent with each successive run.

Though similar in spirit, the work discussed in this dissertation diverges from this body of related work in key areas. Given a web page that does not work, the conceptual approach is to isolate the cause of the configuration fault by simplifying the code until the unsupported code fragment can be isolated. Yet instead of actively modifying or eliminating code fragments to isolate fault-inducing statements, web pages and whether they are positive or negative examples are used as evidence to the supported/unsupported nature of each source code fragment; a learning approach is applied to reconcile support patterns and determine the contribution of a code fragment to faulty category. Also, instead of using one failing example, several are collected and compared to successful examples in order to isolate a host of unsupported features and update knowledge of support criteria based on the results. The main goal is to generalize support patterns from experience as opposed to reducing individual source code examples piece by piece. Moreover, in the case of web application development, tag support can be considered implicit coding rules; the overall goal of this work is to automate the process of investigating causes and updating support knowledge. Moreover, users are expected to submit web application source as the bug reports; instead of requiring an expert to investigate the cause of the bug by opening the source in the corresponding configuration, that process is automated in this work by applying a learning mechanism to diagnose the root of the problem. We only use experts when they are directly supplying knowledge or gathering more evidence of tag/rule support when necessary.

2.2.5 Machine Learning in Software Fault Detection

One of the earlier goals of this research was to identify an inductive mechanism capable of discovering tag support criteria from empirical data. Several other researchers have applied machine learning solutions to quality assurance tasks. The discussion that follows highlights work that falls within this category.

To briefly motivate the use of machine learning to predict the behavior of fielded software, consider that statistical analysis of measurable program features could be used to automatically extract bug revealing information from source code and other software development artifacts. Specifically, in the work presented by Haran et al. [22], the main idea is to support measurement and analysis of fielded software systems by automatically classifying execution data as a passing or failing instance of software behavior. The authors use statistical learning algorithms to isolate revealing predictive factors (*i.e.*, the number of times a method is called, runtime, input size, etc.) and build classification models that can distinguish passing and failing runs.

In addition, Bowring et al. classify program executions using a technique based on Markov models [7]. Their model considers program branches as a significant quality attribute.[4] Podgurski et. al use automated clustering techniques that group software failure reports and automatically discover which ones are likely to be manifestations of the same error [38]. Brun and Ernst [10] use machine learning approaches to identify dynamic invariants that are likely fault indicators. The approach they present takes a set of program properties for a given program as input and returns a ranked subset of those properties that are more likely than average to indicate errors in the program as output. Their experiments indicate that a machine learning mechanism can identify fault-revealing program properties which result from erroneous code. Brun and Ernst use support vector machines and decision tree learning tools to classify these properties. Cubranic and Murphy [14] use text categorization techniques to assign bug fixes to developers given a plain-text description of the bug submitted by system users.

The work presented by Liblit et al. [31] is very similar to our research in that the goal is to recognize bugs in a system based on user experiences with faulty executions from a large, distributed user community. More specifically, Liblit et al. apply statistical modeling based on feature selection to the problem of fault localization in order to identify program behaviors that are strongly correlated with failure and

[4]Quality attributes are features or attributes inherent in software development that affect quality.

are therefore likely places to look for error. Moreover, the idea is to gather user execution profiles, identify predicates in the source code, and use logistic regression to determine the statements most strongly correlated with system failure. Instead of predicates, the goal of our research is to identify HTML/CSS most strongly correlated with system failure. In addition they use dynamic analysis for discovering the causes of faults; we focus on a static approach.

In this thesis HTML/CSS are the main predictive features and web pages are modeled as either negative (failing) or positive (passing) examples of web application behavior. Much like Bowring et al., the goal of this work is to use static analysis to detect the root cause of bugs experienced during execution. Like Engler et al. another interest is automatically extracting bug-revealing properties from source code in order to dramatically reduce the manual effort needed to check a large, complex systems.

Chapter 3

General Framework Architecture

This section presents a general framework for analyzing web portability across the configuration space and discusses how it aligns with existing tools and techniques. As mentioned in Section 2.2, there are three main classes of web portability analysis approaches: execution-based, look-up based, and source code standardization. The discussion that follows covers the components that define the general framework and provides insight into how each is instantiated for practical approaches to detecting configuration faults for web applications.

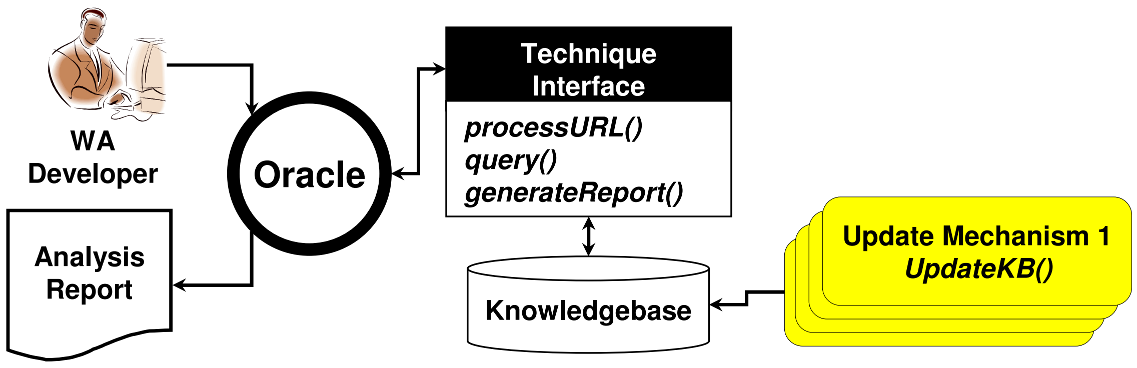


Figure 3.1: General framework architecture for detecting configuration-specific faults in web applications.

3.1 Framework Overview

At a very high level, portability analysis can be conceptualized as a function $\mathcal{F}$ that accepts fault-relevant properties as input and returns an account of configuration faults detected as output. In this discussion of a generic web portability approach, the function $\mathcal{F}$ is called the *oracle*. As shown in Figure 3.1, basic configuration fault detection begins once web applications are submitted to the oracle as input and ends when the oracle returns the *Analysis Report* as output. The interim role and quantity of *knowledge bases* along with the implementation of `processURL()`, `query()`, `generateReport()`, and, `updateKB()` are a direct result of the fault detection strategy used. In general, `processURL()` is responsible for retrieving input and, if necessary, conditioning it for analysis; `query()` implements the analysis strategy;

and `generateReport()` returns analysis results to users. Note that the knowledge base(s) contain(s) data needed to perform analysis and `updateKB()` is used to import more data into the knowledge base(s) when necessary. As shown in Figure 3.1, one or more subcases, or instantiations, of `updateKB()` may co-exist within the framework.

3.2 Manual and Automated Execution-based Approach

Execution-based techniques require that web applications be launched within client configurations in order to detect configuration faults and evaluate the subsequent quality (see Section 2.2.1.1). Though details of the manual and execution based approaches differ, both can be abstracted by the general framework in the following way with slight variation: In both the manual [3, 4, 54] and automated [8, 9] execution-based approaches, `processURL()` respectively returns the deployed website or an image of the deployed website launched in a set of target configurations. The `query()` process, in both approaches, is a comparison between expected and actual observations; in the manual approach, the `query()` process compares the deployed web application with a ground truth model[1] of presentation and behavior; in the automated approach, the image of the deployed web application captured as a result of `processURL()` is compared with the ground truth of presentation. In each case, the `query()` and `generateReport()` processes are manually executed by the evaluator; as the evaluator makes the comparison between actual and ex-

[1]The ground truth is a conceptual model of correct execution/rendering. In most cases, it exists in the mind of the developer, but it can also exist as a mock-up.

pected models (`query()`) they maintain an account of configuration faults detected (`generateReport()`).

Note, for execution-based approaches to web configuration fault detection, the *knowledge base* contains a list of configurations that will be evaluated; what differs, however is the implementation of `updateKB()`. For Berghel's approach and the commercial tools, `updateKB()` simply adds a new configuration to the list when necessary; In Xu et al.'s approach, `updateKB()` adds a new configuration and recalculates the equivalence classes. The work outlined in this thesis primarily improves upon this approach by eliminating the need for access to client configurations during analysis and diagnosing the causes of detected faults.

3.3 HTML Lookup Techniques

In look-up based approaches (see Section 2.2.1.2), `processURL()` fetches the web page source code associated with a submitted Uniform Resource Locator (URL). The `query()` process then compares the list of HTML tags contained in the source code with a list of tags known to be unsupported in various target configurations. The `generateReport()` interface returns results to the user as an account of the unsupported tags found in a given web application source code and the corresponding configurations for which the tag/rule is unsupported. Note, look-up based approaches implemented to date differ from this work by excluding CSS-based configuration fault detection and by, presumably, only supporting an instance of `updateKB()` that accepts support criteria from the tool development team;

the knowledge base, designed as a part of this dissertation, includes knowledge of cross-configuration CSS support and integrates varied strategies as instances of `updateKB()`.

3.4 Source Code Standardization

The source code standardization approach presented by Chen and Shen [11] (see Section 2.2.1.3) instantiates the general framework in the following way: `processURL()` launches the URL associated with a webpage in a browser. The `query()` process then examines the DOM tree of the web application (generated by the browser) to identify the layout of the web page. Nodes in the DOM tree represent segments in the web page and maintain information such as the size, location, and contents of the corresponding segment. Based on the DOM model recovered, `query()` generates a CSS box outline of the web page based on the segments, their size, and their location. Next, the method converts the source code associated with each segment/box to a standardized version. Since this approach is not a fault detection system, `generateReport()` is instantiated slightly differently; instead of returning an account of faults detected, the interface returns a W3C standard version of a web application. The *knowledge base* in this approach contains rules for transforming the code; `updateKB()` accepts rules, presumably, tool developers and incorporates them into the *knowledge base*. The work presented in this thesis improves upon the code standardization approach by developing a detailed understanding of cross-configuration source code support and not relying on a standard definition that may or may not be fully implemented.

Chapter 4

Initial Implementation

This chapter discusses the initial work conducted to demonstrate the feasibility of the research approach and its potential to advance the state-of-the-art in web portability analysis. Initial work included design and development of the following: client configuration and web application models, an inductive learning approach, an algorithm that updates knowledge of configuration-specific tag support criteria, and an algorithm to query the model during portability analysis. In addition to details about each of the aforementioned, this chapter provides an introduction and in-depth analysis of an experiment conducted to evaluate the feasibility of updating support criteria knowledge (through `updateKB()`) through inductive learning. Because this work was conducted at an earlier phase of the project, only HTML tags were considered as possible configuration fault triggers.

4.1 General Framework Instantiation

In both the initial and current phases of this work, web configuration fault analysis is performed the source code level and can be divided into three main tasks: acquisition of code fragment support knowledge, discovery of unsupported source code fragments in web application source code, and presentation of support violations detected during analysis. Recall, the generic framework introduced in the previous chapter and its components; namely *knowledge base(s)* and the `processURL()`, `query()`, `generateReport()`, and, `updateKB()` interfaces. In the context of this work, `updateKB()` acquires knowledge of source code support; `processURL()`, `query()`, and the knowledge base are key in portability analysis; and `generateReport()` is mainly responsible for analysis result presentation.

It is important to note that `query()` implements the portability analysis strategy and uses the output of `processURL()` and knowledge base data to drive configuration fault detection. For more detailed insight, the basic instantiation of `query()` can be formalized as follows: Given the universal set of client configurations, C, and a universal set of web applications, W, each configuration, $c_i \in C$, used to browse and interact with web applications has a set of source code fragments $U_{c_i} = \{u_{(c_i,1)}, u_{(c_i,2)}, ..., u_{(c_i,a)}\}$[1] that are unsupported. Recall, lack of support signifies an inability of a given configuration to process the code properly and link the code with its proper aesthetic/functional properties; subsequently, the intended impact of the code is lost and configuration faults may result. The basic approach defines a web application, $w_j \in W$, as a set of source code fragments $SC_{w_j} = \{sc_{(w_j,1)}, sc_{(w_j,2)}, ..., sc_{(w_j,b)}\}$ where set members include HTML tags (in the initial implementation) and CSS rules (in later work). As shown in Equation 4.1,

$$U_{c_i} \cap SC_{w_j} = \begin{cases} \{f_1, f_2, ..., f_n\}, & \text{if } w_j \text{ contains source code unsupported in } c_i \\ \emptyset, & \text{otherwise} \end{cases} \tag{4.1}$$

[1]In Section 2.1.4 *Definition 1*, U_{c_i} was defined as I

if the intersection of U_{c_i} and SC_{w_j} yields a non-empty result, the source code for w_j contains code fragments, that are unsupported in c_i; code fragments $\{f_1, f_2, ..., f_n\}$ that overlap between the sets are likely to reveal configuration faults. The notation f_m indicates an instance where $sc_{(w_j,t)} = u_{(c_i,v)}$. In the context of this formalized view of the `query()` interface, the goal of our work is to design an effective way to compute the intersection of U_{c_i} and SC_{w_j} for a web application, w_j, with respect to a wide, diverse set of configurations, C.

To gain more detailed insight into the bridge between the general and initial frameworks, consider the following instantiation: each configuration has a dedicated *knowledge base* that contains an account of unsupported HTML tags. To initiate analysis, users submit the URL associated with the home page of a web application to the `processURL()` method. Next, `processURL()` activates a web crawler to gather web pages that comprise the web application, constructs a model of each web page by extracting HTML tags from raw source code, and forwards the resulting models associated with each to the `query()` method. For each configuration represented, `query()` accesses the corresponding knowledge base, retrieves the list of unsupported HTML tags, and compares it with the web application source code to detect matches. If a source code fragment in the web application appeared in the knowledge base of a given configuration, the name of that configuration would be appended to a list of potentially faulty configurations. `generateReport()` returns a flat, text-based list of results indicating the pages with support issues, the unsupported tags they contain, and the configurations that do not support those tags. In this phase of the research, there was only one instance of `updateKB()` and it was defined as a supervised machine learning method. Briefly, `updateKB()` was designed to accept both faulty (negative) and correct (positive) examples of web page behavior in a given configuration, monitor source code inclusion patterns (*e.g.*, the number of times specific HTML tags, attributes, and attribute values were included in positive vs. negative examples) and to derive support criteria knowledge from those observations. Tags highly correlated with faulty labels examples were added to the *knowledge base* associated with each client configuration.

4.2 Inductive Model

A key contribution of this work is the development of client configuration models that encapsulate source code support. Given the approach in this dissertation, the configuration model is used for both accumulation of support criteria knowledge and for use during portability analysis. The initial client configuration model contained two parts: (1) a *graph representation* of the entire client configuration space and (2) an *association vector* for each client configuration. The association vector encapsulated knowledge of how HTML tag inclusion web application failures in specific client configurations. This section provides further details for both parts.

4.2.1 Modeling Client Configurations

In the initial implementation, each client configuration was described in terms of *options* such as operating system installed, browser, browser settings, network speed, geographical location, etc. Each option takes its value from a discrete set of *settings*. For example, the operating system option (called `OS`) in the empirical study outlined in Section 4.3.1 takes values from the set {WinXP, Mac OS X}. A client's mapping from options to settings is called a *configuration* and is represented as a set $\{ (V_1, C_1), (V_2, C_2), \ldots, (V_N, C_N) \}$, where each V_i is an option variable and C_i is its constant value, drawn from the allowable settings of V_i.

In practice not all configurations make sense (*e.g.*, feature $(Browser = WebTV)\&(Versio$ 2.0) is not supported when $(OS = Linux)$). Therefore, the framework will support *inter-option constraints* which limit the allowable settings of one option based on the settings of others. Constraints are represented as follows: $(P_i \rightarrow P_j)$; this means "if predicate P_i evaluates to $TRUE$, then predicate P_j must evaluate to $TRUE$." A predicate P_k can be of the form A, $\neg A$, $A\&B$, $A|B$, or simply $(V_i = C_i)$, where A, B are predicates, V_i is an option and C_i is one of its allowable values. A practical constraint example is: $(Browser = IE)\&(Version = 6.0) \rightarrow (OS = WindowsXP)$; this indicates that the operating system must be Windows if the browser is Internet Explorer. This constraint is attributed to the property that version 6.0 is not available for other operating systems). A *valid configuration* is a configuration that violates no inter-option constraints.

Figure 4.1 shows an example of a client configuration space. Each node represents a valid configuration. Edges connect two nodes that differ by exactly *one* option setting. For example, nodes 1 and 2 differ by one option setting (`OS=Linux` vs. `OS=WinXP`); similarly, nodes 1 and 3 differ in one option setting (`Browser=Netscape` vs. `Browser=Internet Explorer`). Nodes 2 and 3 are not connected since they differ by more than one option setting. These edges are used to traverse the client configuration space (in the algorithm discussed in Section 4.2.3). Without loss of generality, the client configuration space is assumed to be connected (*i.e.*, it is one connected graph; "dummy" nodes are used to connect disjoint parts).

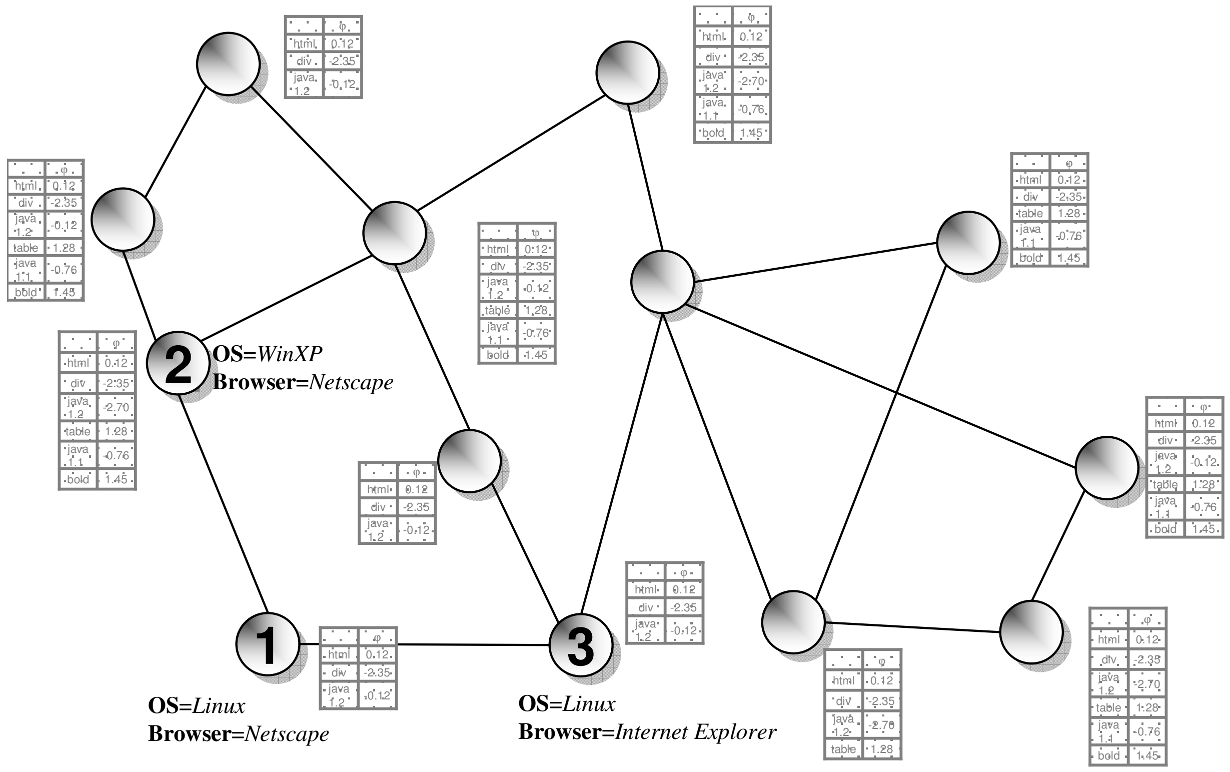


Figure 4.1: An Example of a Client Configuration Space.

4.2.2 Modeling the Association Vector

Each point in the configuration space is mapped to an association vector. Intuitively, the association vector encodes the likelihood that a given tag is associated with incorrect execution. For example, results of the empirical study (Section 4.3) indicated that the <`blink`> tag does not work in Netscape browsers; subsequently, the association vector for each client configuration with the setting `Browser`=`Netscape` should correlate the <`blink`> tag with a high probability for failure.

In the inductive methodology presented here, web pages that comprise web-based applications are the raw material for training. HTML tags that structure each web application and the manual classification of the web application as either a *positive* (correctly executing) or *negative* (incorrectly executing) instance provides a statistical basis for determining the influence a given tag has on the web applications' execution in a client configuration. In the initial implementation, the first step in deriving tag support knowledge is to evaluate the correlation coefficient, ϕ, of each discovered tag [55, 36]. Intuitively, ϕ observes positive and negative phenomena to estimate the association of an element to one category or another. Note, the association vector is essentially a collection of ϕ values for all of the tags in a client configuration. Since the ϕ values are unique for each client configuration, one association vector is mapped to each point in the client configuration space. The following formula is used to compute ϕ for tag t and client configuration c:

$$\phi(t,c) = \begin{cases} \frac{\sqrt{N}\times(AD-CB)}{\sqrt{(A+C)\times(B+D)\times(A+B)\times(C+D)}} \\ 0, & \text{if A+C=0; B+D=0; A+B=0; C+D=0} \end{cases} \tag{4.2}$$

where (for a configuration c) N is the number of instances observed, A is the number of correctly executing instances that contain tag t, B is the number of incorrectly executing instances that contain tag t, C is the number of correctly executing instances that do not contain tag t, and D is the number of incorrectly executing instances that do not contain tag t.

Positive instances of web applications can be accessed, read, understood, and interacted with as intended by the developer. Negative instances of web applications, on the other hand, execute incorrectly in a client configuration. Note that since A+C is the total number of positive instances and B+D is the total number of negative instances, the denominator goes to zero if there are no positive instances (A+C = 0), no negative instances (B+D = 0). In addition, the denominator for ϕ evaluates as zero when there are no occurrences of a given tag (A+B=0), all positive and negative instances contain the tag (C+D=0) or there are no instances at all (A=B=C=D=0). When the denominator is 0, ϕ evaluates to 0.

Evaluation of ϕ: The use of ϕ as a predictive tool centers around the sign as well as the magnitude of the value. A negative value indicates that the tag is expected to be unsupported in the corresponding client configuration while a positive value indicates that the tag is expected to execute correctly. A value of zero indicates that the tag is not expected to have any influence on application execution. For example, this value is assigned to tags such as <HTML> that occur an equal number of times in both positive and negative instances. The magnitude of ϕ provides insight into the strength of association between the tag and the corresponding category (positive or negative) given the instances examined. The larger the value, the better the possibility that the tag has been correctly characterized. Subsequently, ϕ predicts the risk that an HTML directive is unsupported in a given configuration. Note, the tags themselves are not faulty, they are either supported or unsupported in a given environment. The appearance of an unsupported tag in a web application, however, increases the risk for faults if the application is launched in an incompatible envi-

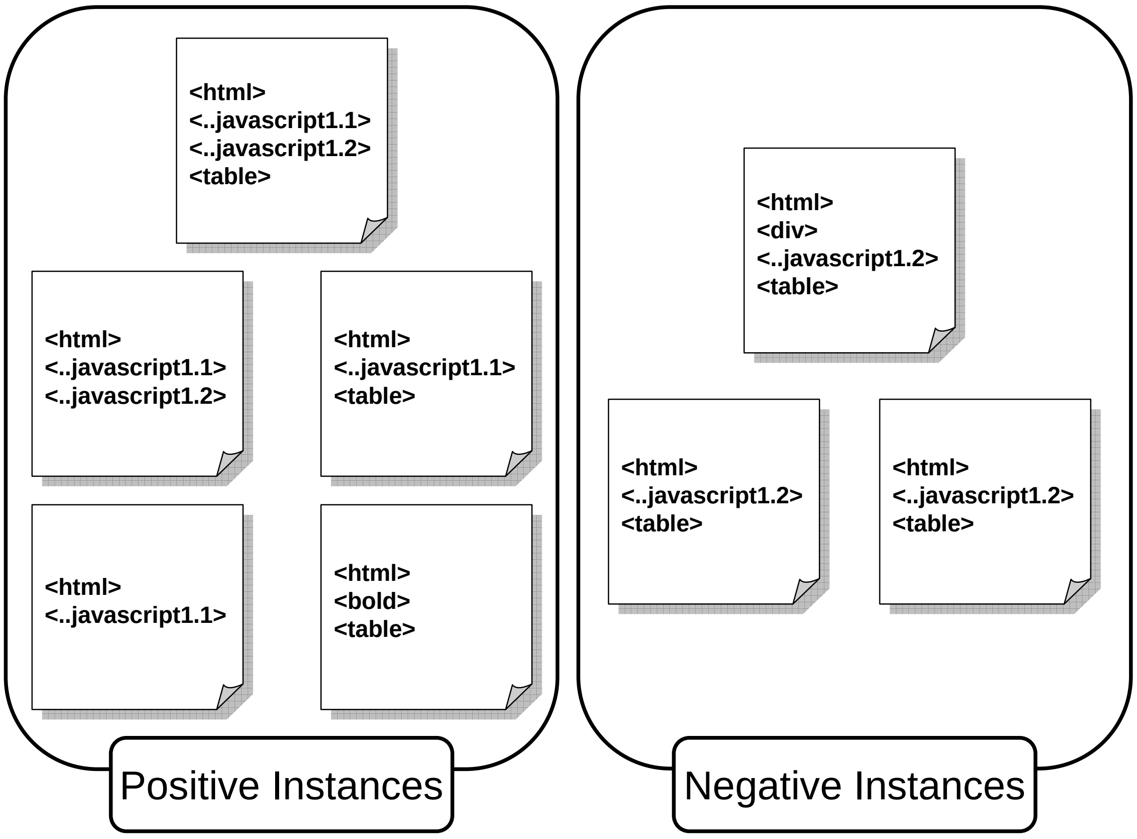


Figure 4.2: Set of Web Applications Classified as Positive or Negative.

ronment.

For a concrete example, consider the set of web applications classified as positive or negative for an arbitrary client configuration shown in Figure 4.2. Note, there is a combined total of eight applications in the set (*i.e.*, $N = 8$), five positive and three negative instances. Also note that the tag names have been modified to save space and improve presentation. Consider the <`div`> tag. It does not occur in any positive instance ($A = 0$, $C = 5$) and occurs in one negative instance ($B = 1$, $D = 2$). As a result, the ϕ value associated with <`div`> is -1.38. This suggests that web applications that contain this tag will execute incorrectly in this client configu-

Table 4.1: ϕ Values for All Tags in the Example of Figure 4.2.

Tag	html	div	javascript 1.2	table	javascript 1.1	bold
φ	0.00	-1.38	-.1.70	-1.26	2.19	0.83

ration. On the other hand, the `<bold>` tag occurs in one positive instance ($A = 1$, $C = 4$) but never in a negative instance ($B = 0$, $D = 3$). Accordingly, its ϕ value is 0.83. This suggests that the `<bold>` tag is supported in the client configuration, but since its magnitude (0.83) is smaller than that of the `<div>` tag (1.38), it has a weaker association with correct execution than the `<div>` tag has with incorrect execution. A full list of tags for all the instances is shown in Figure 4.2 and their corresponding ϕ values is shown in Table 4.1.

In the next section, the algorithms that accomplish the following tasks are described: (1) create/update an association vector when new positive/negative instances of web applications are available and (2) use the vector to test a given web application.

4.2.3 Algorithm to Generate/Update the Inductive Model

The association vector for a given client configuration is updated each time new information, in the form of positive and negative instances, is available for that client configuration. The `updateVector()` algorithm shown in Figure 4.3 is invoked *for each web-page* instance submitted.

```
1   Algorithm::updateVector(T /*currentPageTagset*/, Config /*clientConfiguration*/, isFaulty){
2           associationVector = getVector(Config);
3           IF (!exists(associationVector)) {associationVector = createVector(Config);}

4           /*update A and B values for the phi equation*/
5           FORALL t ∈ T DO {
6               IF (t ∉ Config.tagsSeen) { /* Have we seen this tag before? */
7                       associationVector.insertElement(t);
8                       insert(t,Config.tagsSeen);          }
9               IF (isFaulty) {t.incrementA()} ELSE {t.incrementB()}
10          }

11          /*update unsupported tag list for the current configuration*/
12          IF (isFaulty) {increment(negativeSeen);} ELSE {increment(positiveSeen);}
13          FORALL t ∈ T DO {
14              /*update C and D values for the phi equation*/
15              t.setC(positiveSeen-t.A);
16              t.setD(negativeSeen-t.B);
17              t.calculatePhi();

18              IF (t.associationStrength < 0){
19                      insert(Config.unsupportedTags, t);
20              ELSE IF ((t.associationStrength >= 0) && (t ∈ Config.tagsSeen)){
21                      delete(Config.unsupportedTags, t);
22              }}}
```

Figure 4.3: The `updateVector()` Algorithm.

As shown in Figure 4.3, `updateVector` takes three input parameters: (1) `T`, the set of tags in the web page, (2) `Config`, the client configuration encoded as a set of (option, settings) pairs, and (3) `isFaulty`, a boolean flag indicating whether the page executes correctly or incorrectly in `Config` (Line 1). If an association vector already exists for `Config`, then it is updated (Line 2); otherwise a new (empty) vector is created (Line 3). Each tag in `T` is processed one by one; a new entry is created for each new tag (not already in the vector). The A and B values (corresponding to the ϕ formula) are updated (Line 9). Recall that A and B correspond to the number of positive and negative instances respectively that contain the tag. The A associated with the tag is incremented if this is a positive instance; B is incremented if this is a negative instance. The number of negative/positive instances seen so far is incremented based on the status of the current instance (Line 12).

Once A and B values have been updated, C and D values can be derived (Lines 15-16) and ϕ can be recomputed for affected tags (Line 17). More specifically, C is the number of positive instances seen to date minus A, the number of positive instances that contain the given tag. Likewise for D and B for negative instances. Once A, B, C, and D are computed, the ϕ value is calculated; if it is negative (Line 18), the tag is inserted into a vector called `unsupportedTags` (associated with `Config`) (Line 19). However, if ϕ has a positive value and it is currently recognized as a unsupported tag in `Config`, it is deleted from the vector `unsupportedTags`.The `unsupportedTags` vector is used in the algorithm described in the next section. Recall, in this approach, ϕ is used as a predictive measure of tag support; as the values of ϕ change, the tag can be reclassified as supported/unsupported by the system. However, whether the tag is truly supported/unsupported in a given environment does not change. Although an "aggressive" algorithm that updates ϕ values each time a new positive/negative instance is available has been described, in practice, for reasons of efficiency, the update could be performed *on demand*, *i.e.*, computed when needed, or periodically after several new instances have been seen.

4.2.4 Algorithm to Use the Inductive Model

As mentioned before, the `query()` interface uses the inductive model to determine the set of configurations in which a given web application will execute incorrectly. An algorithm of the process is outlined in `queryData()` as shown in Figure 4.4. The algorithm takes one parameter: `W`, a web application which is a collection of web pages. The set of unsupported tags is retrieved for each client configuration in the inductive model (Line 3). The tags of each page in the web application are extracted (Line 5). If the web page contains at least one tag known to be unsupported in the configuration, the page is marked as faulty (Line 10) and the algorithm returns a set of <*client configuration, unsupported HTML tag, faulty web page*> triples and terminates. Note, in the initial phase of this work, details that might improve the efficiency of the overall process were not addressed.

```
1   Algorithm::queryData(W /*webApplication*/){
2           FORALL config $\in$ clientConfigurations DO {
                /*get the list of unsupported tags association with the current configuration*/
3               unsupportedTags = config.unsupportedTags;
4               FORALL w $\in$ W DO {
                    /*get the list of tags in the current web page*/
5                   currentTags = getTags(w);
                    /*check to ensure that the current web page does not include
                    any of the unsupported tags*/
6                   FORALL f $\in$ unsupportedTags DO {
7                       IF (f $\in$ currentTags) {
8                           RETURN_FAULTY(config,f, w); break}}}}
9
10
```

Figure 4.4: The `queryData()` Algorithm.

4.3 Empirical Study

In order to evaluate the feasibility and utility of the initial web application/client configuration models and the processes/algorithms that create/update and use the inductive model, an empirical study was conducted. The major research questions that were addressed center around the ability of the correlation coefficient,

ϕ, to distinguish between supported and unsupported tags and the impact of sample size on the results.

4.3.1 Infrastructure

In order to conduct the study, the algorithms listed in Figures 5 and 6 were implemented in Java. Subject usage environments were chosen for the study by varying several browsers, operating systems, and browser settings. In particular, Internet Explorer 6.0, Mozilla 1.5, Netscape 4.8, and Opera 6.0 were used on WinXP and Mac OS X platforms. In terms of individual browser settings, Javascipt was enabled/disabled. Hence the client configuration space contained $4 \times 2 \times 2 = 16$ points. These 16 points will be referred to as c_1 through c_{16}. A detailed listing of the client configurations associated with each point is provided in Table 4.2. This particular set was selected in order to reflect wide diversity in usage environments. To analyze the results, the gold standard, or actual knowledge of tag support rules, was modeled in Microsoft Excel and developed several visual basic scripts to summarize the data.

4.3.2 Empirical Method

4.3.2.1 Research Questions and Evaluation Strategy

The empirical method utilized was designed to answer the following questions:

1. Do many fielded web applications really have client-configuration-specific problems?

Table 4.2: Configuration Point Details.

Configuration Point	**Client Configuration**
c_1	< Netscape 4.8, WinXP, Javascript enabled >
c_2	< Netscape 4.8, WinXP, Javascript disabled>
c_3	<Netscape 7.01, Mac OS X, Javascript enabled>
c_4	<Netscape 7.01, Mac OS X, Javascript disabled>
c_5	<Internet Explorer 6.0, WinXP, Javascript enabled>
c_6	<Internet Explorer 6.0, WinXP, Javascript disabled>
c_7	<Internet Explorer 5.0, Mac OS X, Javascript enabled>
c_8	<Internet Explorer 5.0, Mac OS X, Javascript disabled>
c_9	<Opera 6.0, WinXP, Javascript enabled>
c_{10}	<Opera 6.0, WinXP, Javascript disabled>
c_{11}	<Opera 6.0, Mac OS X, Javascript enabled>
c_{12}	<Opera 6.0, Mac OS X, Javascript disabled>
c_{13}	<Mozilla 1.5, WinXP, Javascript enabled>
c_{14}	<Mozilla 1.5, WinXP, Javascript disabled>
c_{15}	<Mozilla 1.5, Mac OS X, Javascript enabled>
c_{16}	<Mozilla 1.5, Mac OS X, Javascript disabled>

2. How well does the association vector approach help to identify such problems?
3. How much manual effort is involved in identifying and submitting positive/negative examples of web applications?
4. How much manual work is involved in classifying web applications returned by the automated acquisition process as negative and positive?
5. Are the results obtained from this technique always accurate? Are there any false positives?
6. How is the rate of false positives affected by the total number of observed instances?

The first question is important mainly because it justifies the purpose of this research. In the same vein, the second question was designed to analyze the utility of an *association vector* model for client configurations. The third question was posed because users play a key role in the application of the proposed approach; ease of use, therefore, is an important consideration. The fourth question, on the other hand, addresses the ability to utilize user input to expand and improve the model. To address the spirit of the fifth question, the misclassification of tags, in terms of false positives (FP), is expected to have a large impact on feasibility of the approach. To be clear, negative classification of a tag is indicates that the tag is not supported in a given environment; positive classification indicates that the tag is supported. Subsequently, a false positive is an unsupported tag labeled incorrectly as supported. Since a direct consequence of a false positive is that a faulty page could unwittingly be released into the field, it is necessary that the measure used to evaluate the approach penalizes techniques that allow for more false positives. The measure which fits these criteria,FPR, is defined and and calculated as follows:

$$FPR = \frac{FP}{total\ number\ of\ tags} \tag{4.3}$$

The sixth and final question was posed to observe whether FPR improves as more information is obtained.

4.3.2.2 Independent and Dependent Variables

The only independent variable in this study is the size of the training set. The dependent variable is the accuracy of tag classification predictions measured here by FPR. Because client configurations are simply subjects in the experimental design, the client configuration is neither an independent or dependent variable.

4.3.2.3 Experimental Procedure

The following process will be used to conduct the study:

Step 1: Select a set of client configurations, C, where c_x is the x^{th} configuration in C and

$1 \leq x \leq 16$.

Step 2: For each $c_x \in C$, select an initial pool, P_{c_x}, of positive and negative web pages.

Step 3: Parse web application source code, extract the HTML tags, and abstract the tags using conditioning technique, TC. This will produce $P_{c_x,TC}$, a representation of the positive and negative web pages in which tags contained in the source have been processed to facilitate the inductive process. Tag conditioning is explained further in Section 5.2.6.

Step 4: Generate the gold standard of tag support rules for later evaluation.

Step 5: Evaluate ϕ for tags discovered in a set of web pages using the following sub-steps:

A. Randomly select 50 web pages from c_x without replacement.

B. Generate the inductive model by calculating ϕ for $P_{c_x,TC}$.

C. Calculate the corresponding FPR value.

D. For five iterations...

i. Randomly select 25 web pages from c_x without replacement. (None of the web pages selected during this step will have been observed in any previous steps)

ii. Generate the inductive model by calculating ϕ for $P_{c_x,TC}$.

iii. Calculate the FPR value for the inductive model.

A detailed account of each step follows in subsequent sections:

4.3.2.4 Step 1: Client Configuration Selection

The overall strategy in selecting subject client configurations was to include a broad range of older and newer browsing environment configurations; this was done to reflect the widely varied usage profiles in use in the "real world". The set of 16 configurations we chose with this criteria in mind is shown in Table 4.2.

4.3.2.5 Step 2: Training Set Selection

Each of the 16 configurations listed in Table 4.2 had an initial application pool of 200 web pages, 100 negative instances and 100 positive instances. Some of the negative instances are shown in Table 4.3. Retrieval of positive and negative web pages was guided by the gold standard that provided data on the tag support in the various environments. The Google search engine was used to locate pages that incorporated fault-inducing tags. Because Google ignores the brackets (" < " and " > ") that sandwich HTML tags and there was no feasible way to pose queries to ensure that pages which merely mentioned the tag name and did not actually use it were not included in the result set, retrieval of web pages with desired tags was a challenge. More specifically, Google provided a basic mechanism for locating pages, identifying returns that were actually useful was tedious at best.

Once all the web applications had been identified, submitting them to the `updateKB()` component of the framework for automated analysis took a few seconds per web page. Note, submission to `updateKB()` only entails saving the source code of the web page and identifying it as either a positive or negative instance. This is currently implemented with a folder reserved for positive instances and a folder reserved for positive instances; users save the source code to the appropriate folder for later analysis.

As expected, some of the tags existed in too few web applications to accurately predict whether the client environment provided support. For example, for a certain configuration, the tag <`html lang = en`> occurred in 13 positive instances and no negative instances. Similarly, for another configuration, <`div align = left`> appeared in 5 positive instances and 8 negative instances. The inductive model did not contain sufficient information about these tags to be useful for analysis; subsequently, the automated acquisition process was initiated using the Google search engine. More specifically, queries were posed to the Google engine that would retrieve web pages with a given tag. An example query that was used to retrieve web pages containing the <`html lang = en`> tag is $\boxed{\text{html lang en \{href head\}}}$. The latter query elements (shown in curly braces) were issued when the query posed by the first three terms yielded too many pages which only mentioned the tag. Once pages which actually used the tag were returned, they were loaded in the associated client configuration and observed to determine whether they were positive or negative instances. Negative instances that had visual abnormalities were relatively easy to identify. Negative instances with non-visual errors (such as non-support for the `accesskey` tag had a greater chance of being incorrectly labeled as a positive instance.

Table 4.3: Part of the Negative Instance Set of the Initial Web Application Pool.

URL	**Configuration Point**
www.hasbro.com/scrabble/home.cfm	c_1
home.netscape.com/	c_2
www.nasa.gov/externalflash/exp12_front/index.html	c_3
www.juiceguys.com/	c_4
www.richinstyle.com/bugs/operademo.html	c_5
www.ameristarcasinos.com/cactus/index.asp	c_6
www.useractive.com/learning/dhtml/dhtmltut7.php3	c_7
www.simonstl.com/dynhtml/update/code/chap5/onbounce.html	c_8
retreatvillage.com/activities.html	c_9
www.gsn.com/	c_{10}
www.physics.utah.edu/news/y04m02d26.html	c_{11}
www.sinel.com/esp/home.htm	c_{12}

4.3.2.6 Step 3: Tag Extraction/Abstraction

The `updateKB()` interface accepts the HTML source code of positive and negative web pages as raw data and extracts the HTML tags incorporated in the page. In order to derive tag support knowledge from the submitted instances, however, tag data must be conditioned. Given the inductive nature of the algorithm, tag representation has a significant impact on the quality of tag support rules learned. HTML tags can be represented in raw form during inductive knowledge discovery or they can be conditioned so that certain features are filtered; this results in folding a series of tags that differ by at least one variable into one representation. Indeed, such conditioning could drastically reduce the number of tags considered during induction, while, perhaps, losing important information in the process. To put it in perspective, this issue is directly aligned with the challenge of modeling web applications.

More specifically, certain HTML attributes such as `width`, `href`, and `summary` have numbers, URLs, and strings of text as values. The goal is to avoid discriminating between these tags based on their specific values. To understand how this problem is handled , consider the following:

<`table summary="XYZ"`> and

<`table summary="ABC"`>.

In this case, the learning algorithm should only consider the instance of the tag `table` and the attribute `summary`. As a result, both tags are collapsed into one, and represented as <`table summary="#"`> in the association vector.

On the other hand, there are some instances when knowledge of the attribute value is key. Such is the case for the following:

`<script language="javascript1.3">` and

`<script language="javascript1.1">`.

Subsequently, abstraction strategy was designed in which a carefully selected number of predefined attributed values are preserved; attributes with number and URL values, for example, are collapsed. Once tags are discovered, `updateKB()` automatically conditions them.

4.3.2.7 Step 4: Defining the Gold Standard

Generally speaking, the gold standard serves as a ground truth, a way to compare derived values to known values in order to estimate how well a mechanism performs its task. In this case, the ground truth is the actual support provided for a given tag in a particular client browsing environment. The ground truth, in the current implementation, was modeled with the help of a website that provides tag support data. The gold standard can be modeled as a function, GS, that accepts a tag, t and a configuration, c_x, as input and returns a boolean value that indicates support or non-support as output. A more formal definition is provided below in Equation 4.4.

$$GS(t, c_x) = \begin{cases} yes & \text{if tag } t \text{ is actually supported in configuration } c_x \\ no & \text{otherwise} \end{cases} \tag{4.4}$$

4.3.2.8 Steps 5: Tag Classification and Evaluation

Recall, the correlation coefficient, ϕ, is used to predict whether a tag is either supported or unsupported in a target client configuration. During Step 4 (Section 5.2.7), the ϕ-based tag classification strategy was applied to the data and the FPR value was calculated. To determine the number of false positives, the function AR (actual results), which is analogous to the GS calculation shown in Equation 4.4 was used. More specifically, AR is a function that accepts a tag, t, and a configuration, c_x, as input and returns a boolean value that indicates support or non-support based on the inductive model. Like GS, AR is modeled as shown below:

$$AR(t, c_x) = \begin{cases} yes & \text{if } c_x \text{ is predicted to support } t \\ no & \text{otherwise} \end{cases} \tag{4.5}$$

Subsequently, a false positive occurs when $GS(t, c_x) = no$ and $AR(t, c_x) = yes$. In this step, the predicted classification of the tag was compared with that of the gold standard and the FPR equation was used to determine how well the classification strategy performed.

4.3.3 Threats to Experimental Validity

4.3.3.1 Internal Validity

The internal validity of experimental results is threatened when results of the dependent variable could be tainted by modeling and measurement errors. In each of the questions addressed, FPR is the primary dependent variable. Hence threats to internal validity, in this context, include possible errors in measuring/designating the training set and modeling/executing both the tag abstraction scheme and tag classification strategy.

Another threat lies in the correctness of the gold standard. The source used as the basis for the gold standard, in some instances, relies on the documentation provided from the browser manufacturer. Since this can be erroneous at times, it can have an undesirable impact on false positive rate evaluations. More specifically, recall that a false positive occurs when $GS(t, c_x) = no$ and $AR(t, c_x) = yes$. If $GS(t, c_x)$ should actually be *yes* in a given case, but it incorrectly returns a value of *no* as a result of incorrect documentation, the FPR will evaluate to a higher value than it actually should.

4.3.3.2 External Validity

Threats to external validity, on the other hand, limit the generalizability of results. Several candidates for this constraint apply. For one, only pages in which there are HTML-induced faults that can be linked to a certain tag and not, perhaps, Javascript errors that can be linked to a faulty variable are currently considered. Other threats include possible misclassification of web pages on the part of submitters and low usage of a given client configuration platform (resulting in less training data for the inductive algorithm). Given an overall expectation that inductive model accuracy will improve as more examples are submitted, the volume of data provided is important. This has been acknowledged in the attempt to include an adequate number of pages in the experiment; similar considerations must be made to ensure the success of the tool in practical settings.

4.3.4 Results and Discussion

Recall, there were six major questions that were to be addressed as a result of this study. Concerning the first question, quite a few web applications that rendered and behaved properly in one environment yet were faulty in another (examples of this follow in Figure 4.6) were observed. In addition, upon discovering the nature of this work, several individuals who have run across such problems in very frustrating situations have shared their stories. Subsequently, client-configuration specific problems are a reality in many fielded web applications.

In the case of the second question, the results of this study showed that even with a relatively small set of 200 instances, the approach of using the association vector was successful at detecting client-configuration-specific tag support issues in fielded web applications. In addressing the third question, which dealt with the manual effort involved in submitting examples, conducting the experiment revealed that it takes web application a few seconds to report a problematic web application and associated client configuration, indicating minimal manual effort. Note end-users are not responsible for indicating why an error occurred; they merely submit faulty pages to `updateKB()`, hence the low amount of manual effort. One key part of this approach is that it allows end-users using a given configuration who have discovered a problem in their normal web navigation to help improve the knowledge of supported and unsupported tags by merely submitting the raw source code. In regards to the fourth question, classifying web applications returned by the automated acquisition process as negative or positive took a few seconds. This classification process mainly entailed loading the page, observing, and interacting with it to ensure that it rendered and executed properly. In some cases, an unsupported tag with non-visual effects could be included in the source code of a page that appeared to be a positive instance. Such misclassifications served as the root cause for the appearance of false positives in respective inductive models.

In regards to the fifth question, which addresses the performance of the technique, the inductive model utilized in this study yielded useful results. Given the data generated as a result of this study, the preliminary approach rarely labels an unsupported tag as supported in a given environment. While this model provides a promising basis, in principle, it is not complete. This is largely because of two issues: (1) information for every possible tag is not included in the association vector and (2) the tag information represented is not extensive. Consequently, false positives were observed during data analysis.

The graph shown in Figure 4.5, which plots the rate of false positives as the training set size grows, was generated to address the sixth and final questions. As evident, the false positive rate remains low for each of the environments. This, of course, is a promising result since this indicates that the approach we use has a low incidence of labeling an unsupported tag as supported in a given environment. One issue, however, is what appears to be fluctuating FPR values. Note, however, that this occurs at alternating points in the graph. Currently, this is attributed to the use of more negative examples for the training sets with 75, 125, and 175 examples. When the training set was 75, as an example, there were 38 negative examples and 37 positive examples. Taking this into consideration, it appears as if the FPR trend continues down as the training set grows, for every other data point. More specifically, the rate of false positives generally decreases from 50 to 100 to 150 and from 75 to 125 to 175. Subsequently, it can be concluded from this data that the results generally improve as training set grows and that the false positive rate is best when there is more negative examples than positive examples.

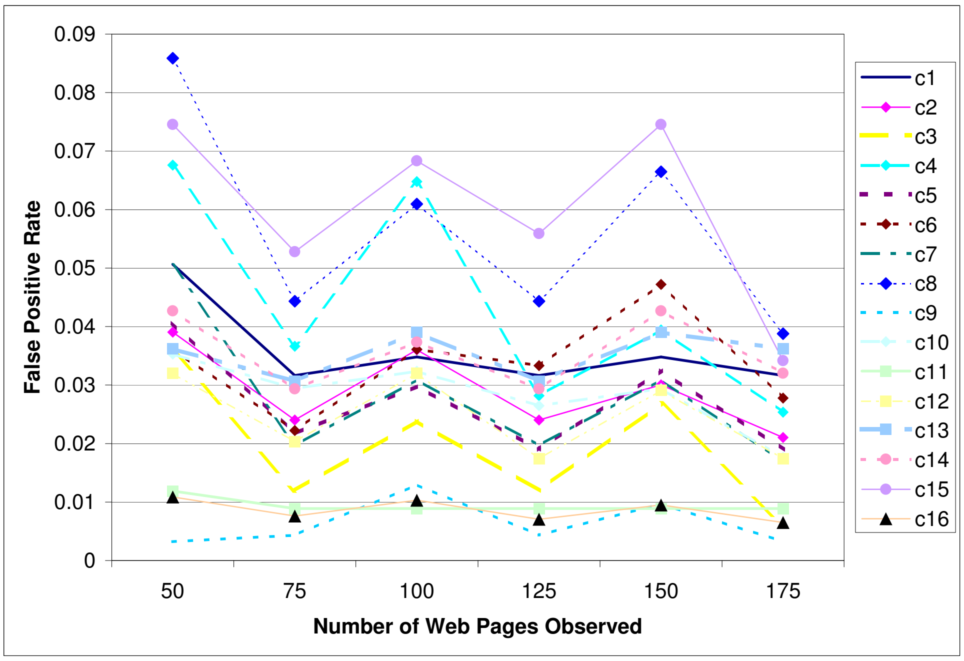


Figure 4.5: False Positive Rate with Respect to Training Set Size.

Web Application Evaluation: Once the inductive model had been derived from real-world data, it was used to evaluate a new set of web applications. Twelve (12) popular web applications were selected as subject applications. Note that these applications were not part of the application pool used to create the model. They

Table 4.4: Evaluation Results.

#	**URL**	**Client**	**Problematic Tags**
1	www.aidsreagent.org/	$c_{2,4,6,...,16}$	$<script ... = "JavaScript">$
2	www.radissonedwardian.com/aboutus/home.jsp	c_1	$<..style = "height : ..>$
3	www.jegsworks.com/demos/DemoDHTML/bghead.htm	c_{5-16}	$<layer\ bgcolor = \#>$
4	students.washington.edu/siutai/index.shtml	$c_{2,4,6,...,16}$	$<div\ onmouseover() = \#>$
5	members.dcn.org/ez112654/html/h51.html	c_{5-16}	$<blink>$
6	www.musiciansunion.org.uk/html/index.php	$c_{1..4,9..16}$	$<a\ accesskey = \#>$
7	www.execlangser.com/	$c_{1..4,9..16}$	$<basefont\ face = \#>$
8	www.koko.gov.my/CocoaBioTech/Southern.html	$c_{1..4,9..16}$	$<marquee>$
9	www.sltrib.com/	c_{5-16}	$<ilayer\ bgcolor = \#>$

are shown in Column 1 of Table 4.4. Several problems with these applications were discovered. The client configuration in which these applications did not execute correctly is shown in Column 2 of Table 4.4; Column 3 shows one of the problematic tags.

Screen-shots of some of these applications are shown in Figure 4.6. Each of the examples featured in Figure 4.6 have visibly evident errors. Yet, as noted before, while an image of web applications functioning in varied environments alerts (indicates) the existence of problems, the causes of such problems can only be identified with deeper analysis. Using the documented technique, however, the missing menu elements in the far left corner of the `NIH AIDS Research & Reference Reagent Program` web application can be attributed to lack of support for the javascript tags which specify properties of the menu items. The ill-formatted `Radisson Edwardian` web page can be attributed to the fact that Netscape 4.8 does not properly render the `height` value of the `style` attribute. Moreover, the barely visible text on the `Executive Language Services, Inc.` header can be associated with Opera's non-support for the `face` attribute of the `basefont` tag. One example in the table, however, namely the Musicians Union website (#6 in the table), highlights the problem when screen-shots are not enough to recognize errors. More specifically, the web application features "hotkey" access to various tool components, yet this functionality is not available in Opera 6.0. The technique outlined here enabled diagnosis of the root cause of this problem; namely a lack of support for the `accesskey` attribute in client platforms which feature Opera browsers.

As noted in Table 4.4, several tags caused problems in multiple client configurations. An example is the <`div onmouseover()= #`> tag that caused problems in configurations $c_{2,4,6,\ldots,16}$. In some specialized instances, a given tag will influence incorrect effects within client configurations that share an environmental characteristic. In this particular case, each of the environments for which <`div onmouseover()= #`> produced faulty results had Javascript disabled. In future work, the nature of such instances will be explored. Recognizing that such similarities exist can effectively prune the search space utilized by `queryData()` (Section 4.2.4) since the detection of a such a tag would signal failure for all associated browsing environments.

While tag presence alone can influence faulty behavior, the arrangement of tags and well-formedness of web applications can also have a significant impact on behavior and appearance. In addition, the degree of impact can be heavily influenced by the client configuration used to launch the web application. Consider Figure 4.7 as an example. Here the presence of faulty tags is not the issue. In this case, the fact that a tag was not properly closed affects the ability of site visitors using Netscape 4.8 to utilize the search functionality featured on this page. In a sense, the Mozilla browser is more forgiving when source code is not well-formed. Indeed, the featured approach currently does not detect the existence of such errors. In cases such as these, HTML validators will be useful in diagnosing such issues.

4.4 Summary

As a result of the initial work, several goals were accomplished. In terms of the smaller picture, an empirical technique based on association vectors, each mapped to formally defined client configurations was explored. In this phase of the work, vectors evolve for specific environments as additional empirical data in the form of correctly/incorrectly (positive/negative) executing web applications on specific platforms becomes available. Results of an empirical study showed that the approach is feasible and useful. Several client configuration specific problems in fielded web applications were discovered. It appears that the modeling choices made were effective in capturing and detecting client configuration faults, yet changes must be made to encourage the acquisition of more complex tag support rules such as tag interactions. Also, in the initial work, only HTML support knowledge is learned. In future endeavors, a richer basis for learning, that includes CSS directives, must also be considered.

In terms of the bigger picture, the initial work provided a strong basis for further exploration. Many of the basic back-bone algorithms were implemented with provisions made to easily substitute different web application and client configuration models and to utilize different inductive learning approaches. In addition, quite a few subject applications were collected. Subsequently, much of the engineering issues were taken care of; slight modifications were expected over the remainder of this research, but quite a bit of work was accomplished. Future work builds on this work by allowing more flexibility in web application/client configuration modeling, exploring varied learning strategies, and accounting for tag interaction.

Chapter 5

Current Framework Implementation

In the initial phase of this work (outlined in the previous chapter), client configuration and web application models were developed to facilitate portability analysis, an inductive learning approach to support knowledge acquisition was implemented, algorithms for updating knowledge of tag support criteria and querying the configuration model during portability analysis were designed, and each of these components were integrated into a prototype framework. In that phase of the work, the web application model, client configuration model and subsequent analysis were based on HTML tags[1]; detection of configuration faults involved identifying unsupported HTML tags in the source; support knowledge was maintained locally for each client configuration; and fault detection analysis results was presented as a text-based list. In addition, the primary method for acquiring source code support knowledge was a machine learning method and the learning mechanism was only evaluated based on the false positive rate of classification. Though feasible and effective, as established by an empirical study, the initial approach was weakened by a few key factors. The attempt to correct key approach deficiencies, build upon the initial work, make analysis more efficient, and maximize framework practicality have been addressed as follows:

- Initially, tag support knowledge was solely acquired through machine learning. To allow for a more comprehensive knowledge store, alternate data acquisition strategies have been integrated; they include enabling human experts to update tag/rule support knowledge directly and collecting more information when confidence in classification is low after machine learning.

- Recognizing that lack of support for CSS rules can have a negative effect on web application portability, inclusion of CSS rules in web application and client configuration models has been expanded in addition to the HTML tags captured in prior work.

[1]Inclusion of CSS rules was limited to in-line elements

- In the previous portability analysis strategy, the mere inclusion of an unsupported tag in source code served as a configuration fault index. For more precise analysis, the impact of support violation offsets (Section 2.1.4, *Definition 3*) has been integrated into the current framework. To account for this phenomenon, the client configuration model has been converted from a vector to a matrix in which the intersection of columns and rows account for the interaction of tags.

- Knowledge base locality was a key deficiencies in the initial work. Since each configuration accounted for in the framework maintained its own local knowledge base, `query()` had to access each individual instance of support knowledge to achieve full configuration space coverage. To allow for more efficient analysis, support knowledge is currently maintained in a global location. Among other factors, the updated model enables commonalities between configurations to be accounted for and, consequently, analysis and subsequent presentation of results is improved.

- Once configuration faults were detected, analysis results were previously returned as a flat list of ⟨unsupported HTML tag, web page, configuration⟩ triples. When a large number of configuration faults are detected, this list is long and difficult for humans to process efficiently. In this phase of the work, visualization techniques were investigated to encode analysis results in a format that encourages quick, efficient overview of the configuration faults discovered during analysis.

- Finally, in the initial evaluation of the inductive approach to support knowledge acquisition, the effectiveness of the learning technique was measured by the rate of false positives. Currently, a more holistic measure, *accuracy*, is used as the evaluation criterion.

5.1 Current Framework Design

In the effort to enhance key framework components, feature more accurate, efficient portability analysis, and improve presentation of results the previous approach has been extended in the following ways: CSS rule inclusion has been expanded in the web application and support criteria models; detection of tag interaction was implemented in order to account for support violation offsets; support knowledge was centralized; a minimization algorithm was explored to simplify configuration support pattern descriptions; and visualization techniques were investigated to encode analysis results in a format that encourages quick, effective repair of detected web configuration faults. In this section, details of these extensions are discussed and a bridge between the current approach and the general web portability analysis framework introduced in Section 3.1 is explicitly outlined. The current instantiation is represented visually in Figure 5.1. Recall, generic framework components include some number of *knowledge bases* and the {`updateKB()`, `processURL()`, `query()`, and `generateReport()`} interfaces. A discussion of the current instantiation begins with an overview of how support criterion is modeled (knowledge base, Section 5.1.1) and accumulated (`updateKB()`, Section 5.1.2). A discussion of how that knowledge is utilized once web application source code is retrieved (`processURL()`, Section 5.1.3) to perform web configuration fault detection (`query()`, Section 5.1.4) follows. This section concludes with an overview of how analysis results are presented to users (`generateReport()`, Section 5.1.5).

5.1.1 Knowledge Base

The knowledge base is a critical component in the outlined approach because it maintains a predictive model of source code support across the configuration space and is used during portability analysis to detect support violations in source code inclusion. While the depth, breadth, and overall quality of support knowledge are important to analysis accuracy, the locality and structure of knowledge are significant factors in analysis efficiency. To optimize the knowledge base design in terms of *locality*, domain knowledge is stored in a central, globally accessible location. With regard to *structure*, code support knowledge is concisely modeled on two distinct levels: support criterion is listed at the primary level and the configurations that lack support are listed on the secondary level. This design (shown in Figure 5.2) contributes to analysis efficiency because it enables *code-centric* analysis, where the number and support of fragments in web application source code drives analysis instead of the number of configurations represented. The initial approach featured a more *configuration-centric* analysis where, for each configuration represented, `query()` examined web application source code for inclusion of unsupported source code patterns. This factor caused the time to complete analysis to rely most heavily on the number of configurations represented in the framework. With *code-centric* analysis, the paradigm shifts such that, for each source code element, `query()` identifies unsupported environments. This allows more focused analysis since each query is a direct index to a consolidated list of unsupported configurations; in configuration-centric analysis, configuration support must be accumulated by checking each individual configuration to discover support issues. The remaining discussion of the knowledge base begins with the support criteria model on the primary level and concludes with the configuration subspace model on the secondary level.

5.1.1.1 Support Criterion Structure

Two source code support issues are considered during web portability analysis. In the *deterministic* case, the mere inclusion of an unsupported source code fragment introduces configuration faults. In the *conditional* case, source code inclusion is not an accurate fault index on its own; the impact of the unsupported source code fragment is dependent upon the inclusion or absence of a *support violation offset* (*i.e.*, a counteractive tag that enables portability when a related tag is unsupported, [Section 2.1.4, *Definition 3*]). To get a better understanding of the conditional case, consider two HTML tags <U> and <S> in client configuration *c*. Assume that <U> is unsupported in *c* (*i.e.*, web applications that contain <U> have rendering/execution problems). Tag <S> is supported. However, if an application contains both <U> and <S>, then it executes/renders correctly in *c*. Hence

<S> is a support violation offset for <U>. As a practical example of this factor, different versions of a tag or scripting language can be co-included in the source just in case one or the other is unsupported (Figure 5.3). Since a configuration fault will only result if the support violation offset is omitted from the source code, it is important that the offset is accounted for during analysis to improve accuracy of results. With co-inclusion of a support violation offset, a page should not be flagged as faulty.

In alignment with the support issues, two types of criteria are represented in the knowledge base on the primary level depending on whether they denote a lack of support (deterministic) or the existence of support violation offsets for a source code fragment in constituent configurations (conditional). In the deterministic case, a support criterion is represented with a single source code element on the primary level and the configurations listed on the secondary level lack support. General templates for HTML tags and CSS rules on the primary level are, respectively, as follows:

$$< \text{tag attribute} = \text{value} >$$

$$\{\text{property} : \text{value}\}$$

Note, the selector is not included in the CSS rule to make the criterion general.

On the other hand, support criterion with a conjunction of tags/rules on the primary level indicates that the initial tag has conditional offsets, listed subsequently, one of which must be included to avoid support violations in corresponding configurations. In other words, unless the offset tag is included in the source code, the listed configurations will lack the proper support. This notation, which joins tag templates with simple logical connections, allows interaction (read violation offsets) between source code fragments to be accounted for during analysis and helps to ensure that configuration fault indices are only reported as such if the offset is not included in the code. An explicit representation of the conditional case in the knowledge base helps to avoid the problem of false reports in instances when an offset violation applies.

A general template for conditional support criteria on the primary level follows:

$$\overbrace{< \text{tag attribute} = \text{value} >}^{\text{support criterion}} \wedge !(\overbrace{< \text{tag attribute} = \text{value} >}^{\text{support violation offset}})$$

where the "support criterion" will trigger a configuration fault unless "support violation offset", preceded by the "!" symbol, is included in the web application source code as well.

5.1.1.2 Knowledge Consolidation

By design, the configuration subspaces represented on the secondary level of the knowledge base lack support for the corresponding source code patterns listed on the primary level. In the knowledge base, the configurations are modeled as a conjunction of components/settings. Instead of listing each unsupported configuration individually, one endeavor is to refine support criteria by simplifying the representation of the unsupported configuration subspace. This will be particularly helpful when one configuration characteristic, such as disabled Javascript, can accurately summarize the entire subspace. To retain model fidelity, the generalized result should reliably summarize the description of the configuration subspace with lacking support for the source code pattern on the primary level.

To address this challenge, the Quine-McCluskey (QM) algorithm is used on the secondary level to reduce descriptions of configurations so that only critical component attributes are retained. The QM algorithm is primarily used in mathematics to simplify expressions and improve human-readability. The idea is to reduce a function to a set of *prime implicants* from which as many variables are eliminated as possible.

In the application of QM in this work, configurations are terms and their components are represented as variables. A configuration that has Internet Explorer (IE) as its browser and javascript (JS) disabled could be represented as $\mathrm{IE} \wedge \overline{\mathrm{NS}} \wedge \overline{\mathrm{JS}}$ where NS is Netscape and the bar over the variable indicates that the entity is not included in the configuration. Also, consider an environment in which IE is the browser and javascript is enabled $\mathrm{IE} \wedge \overline{\mathrm{NS}} \wedge \mathrm{JS}$. If a given tag or rule was unsupported in both of these environments, QM could be applied to reduce the description of unsupported environments to simply IE meaning that any client configuration with Internet Explorer as the browser will lack support. Since the subspace is returned to users as a part of the analysis report, it is important to consolidate the description of unsupported configurations into a concise representation to enable a more efficient result overview and maintain human-readability.

5.1.2 `updateKB()`

While HTML/CSS support knowledge provides a simple, effective basis for portability analysis in this approach, configuration fault detection will only be as thorough and accurate as the knowledge of tag and rule support used during analysis. The more support criteria known, the more configuration faults the system can accurately identify. Given that web technology and access options are constantly changing, one critical design consideration for a web portability analysis framework is that it be extensible in the breadth of client configurations covered and the depth of support knowledge maintained. To the author's knowledge, there is no comprehensive knowledge base that accurately maintains support criteria for the diverse configuration space or a complete account of support violation offsets, yet both are necessary to achieve an acceptable level of analysis accuracy. As a result, practical use of the framework developed poses the challenge of capturing a comprehensive, accurate support criteria model with reasonably low overheard [32].

This challenge has been addressed by using multiple knowledge acquisition methods. In the current framework, knowledge of source code support is derived from a combination of expert knowledge (manual update), a machine learning technique (automated acquisition), and a mechanism to solicit more data about learned rules when necessary (information solicitation). Respectively, these processes involve accepting input directly from domain experts (*i.e.*web developers familiar with code support), accumulating knowledge of support indirectly from practical examples, and gathering more evidence of source code support when there is low confidence in the rules derived from automated acquisition. Each of these strategies have been implemented as a subcase of `updateKB()`. In this section, the design of each `updateKB()` subcase is surveyed with particular attention to the main contributors of knowledge in each process, the rationale for including the strategy, and a basic overview of the process. A sample knowledge base is derived along the way based on contributions from each strategy.

5.1.2.1 Manual Update

Web developers and domain experts aware of support criteria within and across web client configurations are the key contributors in manual knowledge base updates. This subcase was specifically incorporated into the framework to integrate expert knowledge into the configuration fault detection process and is viable, in part, because the knowledge base is structured in a simple, human readable form. With the manual update, domain experts are allowed to modify source code support knowledge directly through an interface called a *Criteria Editor*. If a given configuration in which tags/rules are known to be unsupported is missing from the knowledge base, experts are able to add to it by listing the tag/rule on the primary level and configurations with support violations on the corresponding secondary level. Let's begin our practical discussion of how support knowledge evolves with a sample knowledge base that is initially empty. Figure 5.4 results after a domain expert accesses the knowledge base model through the *Criteria Editor* and transfered tacit source code support knowledge.

5.1.2.2 Automated Update

Users who encounter faulty web applications during normal web usage are the key contributors in automated knowledge base updates. The primary focus of this subcase of `updateKB()` is to automatically build knowledge of source code support criteria from user experiences; the idea is to apply a supervised machine learning technique to a large corpus, or body, of labeled source code examples and to discover the relationship between inclusion of a tag/rule in source code and configuration faults.

The automated subcase of `updateKB()` is designed to observe positive and negative examples and automatically characterize source code inclusion patterns that differentiate negative examples from positive ones. Recall that positive web applications are those that render and function properly in a given configuration; negative instances have rendering and/or functionality errors. Effective implementation of this strategy would allow support knowledge to evolve automatically and incrementally as data in the form of *positive* and *negative* examples is submitted by users.

To get a general idea of how machine learning can be applied to reveal the basic connection between the inclusion of tag/rules in source code and anomalous outcomes, consider the eight examples shown in Figure 5.5. Each positive (negative) example was submitted by a user with a particular client configuration who was able (unable) to correctly interact and view the corresponding web page. The depiction of each web application shown in the figure is the model that results once all HTML/CSS has been extracted for each submitted web application.

Once the models have been extracted and grouped according to label, the next step is to compute the correlation between the inclusion of source code elements and web applications labeled as negative examples. Assume an abstraction of the learning mechanism returns the following results:

Tag	html	div	js 1.2	table	js 1.1	bold
Supported?	$\surd$	$\times$	$\times$	$\times$	$\surd$	$\surd$

Here a $\surd$ indicates that, based on the evidence provided, the learning algorithm predicted the tag to be supported in the given environment and an $\times$ indicates that the tag is expected to be unsupported. Given the HTML/CSS labeled unsupported by the learning mechanism, the results of this learning iteration would update the criteria by appending the new knowledge as shown in Figure 5.6. Rules derived by the *learner* have been appended to the original criteria.

5.1.2.3 Information Solicitation

Information solicitation can be characterized as the middle ground between the manual and automated update strategies previously discussed. Recall that the automated update allows for the approximation of support criteria given examples; meanwhile, the manual approach allows domain experts to update the knowledge base through direct data access. As a hybrid, information solicitation allows users to update the knowledge base by investigating whether the expected impact of a tag/rule was fully realized in a given environment and updating the knowledge base manually if it was not. Unlike web users who contribute to the automated approach, contributors in information solicitation do not encounter support violations randomly during normal web usage. Instead, they are given a particular source code fragment to investigate. On the other hand, unlike domain experts who contribute through the manual approach, they are not expected to have prior, tacit knowledge of source code support.

To understand the rationale behind *information solicitation* (IS) as an `updateKB()` subcase, consider the following: assume that after completion of one learning iteration, a given tag is predicted to be unsupported, yet this conclusion was drawn after observing the tag in only *one* negative example. In the learning example outlined in the previous section, this is exactly the case for the <`div`> tag. Given the design of the automated approach, instances like these are expected to be quite common. As users provide positive/negative web application examples to the framework for analysis, instances when source code inclusion is detected in a low number examples can cause confidence in subsequent analysis results to be low as well. Note that learning only allows approximation of support criteria given labeled examples. Since the learning mechanism relies heavily upon inclusion/absence of source code patterns, it is important to gather enough evidence for confident prediction of tag support in a given configuration; otherwise, flawed learning approximations may lead to flawed support knowledge and inaccurate analysis results. Information solicitation was integrated as an update strategy to mitigate the effect of imprecise support knowledge derived as a result of automated knowledge base updates.

The information solicitation strategy essentially retrieves a set of test cases, called a *focus set*, each containing a particular HTML tag or CSS rule, and distributes them to users with specific configuration settings. The basis of this strategy is retrieving more evidence of tag/rule support (or lack thereof) in a given configuration. IS contributors are responsible for investigating the expected contribution of the tag/rule in the web application[2], loading each web page in the focus set, observing the true impact of the specific tag/rule in an assigned configuration, and updating the knowledge base accordingly if they experience a lack of support. In the case of the <`div`> tag, the IS contributor determines the tag to be supported in the given configuration. As a result, they use the manual update approach to eliminate it from the knowledge base as an unsupported tag (Figure 5.7). After discovering that the <`div`> tag is actually supported in the corresponding environment, the user deletes the erroneous criterion from the knowledge base. Given the basics of the experience and skill set needed to complete these tasks, IS contributors must be familiar enough with HTML/CSS and browsing configurations to investigate whether a particular tag is supported/unsupported.

Once the knowledge base has been derived as a result of any combination of manual, automated, and hybrid approaches, web portability analysis can begin with `processURL()`.

[2]This may entail locating a website with an explicit overview of an HTML tag/CSS rule and how it can be used in web application source code.

5.1.3 `processURL()`

Given the static nature of this approach, web application source code is a key factor in analysis. The basic role of `processURL()` is to recover and condition this data. A schematic of this process is shown in Figure 5.8. It is important to note that web applications are regarded as a group of interlinked web pages; to ensure that the full web application is analyzed, a web crawler successively follows hyperlinks, starting at a root URL provided by the developer, and fetches each web page contained in the web application. For each web page retrieved, the raw source code is transformed into a model that the `query()` interface is capable of analyzing. In this instantiation of the general framework, the web application model is generated by assigning a vector to each individual web page recovered; each vector is instantiated by parsing source code, extracting HTML tags and CSS rules, and storing each unique source code pattern as an entry in the corresponding vector.

5.1.4 `query()`

The `query()` interface is most responsible for web configuration fault detection; in short, it is the analysis engine that implements the portability analysis strategy. In the current instantiation, this entails using web application and configuration support models to preform static analysis. In practice, the `query()` interface accepts source code models for a series of web pages as input, sequentially accesses each vector entry (*i.e.*, HTML tag or CSS rule) in the source code model, and uses each retrieved source code pattern as a query to the *knowledge base* (Figure 5.9). To preform analysis, the `query()` retrieves the vector model of the web application generated by `processURL()`. Next, `query()` steps through each vector entry and uses it as a *query* to the knowledge base. If the knowledge base contains the tag, this signals an overlap between unsupported HTML tags/CSS rules and web application source code. An account of intersections is then forwarded to `generateReport()`.

5.1.5 `generateReport()`

As `query()` performs analysis, three key pieces of information are retained: web pages that contain unsupported source code fragments, the client configuration(s) that have support violations, and the unsupported HTML/CSS. In the context of generic framework components, web developers gain access to this information and insight into configuration faults discovered during analysis by way of `generateReport()`. One factor that is expected to have a significant impact on the usability of analysis results is the presentation; namely, whether the format enables a quick overview of results and helps developers prioritize the correction of web configuration faults discovered. Depending on the number of support violations detected, a text-based list (as used in the initial work) could be overwhelming to process. Recognizing that a purely text-based presentation of this information does not adequately fulfill these requirements, visualization techniques have been investigated and a preliminary approach has been derived that provides a concise overview of analysis results and a basis for efficient data interpretation (Figure 5.10). This instantiation of `generateReport()` is expected to effectively reduce data analysis from the fairly taxing cognitive task of processing verbose textual data to the less intensive task of analyzing visual cues; quick, effective code modifications are expected to be the ultimate benefit of encoding analysis results in this visualized format, making it more practical for use in tight web development schedules.

Two coordinated modes of result presentation are featured in the current instantiation of `generateReport()`. In the right-most pane of the interface (as shown in Figure 5.10) web pages in a web application are listed hierarchically. In the corresponding tree, the web page corresponding to the root URL is listed as the root of the tree on the first tier; each subsequent page appears in the hierarchy at the level from which it can be reached from the home page. If it can be reached directly, it appears on the second level and so on.

The purpose of the left-most presentation mode is to provide a high-level overview of support violations detected. To accomplish this goal, web pages are portrayed as points on a two-dimensional plane; the position of the point is a function of the number of unsupported source code elements in the page and the number of client configurations in which the page will be faulty. This presentation is expected to support fault correction prioritization by indicating, for instance, whether a given page will be diminished in many browsing environments or just one extremely obscure environment. When data points in the plane are clicked with a mouse, the corresponding web page is highlighted in the hierarchal web application overview shown in the upper-right pane and the lower-right pane lists the configurations in which the web page is expected to fail and a more detailed analysis of the particular tags expected to fail in those configurations.

5.2 Machine Learning Knowledge Base Updates

Knowledge of how software behaves in the field has long been considered a valuable resource in quality assurance [15, 23, 31, 34]. That principle has been adopted in this thesis as well. From this perspective, as users navigate the web, their encounters with faulty web pages are a gateway to the kind of data used in the outlined approach; their experiences could be used to discover HTML/CSS support structure in various environments directly from knowledge of web pages that that did and did not work in the field. With these factors in mind, the automated subcase of `updateKB()` is primarily responsible for accepting user-provided examples of positive and negative web pages and applying a machine learning algorithm to convert the raw data into knowledge about the influence of individual source code elements on web application behavior and functionality. Thus, discovering the link between source code inclusion patterns and configuration faults from real world examples is a useful way to factor user experiences into the predictive model.

Automated acquisition is expected to be a practical, powerful approach to accumulating tag support knowledge because the criteria may not be intuitive or well documented but it becomes evident when when web applications are activated in the field. In light of these factors, supervised machine learning has been implemented as a subcase of `updateKB()`. The basis for this approach was introduced in Section

5.1.2.2. In this section a more detailed discussion is provided about how empirical data is collected, modeled, and used to isolate relevant source code patterns; an investigation of how variation in each of these aspects of the automated knowledge acquisition strategy affect the quality of knowledge derived follows in Section 5.3.

5.2.1 Data Retrieval

Ideally, as web users stumble upon anomalous behavior or functionality in the field, they would be able to manually isolate the cause of faults by accessing the source, incrementally eliminating suspicious source code statements, and reloading the web page to observe whether the fault persists. In practice, however, average web users may not know to access source code, may not be familiar with basic code constructs, and probably would not know how to update the knowledge base manually as required. Even when users have the know-how to incrementally narrow the space of source code fragments to isolate the cause, this process becomes time-consuming and tedious when the search space is large and complex.

The goal of machine learning, from this vantage point, is to retrieve and analyze source code from fielded instances and automate the isolation of code patterns that influence anomalous outcomes; this will essentially minimize human intervention so that no expertise is required, just access to submission engine and examples. A key issue to consider is how this data will be collected for analysis. In the current work, users submit examples as a tuple $\langle \mathbf{S}, c, l \rangle$, where $\mathbf{S}=\{s_1, s_2, s_3, ..., s_n\}$ is the source code (which is comprised of HTML/CSS), $c \in \mathbf{C}$ is the configuration (described by a conjunction of components/settings), and $l=\{true, false\}$ is a binary value that indicates whether the user labeled the example as a positive or negative instance; l is set to *true* if the user considers it to be a positive example.

The components of this tuple $\langle \mathbf{S}, c, l \rangle$ are key in the supervised machine learning approach; Since learning takes place locally in each configuration, c specifies the configuration node that the source code **S** should be routed to and whether the example is a positive or negative instance, l. Given the open-source nature of web applications, web source code has a relatively high availability, making retrieval of **S** straightforward. Yet, as mentioned before, users may not know how to access this data. In response to this issue, the URL of the faulty example is currently used to fetch the corresponding source code. Subsequently, the role of the user is to simply provide the URL of the web page they have observed, specify the make-up of their client configuration (*i.e.*, the browser and version used, network speed, font size, screen resolution, etc.)[3], and characterize the example as positive or negative, l.

To summarize the machine learning approach in the context of the tuple, the components of **S** and the boolean value of l are used to detect fault-relevant source code inclusion patterns for configuration c given positive and negative web application models as evidence. In the remainder of this section, more insight into **S** is provided, namely how it is retrieved and processed for learning; a discussion of learning strategy details follows along with and overview of how **S** updates c given l.

[3]One goal would be to retrieve this data through an automated means. This would further reduce the burden of the users.

5.2.2 Web Application Model

In this work, web applications are modeled as a set of interconnected web pages and the base model for each web page is the set of HTML/CSS source code fragments. In the context of learning, each individual source code fragment is considered a feature that influences whether the web page is a positive or negative example. Much like the model used during portability analysis, the web application model used during automated knowledge acquisition designates a vector for each web page and instantiates the vector with conditioned source code. More specifically, the web application model used for learning is generated by extracting HTML tags/CSS rules from raw source code, conditioning the data, and storing each atomic fragment recovered as an entry in the vector. The raw source code used as the basis of the model is very accessible and inexpensive to retrieve given the open-source nature of the web.

Once HTML/CSS is extracted from raw source code, data conditioning is performed as a normalizing step before source code fragments become components in the vector model. This process is necessary mainly because tags/rules are comprised of various parts and they can be formatted quite differently in practice. Before investigating the issue of source code representation in web application models, consider the basic conditioning that must be performed on the data prior to learning. In terms of HTML tags, one important data conditioning step is condensing tags into atomic entities. To begin the discussion of the need for this process, consider the fact that HTML tags are generally comprised of a tag name, attribute, and attribute value. In the example

$$< \overbrace{\texttt{H1}}^{\text{tag}}\ \underbrace{\texttt{ALIGN}}_{\text{attribute}} = \overbrace{\texttt{center}}^{\text{attribute value}} >$$

`H1` is the tag name, `ALIGN` is the attribute, and `center` is the attribute value. In this particular instance, there is only one attribute/attribute value pair contained in the tag; in practice some tags have several attributes. For example, consider the tag:

`<body bgcolor=‘‘#FFFFFF" text=‘‘#000000">`

In such cases, the tag is tokenized into an atomic entity in which each <*tag, attribute, attribute value*> triplet is represented individually. More specifically, the corresponding tag becomes:

`<body bgcolor="#FFFFFF">`,

`<body text="#000000">`.

Once HTML tags are properly atomized, the challenge regarding the extent to which tag attributes and attribute values are represented in the model must be addressed. Consider that there are two extremes in addressing this challenge: either all of the attribute values will be represented or all of the attribute values will be abstracted. Both approaches have disadvantages; in the prior, the knowledge base will be quite large if each attribute value is retained (such as cases where the attribute value is a number) presenting a challenge to maintaining an efficient analysis. In the latter case, the knowledge base will be much smaller, but the accuracy of the analysis may be compromised by the loss of precision. Subsequently, a trade-off exists between the breadth of information considered (perhaps to improve efficiency) and the correctness of the resulting model.

Because the extreme of including all of the attributes is intuitively inefficient, the three remaining tag abstraction strategies will be evaluated later in the experiments (Section 5.3). Named M_1, M_2, and M_3 the strategies are as follows:

- M_1: all attributes and attribute values are filtered.
- M_2: all attribute values are filtered.
- M_3: only predefined attribute values are filtered[4].

The latter abstraction strategy is slightly more intelligent than the others mainly because it includes attribute values, some of which may have configuration-dependent constraints, to be represented and analyzed.

In the previous version of the framework, M_3 was the only filtering strategy considered and data extraction/conditioning was only performed on HTML tags. Since CSS rules must also be processed during learning, however, data conditioning is also necessary. The discussion of CSS rule conditioning begins with the knowledge that rules have a selector that specifies the document elements to which the rule applies, and declarations that specify the stylistic effect of the rule. The declaration is a set of property/value pairs. In the example

$$\overbrace{\texttt{H1}}^{\text{selector}}\ \underbrace{\{\texttt{font}-\texttt{size}}_{\text{property}}:\ \overbrace{\texttt{13pt}\}}^{\text{property value}}$$

`H1` is the selector, `font-size` is the property and `13 pt` is the value. In conditioning CSS rules, the thesis approach was to disregard the selector and mainly

[4]

concentrate on the property and property values. In this particular phase of the work, however, only a variant of the M_2 scheme to rule abstraction was applied, where all property values were abstracted.

In summary, the extraction and conditioning of source code elements provide the basis for learning as input for the learning strategy used. The use of these strategies with practical examples will be explored in a later section; in the next section, an overview of the learning strategies employed will be provided.

5.2.3 Learning Strategies

The main foundation of this work is deriving knowledge of tag support from fielded web applications. In this context, the HTML tags found in both faulty and correct web pages are the learning units; by noting their membership in positive/negative examples, the idea is to learn whether they are supported or not. To address this issue, techniques originally applied to text categorization have been employed; this is a highly analogous endeavor because in the latter domain, one train of thought is to analyze text documents and identify keywords that help to distinguish documents that belong to a category from ones that do not. In this work, source code inclusion patterns are the features of interest and the categories are correct or incorrect.

Many techniques have been applied in text categorization such as Bayesian Networks, decision trees, neural networks, support vector machines, k-nearest neighbor approach, etc. In choosing a learning strategy, one key goal was to recover the learning results in a form that could be easily read and comprehended by a human observer. An additional criteria was efficient learning phases and a resulting predictive model that could be easily interpreted and tuned by humans. In light of these requirements, featuring scoring measures used in feature subset selection have been used to measure influence a tag and elements have on achieving portability.

In each of the measures implemented to date, both use raw counts of source code inclusion/omission from a positive/negative example of web application functionality/execution. The first one, called the correlation coefficient, L_1, is a modified measure adapted from Yang and Penderson [55] and measures the strength of correlation between a feature and a category. The odds ratio, L_2, is used heavily in etiology to discover risk factors strongly correlated with a condition.

$$L_1(t, c) = \frac{\sqrt{N} \times (AD - CB)}{\sqrt{(A + C) \times (B + D) \times (A + B) \times (C + D)}} \tag{5.1}$$

$$L_2(t, c) = \frac{B \times D}{A \times C} \tag{5.2}$$

The raw counts used to compute both L_1 and L_2 are shown in the 2×2 contingency table (Table 5.1). Note, N, is the total number of examples, A is the number of positive examples that contain a given tag/rule, B is the number of positive examples that do not contain the source code fragment, C is the number of negative examples that contain the tag/rule, and D is the number of negative examples that do not contain the tag/rule.

Since machine learning algorithms do not generate perfect models, L_1 and L_2

		Web Application Status	
		Correct	Faulty
Tag Status:	Present	A	C
	Absent	B	D

Table 5.1: Contingency table illustrating the four possible states of tag/category co-occurence

will be evaluated in a latter section to measure the accuracy of models derived from each.

5.2.4 Data Storage

In the current approach, learning takes place locally at each configuration node. Using the notation presented earlier, the configuration ,c, that users indicate when submitting an example specifies the configuration node that the source code **S** should be routed to and whether the example is a positive or negative instance, l. The client configuration model in this paper is significantly different from the one utilized in our previous work largely because of expanded CSS rule inclusion and consideration toward the effects of tag interaction on configuration faults during analysis. In this version a matrix maintains an account of tags/rules retrieved from positive and negative examples as both its rows and columns. As shown in Figure 5.11, the interaction matrix is an $n \times (n + 1)$ structure in which the first column of the matrix represents the strength of association with the tag to negative web applications as indicated by either L_1 or L_2. The locations where columns and rows meet contain a value that indicates the affinity of two tags to negative categorization of web applications. When there are a pair of tags in which (1) one is expected to be unsupported and the other is expected to be supported and (2) the value associated with the co-occurrence is within the valid threshold, the supported tag is expected to be a positive offset of the negative tag in the corresponding configuration. Please note, as source code models, $\mathbf{S}$, are routed to the corresponding configuration defined in the tuple, the code fragments they contain and the value of l determine how A, B, C, and D are updated.

5.3 Research Questions and Metrics

A major goal of the current research is to improve the effectiveness of the web application portability approach. As mentioned earlier, one key aspect of thus approach is the distinct relationship between the accuracy of support criteria used during analysis and the applicability of portability analysis results. If the knowledge base does not adequately capture the actual support of HTML tags and CSS rules across the vast configuration space, at worst, configuration faults will remain latent after analysis and web developers will have a false confidence in the portability of their web applications. To enable a comprehensive knowledge store, three support criteria acquisition methods have been integrated. In each of the experiments discussed, the major focus is the automated acquisition strategy applied to this problem and exploring how aspects such as the modeling of tags and rules during learning, the learning technique applied, and training set size and composition affect the accuracy of support criteria acquired using this method. Consideration has also been given to how the web application model affects cost, how well the learning techniques detect interactions, and how inclusion of CSS in the model can affect analysis. In the sections that follow, insight is provided into the research questions posed and the experimental setup. Details of the investigations and the results are provided as well.

5.3.1 Research Questions

Having outlined the nature and motivation of the experimental goals, the list of questions that will be addressed, for both clarity and later reference, is as follows:

Q1: What effect does web application model, learning strategy, and training set size have on whether tags or rules are properly classified as supported or unsupported?

Q2: How does the web application model affect analysis costs in terms of tags/rules evaluated and the time needed for analysis?

Q3: How does the ratio of positive examples to negative examples affect classification accuracy?

Q4: How does the inclusion of CSS rules in the web application model affect the learning process?

Q5: How well is tag interaction captured using different learning strategies?

5.3.2 Configuration Subject and Data

In a previous study [17] the false positive rate of learning strategy L_1 was evaluated for 16 different client configurations. In this study, more learning strategies and web application models are used and, in the presentation of results, the focus is concentrated on a sample client configuration in which Opera 6.0 is the browser and browser version that operates on Windows XP with javascript enabled. It is important to note that results of learning process are more a function of the training set and less a result of client configuration attributes. In effect, the client configuration and the positive and negative web page instances provide a realistic basis for evaluating the learning strategy and place a realistic context on how aspects such as web application modeling and training set affect accuracy.

A driving factor in automated support criteria acquisition is submission of positive and negative examples. To support the research effort, 100 positive examples and 100 negative examples were collected for the corresponding client configuration. This data was accumulated in two phases: passive and active. In the passive phase, random web pages were loaded in the subject client configuration and judged whether they were a positive or negative example by co-loading it in a more popular browser. Negative pages went into one folder and positive examples went into another. For the most part, this phase accumulated more positive examples than negative ones. In the active phase, the Google search engine was used to locate pages that incorporated specific fault-inducing tags/rules. Because Google ignores the brackets and parenthesis inherent in HTML/CSS, retrieval of web pages with desired source code fragments was a challenge. Query returns included web pages that actually included the tag/rule in source code to those that merely mentioned the tag/rule or, if the tag/rule was a common name, the return set was even more inflated; because there was no feasible way to narrow returns to those of interest when the query was issued, identifying returns that were actually useful was tedious at best.

5.3.3 Evaluation Metrics

The costs involved in employing a tool or technique can ultimately have a significant impact on subsequent feasibility and usefulness. Within the context of this work, the cost of the learning strategy is influenced by the amount of computational effort required to label source code fragments as either supported or unsupported and the penalties resulting from source code fragment (tag/rule) misclassification. Given the nature of the learning algorithms used, the computational effort required is expected to be reasonable and to depend largely on the amount and model of data being processed. On the other hand, misclassification of source code fragments is expected to have a large impact on the cost model. To get an understanding of source code misclassification during learning, consider the fact that during experimentation, learning strategies will be applied to positive/negative examples. The strategies will then predict whether a given tag/rule is supported or unsupported given inclusion patterns detected in the examples. In the following sections, a discussion how the actual and predicted support of tags/rules are computed is provided along with how they are used to quantify how well a learning algorithm performs in terms of accuracy.

5.3.3.1 Actual Support

In this study, the ground truth, or gold standard, maintains an account of the actual support provided for a particular tag in a given client configuration. The ground truth was manually defined using a resource that specializes in cross-configuration tag/rule support data [46]. For the sake of discussion, consider the gold standard to be a function, GS, that takes a source code fragment, t and a configuration, c, as input and returns a boolean value that indicates support or lack thereof as output. A more formal definition is provided below in Equation 5.3.

$$GS(t, c_x) = \begin{cases} yes & \text{if } c_x \text{ actually supports } t \\ no & \text{otherwise} \end{cases} \tag{5.3}$$

5.3.3.2 Predicted Support

Support is predicted with the help of the learning mechanism. In this work, the learning algorithm has been abstracted as a function, AR,that accepts a source code fragment, t and a configuration, c as input and returns a boolean value that predicts whether the fragment is supported or unsupported. Like GS, AR can be modeled as shown below:

$$AR(t, c_x) = \begin{cases} yes & \text{if } c_x \text{ predicted to support } t \\ no & \text{otherwise} \end{cases} \tag{5.4}$$

5.3.3.3 Accuracy

In this work, *accuracy* is used to measure how often the learning algorithm made correct and incorrect predictions of tag/rule support. As it applies to the learning process, *negative* classification of a source code fragment is considred to mean that it is not supported in a given environment and *positive* classification indicates that it is supported. Subsequently, a false negative is a supported tag labeled as unsupported and a false positive is an unsupported tag labeled incorrectly as supported. To examine how TP, FP, TN, FN values are calculated in terms of AR and GS, consider Table 5.12 below.

The effects of misclassification on cost, in terms of portability analysis, are largely a result of false negatives and positives. A direct consequence of a false positive is that a faulty page could unwittingly be released into the field. On the other hand, the threat of false negatives lies in the fact that they erroneously signal a need for manual effort. Consequently, they influence an increase in the cost of resources devoted to addressing what is, in actuality, a non-issue. With these factors in mind, accuracy is defined in a way that will penalize processes that allow for more false positives.

$$accuracy = A(M_y, L_z, size) = \frac{TP + TN}{TP + FP + TN + (FN * 0.5)} \tag{5.5}$$

As defined here, higher accuracy values mean fewer false positives and false negatives than those with lower values. The accuracyfunction is paramaterized by the source code model used M_y (where y = {1,2,or 3}), the automated acquisition strategy L_z (where z = {1 or 2}), and the size of the training set, *size*. Note in the equation above, false negatives have been weighted. This step was taken because false negatives do not levy the same costs on the analysis results as do false positives. The former is a nuisance in that developers may spend time attempting to fix web configuration faults that do not exist. The latter, on the other hand, could cause web developers to miss crucial inclusion patterns.

5.4 Study Design, Results, and Discussion

Empirical investigations of **Q1** through **Q5** were carried out to gain a better understanding of learning parameters and their impact on the accuracy of support criteria learned. The results of these studies are expected to improve automated knowledge acquisition by looking at the individual models and algorithms that will produce the most accurate results. In the remainder of this section, insight into the experimental process used to investigate each research question, the results we observed, and how we interpreted them is provided.

5.4.1 Q1 Study: The effect of web application model, strategy, and training set size on learning accuracy

The goal of this study was to quantify how the learning strategy and attributes of the training set in terms of quantity and source code model affect the accuracy of support criteria derived. The overall basis is to ensure that extraction and representation of data and learning techniques provide the proper basis for knowledge discovery. Given the inductive nature of the approach, data representation, namely how raw source code model is conditioned prior to learning, can have a significant impact on the quality of knowledge derived.

5.4.1.1 Experimental Procedure

The following process was used to explore Q1:

Step 1: Retrieve an initial pool, P_{c_x}, of positive and negative web pages for client configuration c_x.

Step 2: Parse HTML/CSS to extract source code using each of three abstraction techniques (M_1,M_2, and M_3). This will produce P_{c_x,M_1}, P_{c_x,M_2}, and P_{c_x,M_3}.

Step 3: Retrieve the gold standard of tag support rules for later evaluation.

Step 4: For each abstraction strategy (M_1,M_2, and M_3), do the following 100 times:

1. Randomly select 15 positive and 15 negative web pages from P_{c_x}.

   (a) Extract source code elements and estimate whether the corresponding tags/rules discovered are strongly correlated with negative examples by applying learning strategies L_1 and L_2.

   (b) Calculate $A_{M_y,L_z,size}$ where $1 \leq y \leq 3$, $1 \leq z \leq 2$, and $size$ =30.

   (c) Note the co-occurrence of tags/rules in the sample and estimate interaction factors by calculating the corresponding L_1 and L_2.

2. For six iterations, without replacement, randomly select 15 positive examples and 15 negative examples from the pool until it has been exhausted.

   (a) Gather new evidence for the tags/rules discovered in step 1(a) and update estimations of negative example correlations for both L_1 and L_2.

   (b) Update $A_{M_y,L_z,size}$ where $1 \leq y \leq 3$, $1 \leq z \leq 2$, and $size$ =30×iteration.

   (c) Note the co-occurrence of tags/rules in the sample and update interaction estimation by calculating L_1 and L_2.

5.4.1.2 Results

It appears in Figure 5.13 that M1 is the best web application model, L1 is the best learning strategy, and that learning accuracy generally improves as the training set size increases. While M1 may provide the most accurate model for automated acquisition, it is important that more information be included in the knowledge base. Tag support is generally favorable at the tag level alone. As a result, it is very important that details, such as attribute, their values, and subsequent support are accounted for. In that regard, it may be best to get this data from manual updates and information solicitation.

5.4.2 Q2 Study: How does the web application model affect analysis costs in terms of tags/rules evaluated and the time needed for analysis?

To evaluate the affect of the model on time for analysis and the number of tags analyzed, we monitored each during the study of **Q1**. Our results are shown in Figure 5.14. Since M1 provides the most abstraction, it follows quite naturally that it requires the least amount of time to evaluate and that less tags are analyzed per training set size.

5.4.3 Q3 Study: The effect of training set imbalance on false positives.

In our previous work, we observed that the median false positive rate was much lower when there was an extra negative example. To observe what happens when an extra positive example is included, we essentially preformed the same steps carried out for Q1 with an uneven set of training examples. In this case however, we only evaluated the false positive rate for M3 in order to maintain the same conditions we used in our previous study, save the inclusion of an extra positive example instead of a negative example. The results are shown below in Figure 5.15. As shown, more negative examples provide a basis for lower false positive rates. Recall, each instance contains a number of tags/rule and serve, as a whole, as evidence of the support or lack thereof for each source code fragment contained. Subsequently, with an extra negative example, the corresponding tags get more supporting evidence. As a result, there is less chance that a truly unsupported tag will be labeled as supported.

5.4.4 Q4 Study: The impact of CSS inclusion during the learning process

Our evaluation of the impact of CSS inclusion was carried out by determining the number of unique CSS rules recovered during learning. This is an important factor because one of our main goals is to include as much support criteria in the knowledge as possible to increase the accuracy of web configuration fault analysis. We observed 39 unique CSS elements in the source code; for reasons stated above, we expect the addition of this information to forge a positive step towards achieving a comprehensive knowledge base.

5.4.5 Q5 Study: The impact of Tag Interaction during the learning process

In order to capture tag interaction, we applied L_1 and L_2 to the data yet, instead of checking to see whether a single tag or rule appeared in the source, we checked for the times that tags occurred jointly in positive and negative examples. To observe how well the learning strategies performed, we had the algorithm return a list of tags expected to have joint impact on the correctness of web pages. In its current form, correct interactions are detected, however, spurious/non-interacting returns are provided as well. This, again, speaks to the issue of false positives discussed earlier.

5.4.6 Threats to Experimental Validity

5.4.6.1 Internal Validity

The internal validity of experimental results are threatened when results of the dependent variable can be tainted by modeling and measurement errors. In each of the questions we address, accuracy is the primary dependent variable. Hence threats to internal validity, in this context, include possible errors in measuring/designating the training set and modeling/executing both the tag abstraction scheme and tag classification strategy.

Another threat lies in the correctness of the gold standard. The source used as the basis for the gold standard, in some instances, relies on the documentation provided from the browser manufacturer. Since this can be erroneous at times, it can have an undesirable impact on accuracy evaluations.

One final internal validity threat lies in, what amounts to, equal weighting for false positives and false negative in our accuracy model. More specifically, in the case that one is actually more important than the other, the equation should be weighted accordingly in order to derive correct values.

5.4.6.2 External Validity

Threats to external validity, on the other hand, limit ability to generalize experimental results. Several candidates for this constraint apply. For one, we are currently only considering pages in which there are source code-induced faults that can be linked to a certain tag and not, perhaps, Javascript errors that can be linked to a faulty variable. Other threats include possible misclassification of web pages on the behalf of submitters and low usage of a given client configuration platform (resulting in less raw material for the inductive algorithm). We took a great deal of care to ensure that pages were accurately labeled and included a sizable number of positive/negative examples during analysis; these factors may or may not be sustained in the field.

Chapter 6

Conclusions and Future Work

Essentially all deployed software systems have bugs [31]; the extent to which bugs are detected and corrected has a significant influence on software quality. Although portability assessment is crucial for a wide range of modern software systems, it is a particular challenge for web application development. The ability to choose from a varied set of operating systems, browsers, browser versions, hardware, and customize browser settings provides users with expanded flexibility in how they access the web. Yet, from another perspective, this expanded flexibility invokes a need for web developers to not only ensure they their web applications are correct but that correctness persists as the web application is ported.

This dissertation outlines a model-based framework that enables automated detection and diagnosis of web configuration faults. The basic idea of this approach is that unsupported source code patterns (*i.e.*, HTML tags and CSS rules) are indices to potential configuration faults; when support is lacking, the aesthetic or functional properties associated with source code may be lost and configuration faults may result. This approach overcomes the limitations of existing approaches by enabling efficient coverage of the configuration space and linking the faults discovered to unsupported source code. In conducting this research, several interesting questions were discovered. In short discovering effective algorithms and methods and integrating them into the framework is expected to be an continuous process. Tag/rule representation, learning strategy, and tag interaction can each impact knowledge acquisition as well as subsequent portability analysis. Moreover, client configuration space representation models and strategies for visualization of portability threat detection results must constantly be refined to improve fault detection efficiency and to facilitate correction of isolated faults. In the effort to maximize the efficiency of the framework and make more practical and effective in tight web development schedules, several immediate and future goals have been established:

1. **Learning strategies**: As noted earlier, the effectiveness of the configuration fault analysis developed as a result of this project is heavily reliant upon the completeness and accuracy of the tag/rule support knowledge used for analysis. Adequate population of the tag support knowledge base is one of the most

important aspects of the framework since attempting to detect portability issues with truncated or inaccurate knowledge can severely inhibit accuracy. In the immediate future, the idea would be to continue the search for other strategies, to modify them (if necessary) to address this particular problem, and to compare the performance of each in order to determine the most appropriate approach.

2. **Modeling Paradigms**: In the short-term future of this research, web application and configuration models will be incrementally modified and exercised. At the beginning of this work, simple web application examples were used. As the work progresses, the plan is to continue up to the most complex examples, incrementally improving the models and learning approaches along the way in order to account for features such as tag interaction and the effect of nesting on portability.

3. **Web user subjectivity**: Another issue to explore would be web user subjectivity in distinguishing positive examples from negative ones. The issue here is that web users with the same configurations can load the same web page and, in some cases, have different opinions about whether the page is a negative or positive instance. This is an important issue largely because the way in which web applications are classified (positive or negative) has a direct influence on the support knowledge derived. In future work, the goal could be to study this issue with actual web users and to ensure that the learning strategy used can recover in the face of noisy, misclassified, input.

4. **Human-Computer Interaction (HCI) Issues**: One key objective is further exploration of the HCI aspects of the framework including the usability of the implemented tool for both users and web developers. In alignment with this goal, four user studies have been defined to measure tool support for updating the knowledge base and interpreting analysis results. Further discussion of each follows including the goal of the study and how it will be evaluated in terms of a *strawman*, or alternative, approach:

   - *Manual Knowledge Base Updates*: Recall, support criteria experts manually update the knowledge base through a *Criteria Editor*. It is important to ensure that the *Criteria Editor* is designed to collect the appropriate data, does not influence mistakes, and helps to minimize user error. Given that the alternative to the editor would be direct access to the knowledge base, one user study would involve comparing error rates when users enter data into the knowledge base directly and when they use the *Criteria Editor*. Another would compare manual updates from users with varying levels of expertise to determine the particular types of errors made so that the appropriate support mechanisms can be implemented (*i.e.*, constraint checkers). Yet another issue with manual knowledge base updates is that the current design *trusts* experts to import correct data. Future research could include assigning security ratings to experts and adjusting them over time and, perhaps, contexts (*i.e.*, experts may be trusted more when submitting information for one configuration vs. another).

Usability is an issue here because the next step would be conveying this information when analysis results are presented; users should be aware when there is low confidence that a configuration actually lacks support based on the source of the data. In this case, the alternative approach would be that no confidence measure is conveyed to the user; the study would involve observing how analysis results are interpreted in both cases and the impact of those interpretations on prioritizing the correction of configuration faults.

- *Automated Knowledge Base Updates*: Web users automate knowledge base updates by submitting positive and negative examples of web pages. To improve the usability of this update mechanism, it is important to determine the best way to collect this data and to carefully observe how users distinguish positive examples from negative ones. In terms of identifying the best data collection method, one early prototype is a web browser tool bar that has a *green* and a *red* icon that allows users to submit positive examples by clicking on the green icon and to submit negative examples by clicking on the red icon. A slight alternative to this approach is only featuring a red icon on the tool bar since the intuition is that users are more apt to indicate when a page is faulty then when it is correct. A more drastic alternative is that users would have to submit an e-mail detailing the configuration they were using and the URL of the positive/negative web page. The accompanying user study would measure the number of positive and negative web pages submitted with each strategy; in addition, by allowing users to exercise at least two of the three strategies, the idea would be to understand user preference.

- *Solicited Knowledge Base Updates*: In solicited updates, web users with precise configuration settings help to determine the support for a given tag/rule by loading web pages that contain the source in their browsing environments and observing the results. The key factor to measure, from a usability perspective, is the best way to distribute sample pages to users and collect the resulting data. Prototype strategies range from manual acquisition and deployment of the *focus set*[1] in which users access a central database that contains sample pages, downloads the corresponding focus set, and launches the web applications one by one, to fully automated distribution and deployment in which the tool retrieves the focus set and automatically launches them in a browser with red and green icons (as discussed in the manual update). The role of the user in the latter case would be to simply observe the web page and click the green button if the tag/rule appeared to be supported and the red button otherwise. To measure the effect of each strategy, users will have the opportunity to use at least two alternatives and the number of correctly identified positives/negatives would be evaluated along with the time necessary to submit the data and user preference.

[1]A set of web pages that contain a given tag/rule

  - *Presentation of Analysis Results*: Currently, web developers get a post-analysis view of the results in which each individual web page is represented as a point in a two-dimensional plane and the placement of the point is determined by the number of tag/rule support violations discovered (x-axis) and the number of configurations with support issues (y-axis). Future usability studies may involve varying the values represented on the x and y axis and employing *think aloud* techniques to observe how prioritization of fault detection is modified. More specifically, given specific instructions to fix a set amount of web pages in a limited amount of time, the idea would be to observe how developers chose which web pages to correct first and why. An alternate strategy, a purely text-based list of results, could be evaluated as well under the same conditions, to compare how developers prioritized their efforts.

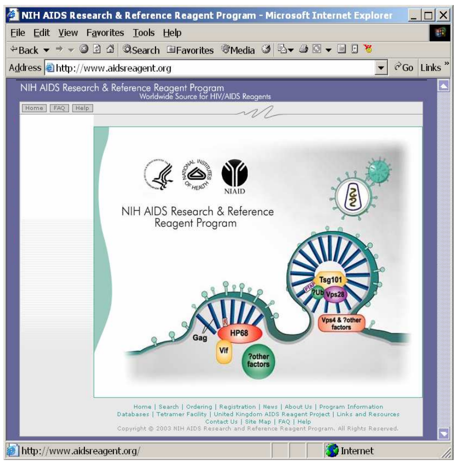
5. **Correcting problems detected**: Currently, the framework can detect a fault and alert web developers of the corresponding configuration. A natural extension of the tool would be to incorporate a mechanism that will provide web developers with a fix for the problem detected. In the future, the idea could be to explore how a repository of fixes can be generated and incorporated into the framework.

6. **Automated detection of positive/negative examples**: Currently, the identification of positive/negative examples for learning support knowledge is a manual process. Identifying automated means for gathering examples and

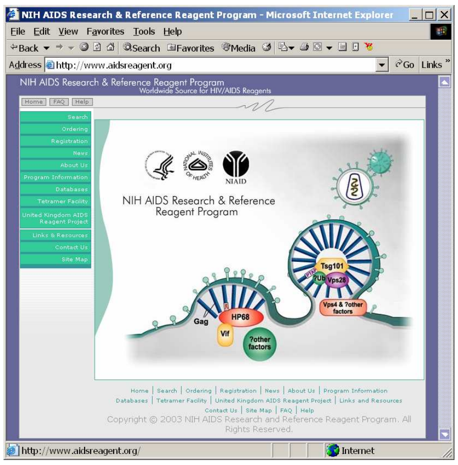
classifying them is expected to improve the quality of knowledge derived by providing the learning algorithm with more evidence for whether a tag/rule is supported in a given environment. A future research direction would be to explore this issue further and develop a method that will automatically gather and classify examples for the learner.

7. **Use of Metadata during Automated Acquisition**: During automated acquisition, the current focus is learning from source code inclusion patterns. To ultimately achieve more accurate support criteria knowledge, it may be helpful to include more metadata in the process. In particular, the time spent on a web page, whether the user simply looked at the page or tried to preform some action, the actual widgets activated, user explanations of the nature of the problem (*i.e.*, the page rendered improperly).

From a very high level perspective, this work speaks to the power of *communication* between the user and development community in evaluating and improving software quality. By allowing users to import support criteria, deriving support criteria from fielded examples, and automating the analysis of an implementation with respect to each, developers benefit from a wide body of knowledge. Most importantly, they do not have to consult individual users or other developers to gather the knowledge and they do not have to resort to any manual means of applying this knowledge during analysis. In addition, deriving knowledge of fault triggers from fielded instances through artificial intelligence is expected to be an effective method for gathering information for analysis when there is an ample number of positive and negative examples. Discovering ways of applying similar means in other software types should provide an effective way for learning how faults in the field can be used for future analysis.

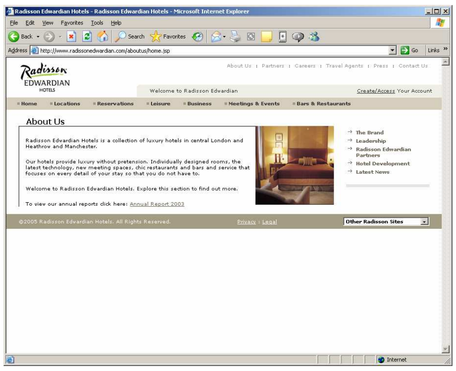

NIH AIDS Research & Reference Reagent Program - Microsoft Internet Explorer
http://www.aidsreagent.org
NIH AIDS Research & Reference Reagent Program
Worldwide Source for HIV/AIDS Reagents
NIH AIDS Research & Reference Reagent Program

No Javascript

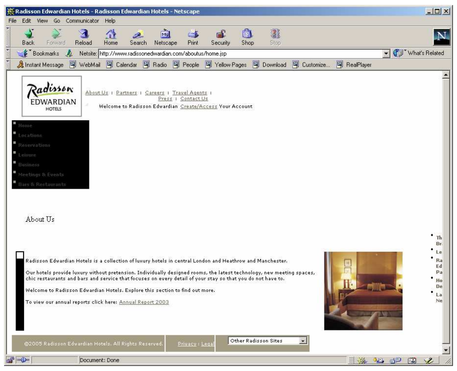

NIH AIDS Research & Reference Reagent Program - Microsoft Internet Explorer
http://www.aidsreagent.org
NIH AIDS Research & Reference Reagent Program
Worldwide Source for HIV/AIDS Reagents
NIH AIDS Research & Reference Reagent Program

With Javascript

`http://www.aidsreagent.org/`

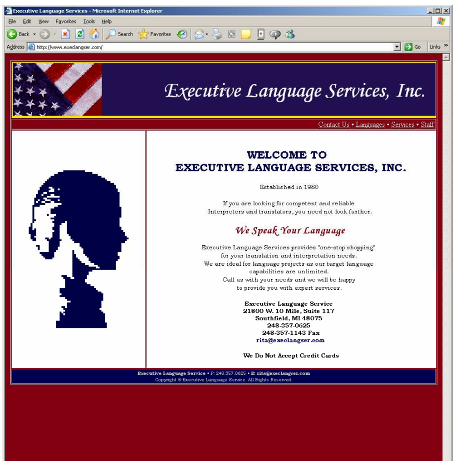

Radisson Edwardian Hotels - Radisson Edwardian Hotels - Microsoft Internet Explorer
Radisson EDWARDIAN HOTELS
About Us

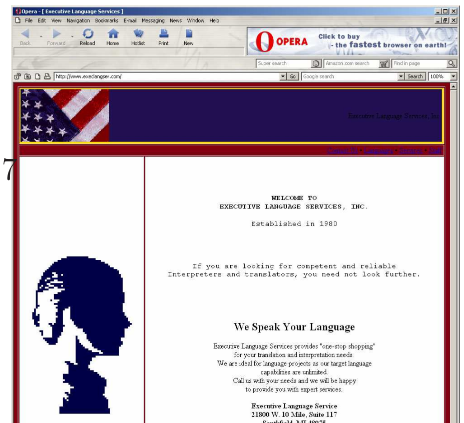
Internet Explorer 6.0

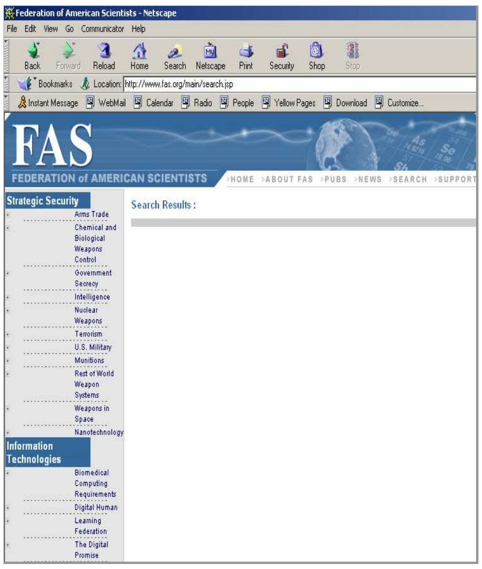

Radisson Edwardian Hotels - Radisson Edwardian Hotels - Netscape
Radisson EDWARDIAN HOTELS
About Us

Netscape 4.8

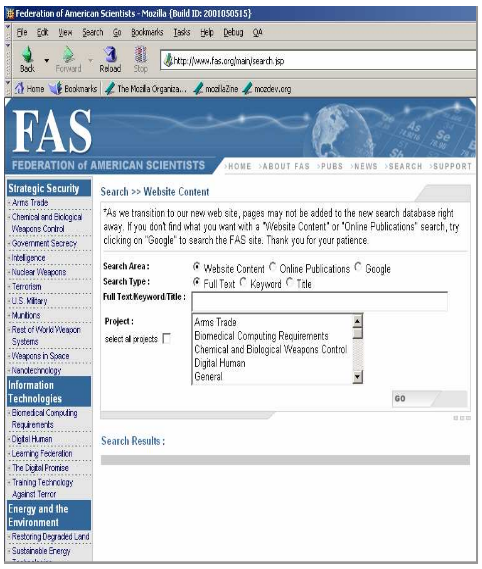
`http://www.radissonedwardian.com/aboutus/home.jsp`


Executive Language Services - Microsoft Internet Explorer
Executive Language Services, Inc.
WELCOME TO EXECUTIVE LANGUAGE SERVICES, INC.
We Speak Your Language


Opera - [Executive Language Services]
WELCOME TO EXECUTIVE LANGUAGE SERVICES, INC.
We Speak Your Language


Federation of American Scientists - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://www.fas.org/main/search.jsp
Instant Message WebMail Calendar Radio People Yellow Pages Download Customize...
FAS
FEDERATION of AMERICAN SCIENTISTS
›HOME ›ABOUT FAS ›PUBS ›NEWS ›SEARCH ›SUPPORT
Strategic Security
Arms Trade
Chemical and Biological Weapons Control
Government Secrecy
Intelligence
Nuclear Weapons
Terrorism
U.S. Military
Munitions
Rest of World Weapon Systems
Weapons in Space
Nanotechnology
Information Technologies
Biomedical Computing Requirements
Digital Human
Learning Federation
The Digital Promise

Search Results :


Federation of American Scientists - Mozilla {Build ID: 2001050515}
File Edit View Search Go Bookmarks Tasks Help Debug QA
Back Forward Reload Stop
http://www.fas.org/main/search.jsp
Home Bookmarks The Mozilla Organiza... mozillaZine mozdev.org
FAS
FEDERATION of AMERICAN SCIENTISTS
›HOME ›ABOUT FAS ›PUBS ›NEWS ›SEARCH ›SUPPORT
Strategic Security
Arms Trade
Chemical and Biological Weapons Control
Government Secrecy
Intelligence
Nuclear Weapons
Terrorism
U.S. Military
Munitions
Rest of World Weapon Systems
Weapons in Space
Nanotechnology
Information Technologies
Biomedical Computing Requirements
Digital Human
Learning Federation
The Digital Promise
Training Technology Against Terror
Energy and the Environment
Restoring Degraded Land
Sustainable Energy
GO

Search >> Website Content

*As we transition to our new web site, pages may not be added to the new search database right away. If you don't find what you want with a "Website Content" or "Online Publications" search, try clicking on "Google" to search the FAS site. Thank you for your patience.

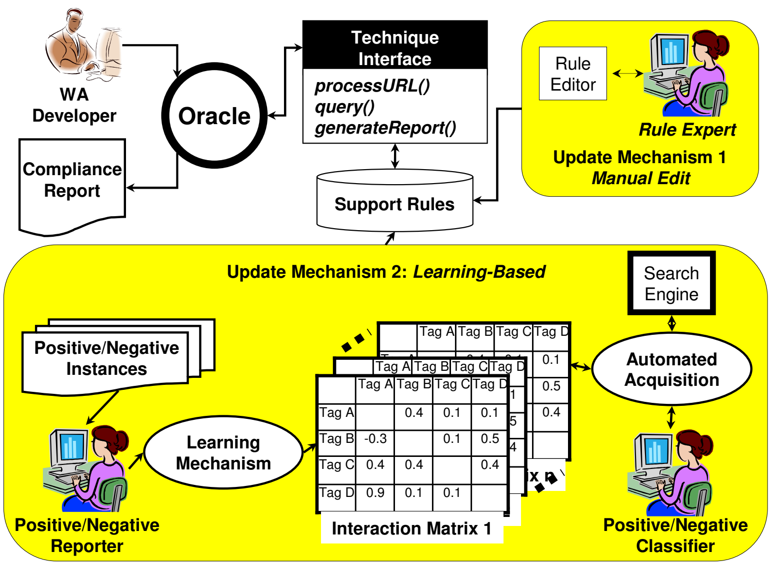
Search Area : [✓ Website Content] Online Publications Google
Search Type : [✓ Full Text] Keyword Title
Full Text/Keyword/Title :
Project : Arms Trade, Biomedical Computing Requirements, Chemical and Biological Weapons Control, Digital Human, General
select all projects ☐

Search Results :

Netscape 4.8 Mozilla

`http://www.fas.org/search`

Figure 4.7: Mozilla is More Forgiving than Netscape when Tags are Misproperly Placed in Source Documents.



	Tag A	Tag B	Tag C	Tag D
Tag A		0.4	0.1	0.1
Tag B	-0.3		0.1	0.5
Tag C	0.4	0.4		0.4
Tag D	0.9	0.1	0.1	

Figure 5.1: Instantiation of the general framework in the current Approach

<tag .* attribute=value.*>

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_A$
$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_B$
....
$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_T$

selector{property:value.*}

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_Q$
$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_R$
....
$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_W$

<tag.* attribute=value.*> ∧ ! <tag.* attribute=value.*>
....

Figure 5.2: A generic representation of the knowledge base.

```
<script language="JavaScript1.2">
        <!--// will only run on any JavaScript1.2+ enabled browser//-->
</script>
<script language="JavaScript1.3">
        <!--// will only run on any JavaScript1.3+ enabled browser//-->

</script>
```

Figure 5.3: A practical example of support violation offsets.

<a accesskey=.*>

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_B$
$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_S$

<script.* javascript=1.4 .*> ∧ ! <script.* javascript=1.3.*>

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_Q$
$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_R$
$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_W$

<layer bgcolor=.*>

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_A$
$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_S$

Figure 5.4: Snapshot of the Knowledge Base after a manual update.

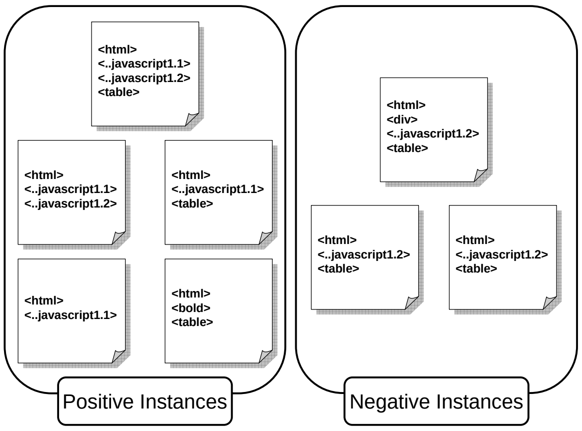


Figure 5.5: Positive and negative web applications in an arbitrary client configuration

<a accesskey=.*>

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_B$

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_S$

<script.* javascript=1.4 .*> ∧ ! <script.* javascript=1.3.*>

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_Q$

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_R$

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_W$

<layer bgcolor=.*>

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_A$

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_S$

<div =.*>

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_A$

<script.* javascript=1.2 .*>

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_A$

<table=.*>

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_A$

Figure 5.6: Snapshot of the knowledge base after an automated update.

<a accesskey=.*>

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_B$
$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_S$

<script.* javascript=1.4 .*> ∧ ! <script.* javascript=1.3.*>

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_Q$
$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_R$
$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_W$

<layer bgcolor=.*>

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_A$
$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_S$

~~<div =.*>~~

~~$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_A$~~

<script.* javascript=1.2 .*>

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_A$

<table=.*>

$(\text{configatt}_1 \wedge \text{configatt}_2 \wedge \text{configatt}_3 \wedge \ldots \wedge \text{configatt}_n)_A$

Figure 5.7: Snapshot of the knowledge base after information solicitation.

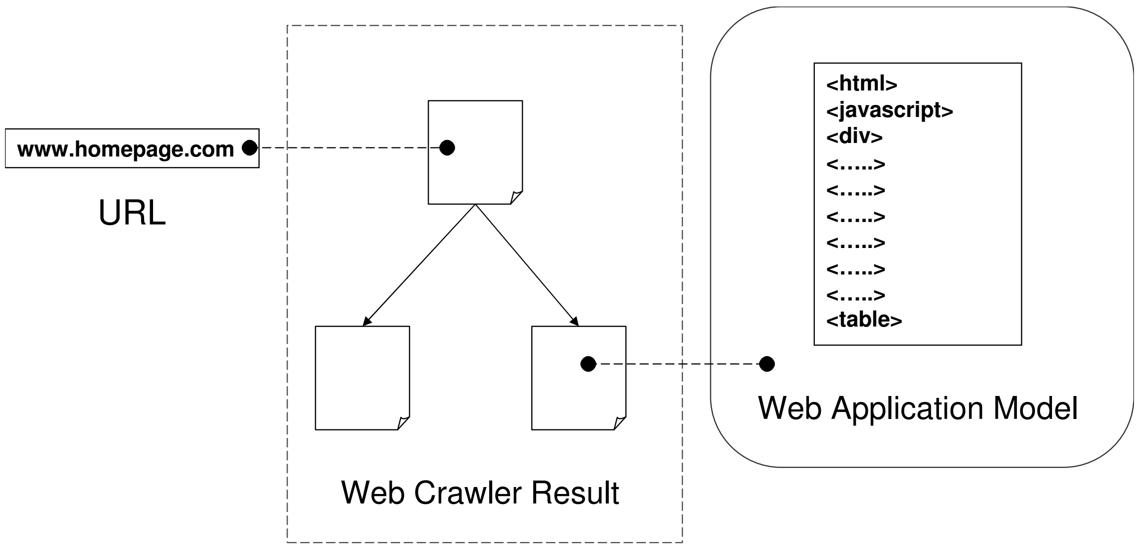


Figure 5.8: The retrieval of data, implemented by `processURL()`, begins once the user submits a URL. From there, the corresponding web page is fetched and, based on the hyperlinks observed, a crawler collects each of the web pages that are a part of the site. Once the source code is retrieved, a vector model of the web application is created.

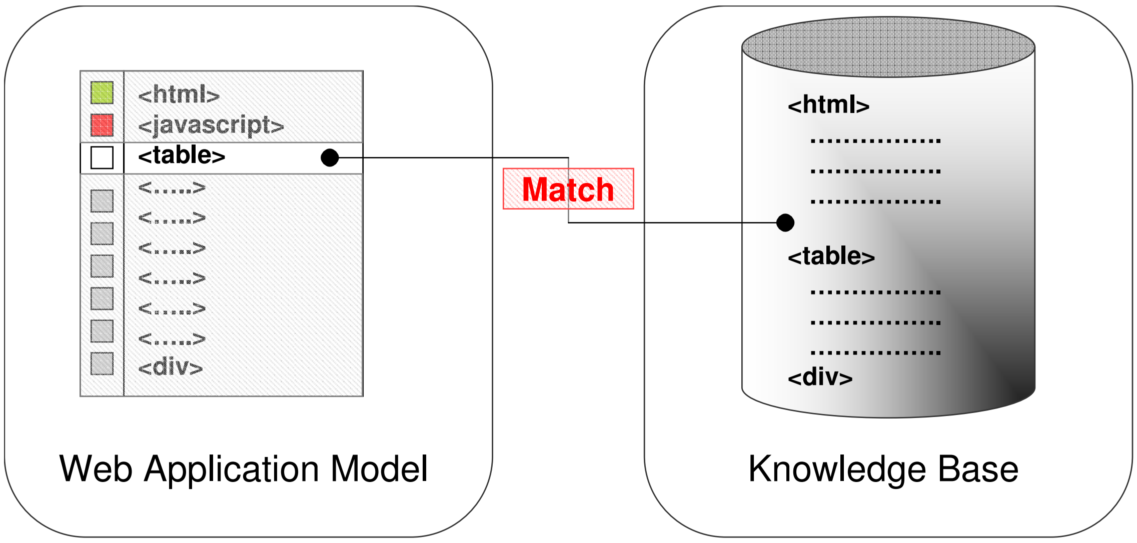


Figure 5.9: An overview of `query()`

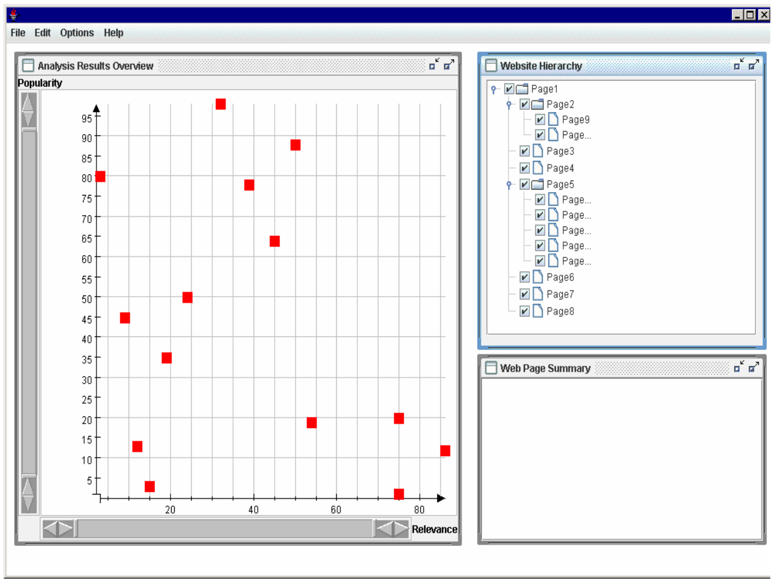
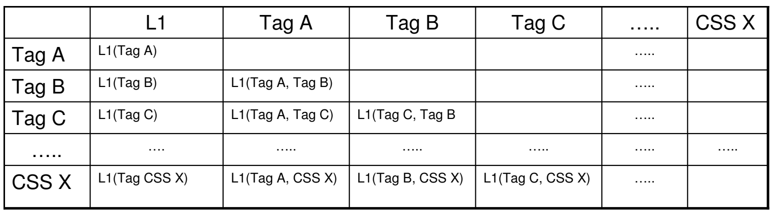


Figure 5.10: Visualization of compliance analysis results

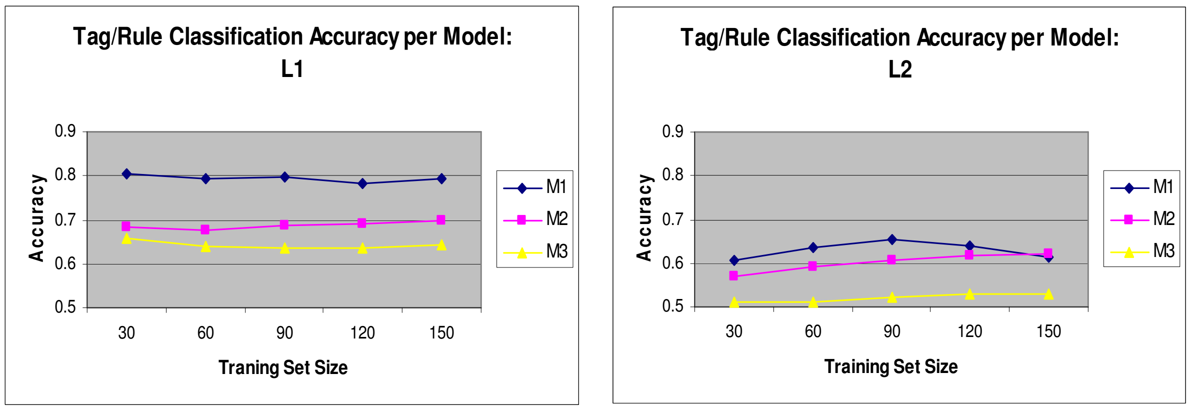


	L1	Tag A	Tag B	Tag C	.....	CSS X
Tag A	L1(Tag A)				.....	
Tag B	L1(Tag B)	L1(Tag A, Tag B)			.....	
Tag C	L1(Tag C)	L1(Tag A, Tag C)	L1(Tag C, Tag B		.....	
.....	....	.....	.....	.....	.....	.....
CSS X	L1(Tag CSS X)	L1(Tag A, CSS X)	L1(Tag B, CSS X)	L1(Tag C, CSS X)	.....	

Figure 5.11: The interaction matrix

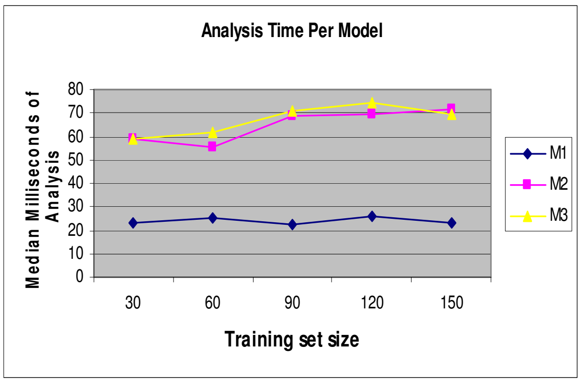
Figure 5.12: Accuracy Values Defined

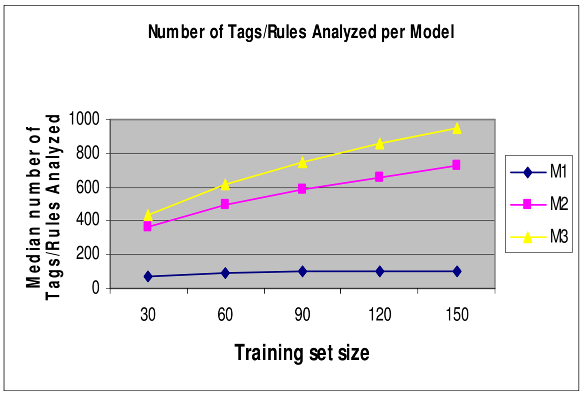
		$GS(t, c_x)$	
		yes	no
$AR(t, c_x)$	yes	TP	FP
	no	FN	TN



(a) (b)

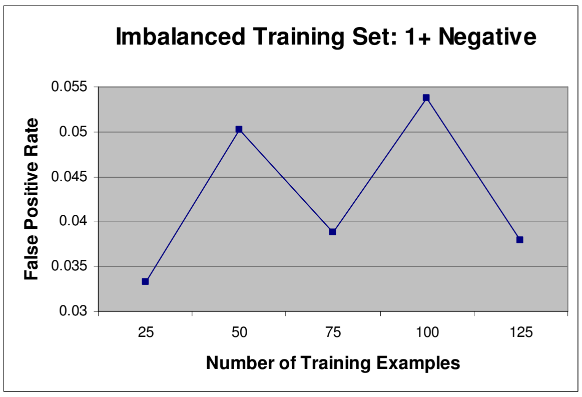
Figure 5.13: The affect of learning strategy, training set size, and web application model on learning accuracy. The graph shown in (a) corresponds with the L1 learning strategy; (b) corresponds with L2.



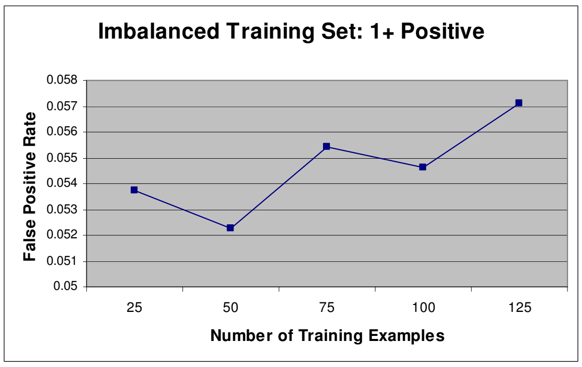


(a) (b)

Figure 5.14: The affect of web application model on time needed for analysis and the number of tags/rules analyzed. The graph shown in (a) shows the time needed(b) shows the number of tags analyzed.



(a)



(b)

Figure 5.15: The affect of training set imbalance on false positive rate. The graph shown in (a) shows what results with an extra negative example (b) shows the results with an extra positive training example.

Bibliography

[1] Badros, G. J., Borning, A., Marriott, K., and Stuckey, P. Constraint cascading style sheets for the web. In *UIST '99: Proceedings of the 12th annual ACM symposium on User interface software and technology* (New York, NY, USA, 1999), ACM Press, pp. 73–82.

[2] Bellettini, C., Marchetto, A., and Trentini, A. TestUml: user-metrics driven web applications testing. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing* (New York, NY, USA, 2005), ACM Press, pp. 1694–1698.

[3] Berghel, H. Using the www test pattern to check HTML compliance. *Computer 28*, 9 (1995), 63–65.

[4] Berghel, H. HTML compliance and the return of the test pattern. *Communications of the ACM 39*, 2 (1996), 19–22.

[5] Bishop, J. Multi-platform user interface construction: a challenge for software engineering-in-the-small. In *ICSE '06: Proceeding of the 28th international conference on Software engineering* (New York, NY, USA, 2006), ACM Press, pp. 751–760.

[6] Bobby. http://www.watchfire.com/products/webxm/bobby.aspx.

[7] Bowring, J. F., Rehg, J. M., and Harrold, M. J. Active learning for automatic classification of software behavior. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis* (New York, NY, USA, 2004), ACM Press, pp. 195–205.

[8] Browser photo by netmechanic. http://www.netmechanic.com/browser-index.htm.

[9] Browsershots. http://browsershots.org/.

[10] Brun, Y., and Ernst, M. D. Finding latent code errors via machine learning over program executions. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 480–490.

[11] Chen, B., and Shen, V. Y. Transforming web pages to become standard-compliant through reverse engineering. In *W4A: Proceedings of the 2006 international cross-disciplinary workshop on Web accessibility (W4A)* (New York, NY, USA, 2006), ACM Press, pp. 14–22.

[12] Clark, J. The glorious peoples myth of standards compliance. http://joeclark.org/glorious.html.

[13] Cohen, M. B., Snyder, J., and Rothermel, G. Testing across configurations: implications for combinatorial testing. *SIGSOFT Softw. Eng. Notes 31*, 6 (2006), 1–9.

[14] Cubranic, D., and Murphy, G. C. Automatic bug triage using text categorization. In *SEKE '04:Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2004)* (2004), ACM Press, pp. 92–97.

[15] Diep, M. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Trans. Softw. Eng. 31*, 4 (2005), 312–327. Member-Sebastian Elbaum.

[16] Doctor HTML. http://www2.imagiware.com/RxHTML/.

[17] Eaton, C., and Memon, A. M. An empirical approach to testing web applications across diverse client platform configurations. *International Journal on Web Engineering and Technology (IJWET), Special Issue on Empirical Studies in Web Engineering* (2007).

[18] Elbaum, S., Karre, S., and Rothermel, G. Improving web application testing with user session data. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering* (2003), IEEE Computer Society, pp. 49–59.

[19] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles* (2001), ACM Press, pp. 57–72.

[20] Hangal, S., and Lam, M. S. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering* (2002), ACM Press, pp. 291–301.

[21] Hao, D., Zhang, L., Mei, H., and Sun, J. Towards interactive fault localization using test information. In *APSEC '06: Proceedings of the XIII Asia Pacific Software Engineering Conference* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 277–284.

[22] Haran, M., Karr, A., Orso, A., Porter, A., and Sanil, A. Applying classification techniques to remotely-collected program execution data. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2005), ACM Press, pp. 146–155.

[23] Streamlining software testing with IBM Rational and VMware. http://www.vmware.com/pdf/rational.pdf.

[24] Kallepalli, C., and Tian, J. Measuring and modeling usage and reliability for statistical web testing. *IEEE Trans. Softw. Eng. 27*, 11 (2001), 1023–1036.

[25] Khaksari, G. H. Expert diagnostic system. In *IEA/AIE '88: Proceedings of the 1st international conference on Industrial and engineering applications of artificial intelligence and expert systems* (New York, NY, USA, 1988), ACM Press, pp. 43–53.

[26] Koltashev, A. A practical approach to software portability based on strong typing and architectural stratification. In *JMLC* (2003), L. Böszörményi and P. Schojer, Eds., vol. 2789 of *Lecture Notes in Computer Science*, Springer, pp. 98–101.

[27] Korpela, J. Lurching toward babel: Html, css, and xml. *Computer 31*, 7 (1998), 103–104,106.

[28] Kung, D., Liu, C.-H., and Hsia, P. An object-oriented web test model for testing web applications. In *Proceedings. First Asia-Pacific Conference on Quality Software* (2000), pp. 111–120.

[29] Li, Z., Lu, S., and Myagmar, S. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng. 32*, 3 (2006), 176–192. Member-Yuanyuan Zhou.

[30] Li, Z., and Zhou, Y. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2005), ACM Press, pp. 306–315.

[31] Liblit, B., Aiken, A., Zheng, A. X., and Jordan, M. I. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, June 9–11 2003).

[32] Mao, Y. Automated computer system diagnosis by machine learning approaches. Tech. rep., University of Pennsylvania, 2005. Technical Report, MS-CIS-05-17.

[33] Matsumura, T., Monden, A., and ichi Matsumoto, K. A method for detecting faulty code violating implicit coding rules. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution* (2002), ACM Press, pp. 15–21.

[34] Memon, A., Porter, A., Yilmaz, C., Nagarajan, A., Schmidt, D., and Natarajan, B. Skoll: Distributed continuous quality assurance. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (2004), IEEE Computer Society, pp. 459–468.

[35] Mooney, J. D. Bringing portability to the software process. Tech. rep., West Virginia University, Department of Statistics and Computer Science, 1997. Technical Report TR 97-1.

[36] Ng, H. T., Goh, W. B., and Low, K. L. Feature selection, perception learning, and a usability case study for text categorization. In *SIGIR '97: Proceedings of the 20th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (New York, NY, USA, 1997), ACM Press, pp. 67–73.

[37] Phillips, B. Designers: The browser war casualties. *Computer 31*, 10 (1998), 14–16,21.

[38] Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., and Wang, B. Automated support for classifying software failure reports. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 465–475.

[39] Ricca, F., and Tonella, P. Analysis and testing of web applications. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering* (2001), IEEE Computer Society, pp. 25–34.

[40] Ricca, F., and Tonella, P. Web testing: a roadmap for the empirical research. In *WSE* (2005), IEEE Computer Society, pp. 63–70.

[41] Sedlmeyer, R. L., Thompson, W. B., and Johnson, P. E. Knowledge-based fault localization in debugging: preliminary draft. In *SIGSOFT '83: Proceedings of the symposium on High-level debugging* (New York, NY, USA, 1983), ACM Press, pp. 25–31.

[42] Sneed, H. M. Testing a web application. In *In Proceedings of the 6th International Workshop on Web Site Evolution* (2004), IEEE Computer Society, pp. 3–10.

[43] Spiesser, J., and Kitchen, L. Optimization of html automatically generated by wysiwyg programs. In *WWW '04: Proceedings of the 13th international conference on World Wide Web* (New York, NY, USA, 2004), ACM Press, pp. 355–364.

[44] Sterling, C. D., and Olsson, R. A. Automated bug isolation via program chipping. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging* (New York, NY, USA, 2005), ACM Press, pp. 23–32.

[45] Software testing glossary. www.chambers.com.au/glossary/configur.htm.

[46] The advanced html/css reference. http://blooberry.com/indexdot/html/index.html.

[47] TONELLA, P., AND RICCA, F. A 2-layer model for the white-box testing of web applications. In *In Proceedings of the 6th International Workshop on Web Site Evolution* (2004), IEEE Computer Society, pp. 11–19.

[48] World wide web consortium. http://www.w3.org/.

[49] Wikipedia - world wide web consortium. http://en.wikipedia.org/wiki/W3c.

[50] Wikipedia. http://en.wikipedia.org/wiki/Web_application.

[51] WILLIAMS, A., AND PROBERT, R. A practical strategy for testing pair-wise coverage of network interfaces. *issre 00* (1996), 246.

[52] XU, L., AND XU, B. A framework for web applications testing. In *International Conference on Cyberworlds* (2004), pp. 300–305.

[53] XU, L., XU, B., AND JIANG, J. Testing web applications focusing on their specialties. *SIGSOFT Softw. Eng. Notes 30*, 1 (2005), 10.

[54] XU, L., XU, B., NIE, C., CHEN, H., AND YANG, H. A browser compatibility testing method based on combinatorial testing. In *International Conference on Web Engineering* (2003), Springer, pp. 310–313.

[55] YANG, Y., AND PEDERSEN, J. O. A comparative study on feature selection in text categorization. In *ICML '97: Proceedings of the Fourteenth International Conference on Machine Learning* (San Francisco, CA, USA, 1997), Morgan Kaufmann Publishers Inc., pp. 412–420.

[56] ZELLER, A., AND HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *Software Engineering 28*, 2 (2002), 183–200.