An empirical approach to evaluating web application compliance across diverse client platform configurations

Cyntrica Eaton* and Atif M. Memon

Department of Computer Science
University of Maryland
4115 A.V. Williams Building
College Park, MD 20742, USA
Fax: 301–314–1353
E-mail: ceaton@cs.umd.edu
E-mail: atif@cs.umd.edu
*Corresponding author

Abstract: Web applications are the most widely used class of software today. Increased diversity of web-client platform configurations causes execution of web applications to vary unpredictably, creating a myriad of challenges for quality assurance during development. This paper presents a novel technique and an inductive model that leverages empirical data from fielded systems to evaluate web application correctness across multiple client configurations. The inductive model is based on HTML tags and represents how web applications are expected to execute in each client configuration based on the fielded systems observed. End-users and developers update this model by providing empirical data in the form of positive (correctly executing) and negative (incorrectly executing) instances of fielded web applications. The results of an empirical study show that the approach is useful and that popular web applications have serious client-configuration-specific flaws.

Keywords: web testing; inductive learning; cross-platform compatibility; HTML tags.

Reference to this paper should be made as follows: Eaton, C. and Memon, A.M. (xxxx) 'An empirical approach to evaluating web application compliance across diverse client platform configurations', *Int. J. Web Engineering and Technology*, Vol. X, No. Y, pp.000–000.

Biographical notes: Cyntrica Eaton is a graduate student in the Computer Science Department at the University of Maryland. She received her BS in Computer Science from Norfolk State University in 2001. She was awarded fellowships from both the National Consortium for Graduate Degrees for Minorities in Engineering and Science, Inc. (GEM) and The David and Lucile Packard Foundation. Her general research interests include software testing and reverse engineering; her current focus is the application of these principles in web-based technology. She is a student member of the ACM and IEEE Computer Society.

Atif M. Memon is an Assistant Professor at the Department of Computer Science, University of Maryland. He received his BS, MS and PhD in Computer Science in 1991, 1995 and 2001, respectively. He was awarded a Gold Medal in his BS. He was awarded fellowships from the Andrew Mellon

Foundation for his PhD research. He received the NSF CAREER award in 2005. His research interests include programme testing, software engineering, artificial intelligence, plan generation, reverse engineering, and programme structures. He is a member of the ACM and the IEEE Computer Society.

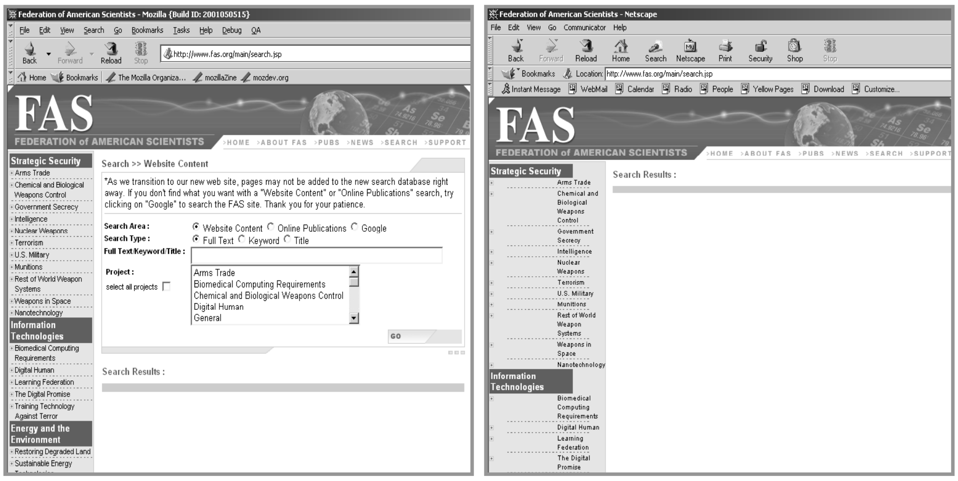
1 Introduction

As one of the most influential technological developments of recent times, the World Wide Web (WWW) (Hansen, 2002) has the potential to impact almost all walks of life (Deshpande and Murugesan, 2001). Currently, WWW users perform many important tasks online, including paying bills, ordering products, communicating via e-mail, and even controlling household appliances. As the WWW has evolved through the years, there have been significant increases in the complexity and variety of tools used to access the web. Moreover, with the surging popularity of the web, the number of web application developers and development tools has increased as well. These trends, coupled with an elevated reliance on web applications and an increased expectation of correctness, make Quality Assurance (QA) of web applications very important (Gaur, 2000; Huang *et al.*, 2003).

This paper focuses on important QA challenges that stem from the increased diversity of client platforms; specifically, on how web applications execute in different client configurations. Ideally, web applications should render and function uniformly across heterogeneous client platforms. In such a situation, QA could effectively be carried out on one client configuration and the results extrapolated for the entire set. Yet, as shown in Figure 1, the makeup of the client configuration has a significant impact on web application execution. Such problems are severely compounded when end-users fine-tune their browsing environments using several dozen options and install different plug-ins, creating an enormous number of unique client configurations. For example, setting/resetting the two binary options 'Use TLS 1.0' and 'Use HTTP 1.1' in Internet Explorer via check-boxes creates four different client configurations, each of which may cause web applications to execute differently.

Existing tools and approaches that evaluate the correctness of web applications across multiple configurations have *limited scope* in that they focus on a small subset of client platform configurations (Browser Photo by NetMechanic[1]; Doctor HTML[2]; Bobby[3]). Moreover, popular tools such as Browser Photo are *non-diagnostic*, *i.e.*, they do not indicate whether a problem was encountered; consequently, they are unable to diagnose the source of the actual problems. To test a web application, the developer 'submits' the *Web Application* (WA), usually by specifying a base URL. For clarity, we define a WA as an application accessed via a web browser over a network.[4] The tool deploys the WA on several popular browsers and returns screen-shots of the WA as displayed in the browsers. The developer examines the screen-shots and relies on visual cues to discover errors. If visual cues (*e.g.*, misrendered page elements) signal an error, the developer then manually examines the application source code to identify the cause. Owing to the limited scope and non-diagnostic nature of these approaches, the result is an incomplete and resource-intensive analysis of the WA. Consequently, client-platform-specific errors surface in fielded web applications. (Section 5 shows several examples of such errors)

Figure 1 The featured web application executes differently on (a) Mozilla 0.9 on Windows XP Professional and (b) Netscape on Windows XP Professional.


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
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...
(a)
(b)

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 :

The research presented in this paper takes an alternative approach on two key fronts. Instead of concentrating on rendering errors flagged by visual cues, we are also interested in another, equally important, class of errors: behavioural faults. Moreover, instead of evaluating a WA on a fixed number of client configurations, empirical data from actual fielded web applications are used to generate an extensible model of the *space* of all possible configurations. Each point in this space represents a unique client configuration. Associated with each point is an inductive model, based on *HTML tags*, of how web applications are expected to execute in the corresponding configuration. HTML tags provide the basis for the inductive model largely because, as building blocks of web applications, they provide directives that indicate how an application should be executed and how users should be able to interact with various application elements. In short, HTML tags are important correctness predictors when support for a given tag is known to be non-existent or insufficient. Therefore, the inductive model *predicts* how a fielded application is expected to execute based on knowledge of the HTML tags in source code and the support provided for each in various target environments. The client configuration model evolves automatically as empirical data in the form of *positive* (correctly executing) and *negative* (incorrectly executing) fielded web applications are discovered for particular configurations. A developer checks the correctness of a WA by 'querying' the model and obtaining, as output, a set of <*client configuration, unsupported HTML tag, faulty Web page*> triples. In general, web applications are usually composed of a set of interrelated web pages. For the purpose of this research, we are interested in the individual web pages that make up the application.

Our new technique has been implemented in a tool called the Internet Compliance Engine (ICE). We evaluate our technique in an empirical study involving 16 client configurations with 100 positive and 100 negative instances each. These instances are used to create a client configuration space and an associated inductive model. The resulting model is then used to evaluate the correctness of several popular web applications based on knowledge of the HTML tags contained in the source and the relative support of each. The results of the study show that our technique is both (a) *practical*, in that the 200 instances were used to create each model with very little effort and (b) *useful*, in that it helped to identify client-platform-specific *support errors* in widely used web applications.

One of the limitations of this technique is the misclassification of tags, in particular, the occurrence of false positives. A false positive is a tag that is actually unsupported yet has erroneously been labelled as supported by the algorithm. In reporting the results of our findings, we use the relative number of false positives discovered as a means of evaluating the technique as the sample size increases.

This paper makes the following contributions to the field of web application QA:

- a formal model of web client configurations
- an inductive model, based on HTML tags, that evolves as users submit positive and negative instances of web applications
- an empirical study that demonstrates that the inductive model is useful and practical
- evidence that popular web applications have serious client-configuration-related problems.

Structure of the paper

The next section presents an assessment of current techniques. Section 3 gives a high-level overview of the operation of ICE. Section 4 presents details of the inductive model and algorithms to create and update the model. An empirical evaluation of the technique is presented in Section 5. Finally, the paper concludes with a discussion of ongoing and future work.

2 Related work

Although the literature recognises client-configuration-specific failures as a serious problem (Kallepalli and Tian, 2001; Xu *et al.*, 2003; Offutt, 2002; Dávila-Nicanor and Mejía-Alvarez, 2004; Brajnik, 2004), none of the tools in 'The QA Toolbox' at the WWW consortium website[5] addresses this issue. The Device Independence Working Group at the WWW consortium[6] has started to discuss issues related to this problem. However, their focus is on developing new standards for future web applications. There are currently very few practical solutions available to web developers. Some of these solutions and their limitations are discussed next.

2.1 Manual execution-based techniques

The most popular approach to evaluating a web application across multiple client configurations is to launch the application in several *<browser, browser version, operating system>* permutations and analyse the results (Eaton and Memon, 2004a–b). Verification of web application correctness, using this strategy, is essentially a qualitative comparison between expected and observed presentation and functionality. While this approach features first-hand exposure to existing web application QA issues, lack of client configuration availability during testing can substantially influence the breadth of target client platforms explored. To verify web application correctness, the application must be manually loaded and if a target environment is unavailable, subsequent analysis is infeasible. In addition, the time and effort required to effectively assess web applications in this manner can also impede the depth of the web application evaluated.

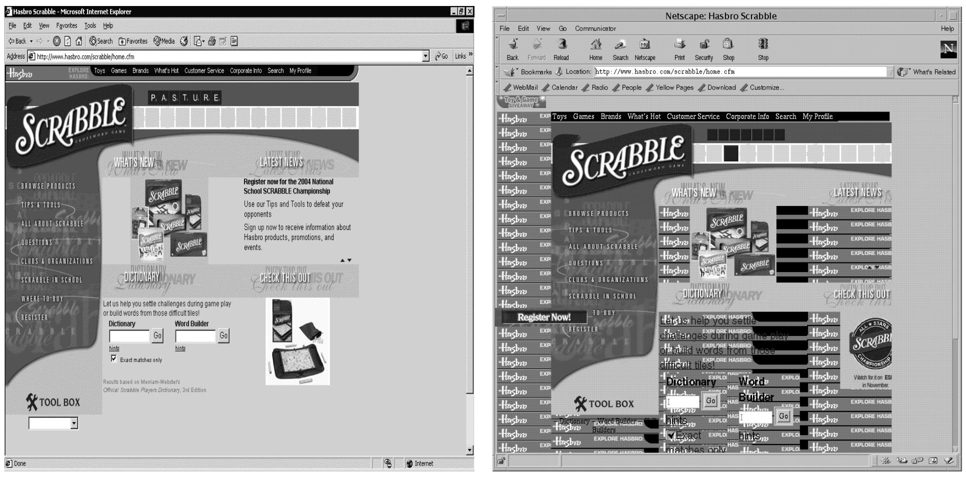
2.2 Automated execution-based techniques

Tools such as Browser Photo partially automate the above approach by loading specific web pages in a pre-defined set of browsing environments and mailing screen-shots of the results back to the user. The user then manually determines the correctness of the WA by examining the screen-shots. If a visible error is discovered, the user manually determines the cause of the problem. While this approach automates the process of loading web applications, it is still resource intensive, non-diagnostic, and limited to a small, fixed set of client configurations. In addition, it can only be used to detect failures evident from visual inspection. One practical example of this problem surfaces when HTML tags have an `accesskey` attribute. Functionally, the `accesskey` tag attribute is intended to provide shortcut access to elements on a page. However, if this attribute is unsupported in a client configuration (*e.g.*, one using the `Opera` browser), it is not evident from visual examination of screen-shots. Moreover, in many of these tools, the web application can only be observed above the fold since scrolling to see the entire screen is not an option in individual screen-shots.

2.3 Automated static techniques

An orthogonal, static approach shifts the basis of quality analysis from a comparison between actual and expected web page appearance to a comparison between the *HTML tags* used to structure web pages and the support provided for each in target environments (Doctor HTML; Bobby). In general, the basis of most static evaluations is recognition of code patterns consistently associated with erroneous behaviour. In terms of the web, characterisation of these error-inducing code fragments is relatively straightforward; browsing environment execution obstacles to arise when unrecognised and, subsequently, unsupported HTML tags are encountered in the document source code. Evaluating the correctness of a web application within a client configuration in this context involves identifying unsupported HTML tags in the underlying source code. Hence, the strength of this analysis depends on the quality and completeness of the tag support rule set. For example, *Doctor HTML* failed to diagnose the problem shown in Figure 2.

Figure 2 The problem featured here was not detected by Doctor HTML.



Hasbro Scrabble - Microsoft Internet Explorer
http://www.hasbro.com/scrabble/home.cfm
Netscape: Hasbro Scrabble
http://www.hasbro.com/scrabble/home.cfm
SCRABBLE
WHAT'S NEW
LATEST NEWS
DICTIONARY
CHECK THIS OUT
TOOL BOX

(a) (b)

2.4 Testing specific configurations

Instead of checking the correctness of a given web application on a variety of client configurations, Berghel (1996) focuses on testing a specific client configuration for a specific set of web applications. In this approach, a *Web Test Pattern* is composed of a suite of test web pages, each of which incorporate several HTML tags and descriptions of the impact they would have if executed correctly. This approach allows users to test their particular client configurations to determine compliance levels. However, this type of testing is severely inefficient for web developers interested in establishing compliance across a wide variety of platforms. To further illustrate the point of inefficiency, consider the example shown in Figure 1. In that case, the developer would be forced to launch the page in Netscape 4.8 on the Windows XP platform before noticing the existence of a problem. The need to have each possible environment available and to launch the page in the corresponding environment in order to detect a problem is clearly time and resource intensive.

2.5 QA research directions for web applications

Given increased interest in the quality and reliability of web applications, there have been quite a few research efforts directed towards establishing effective QA techniques. These include general frameworks (Xu *et al.*, 2005; Sneed, 2004; Ricca and Tonella, 2005; Xu, 2004), test-case generation strategies (Bellettini *et al.*, 2005), traditional white-box testing techniques (Tonella and Ricca, 2004; Ricca and Tonella, 2001), object-oriented strategies (D.C. Kung and Liu, 2000) and statistical testing approaches (Kallepalli and Tian, 2001). These tools and techniques assess the functionality and performance of web applications. While our work shares in the same spirit of attempting to identify effective QA techniques, factoring in the browsing environment as a cause for web application failure sets our work apart from those previously mentioned.

A small body of work does consider the browsing environment to be a critical factor in evaluating the correctness of web application functionality. Lucca and Penta (2003) use the concept of state charts to encode the effect of navigation buttons, such as forward and backward, on the web application state. More specifically, this approach allows for the discovery of failures caused by the use of navigation buttons. This particular work is different from ours because it does not consider the type of browser used. Moreover, in our current model, we do not attempt to test for the interaction between a series of web pages within a given web application.

The work presented in Xu *et al.* (2003) is quite closely related to ours. The motivation behind their work is to ensure that web applications render and function properly in various browser configurations. The crux of the approach is to reduce the number of exhaustive test cases needed based on single factor- and pair-wise coverage criteria. In contrast, our approach is to derive a model of supported and unsupported tags tailored to specific environments. This allows us to derive a detailed knowledge of tag support issues for each environment.

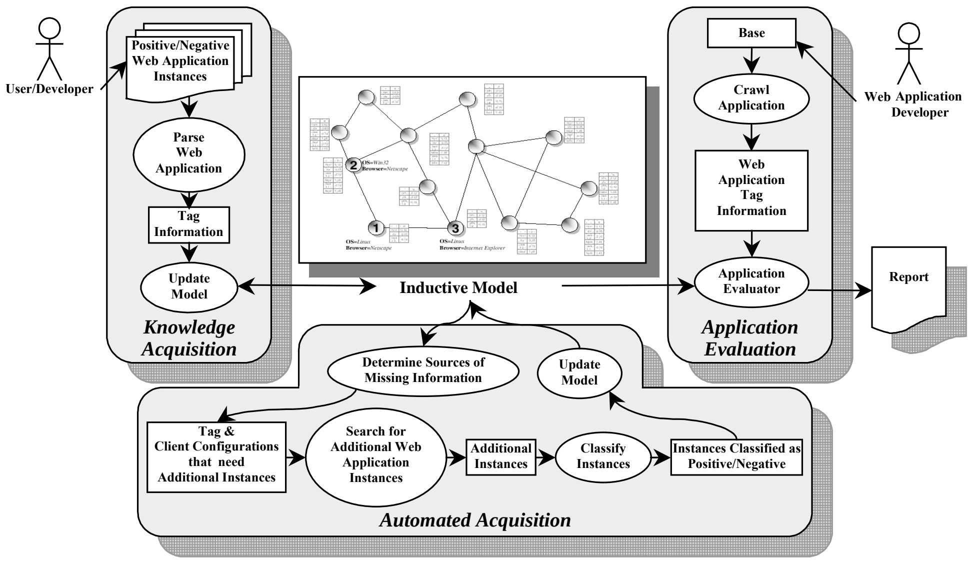
The research presented in this paper builds upon some of the above techniques. It leverages the HTML tags embedded in web applications and builds an inductive model of tag support in client environments. This work improves upon the above techniques by using empirical data, in the form of positive and negative web application instances, to infer knowledge of tag support provided in specific client configurations. Making this information available will improve similar approaches since it will provide the basis for a more accurate, comprehensive evaluation. The next section gives an overview of our technique, which is implemented in the tool ICE.

3 ICE in action

Figure 3 gives a high-level overview of three processes implemented in ICE. The central component, the inductive model (described in the next section), is shared by these three processes. A *Knowledge Acquisition* process takes as input multiple instances of positive and negative WAs. ICE analyses these instances by parsing the HTML source of each, collecting tag information and updating the inductive model. Since the inductive model relies on manually provided instances of positive/negative web applications, it is expected that some of the information may be insufficient for analysis. There is a key relationship between the amount of data submitted and the classification of a given HTML tag as supported or unsupported. This mainly results from the fact that each web application serves as evidence of the behaviour of the tags featured within its source code. Roughly, the more often a tag appears in faulty web applications, the more likely it is to be unsupported. In terms of a more practical example, if users have submitted only one web application that features a specific tag, the results of this submission cannot be extrapolated to other web applications. A process called *Automated Acquisition* determines the sources of missing information (usually tags), spawns a web search (*e.g.*, Google) to locate other applications that contain this specific tag, submits the returns for manual classification as either positive or negative, and updates the model with the results. A third process called *Web Application Evaluation* will be used by web application developers to test their WA. The WA developer specifies the base URL of the application. The WA is traversed using a crawler and its tag information is extracted. An automated *Application Evaluator* then employs the latest version of the inductive model to check the correctness of the WA. The web application is considered to be correct for a given client configuration if and only if every HTML tag embedded in the source code is recognised by the inductive model as supported by the client configuration. The output report is a set of <*client configuration, unsupported HTML tag, faulty Web page*> triples which indicate the specific client configuration, HTML tag and the problematic web page in the web application.

The next section describes the structure of the inductive model and algorithms to create, update and query it.

Figure 3 ICE in action


User/Developer
Positive/Negative Web Application Instances
Parse Web Application
Tag Information
Update Model
Knowledge Acquisition
OS=Win32 Browser=Netscape
OS=Linux Browser=Netscape
OS=Linux Browser=Internet Explorer
Inductive Model
Base
Crawl Application
Web Application Tag Information
Application Evaluator
Web Application Developer
Report
Application Evaluation
Determine Sources of Missing Information
Update Model
Tag & Client Configurations that need Additional Instances
Search for Additional Web Application Instances
Additional Instances
Classify Instances
Instances Classified as Positive/Negative
Automated Acquisition

4 Inductive model

A central component of our approach is a formal model of HTML support in specific client configurations. This model contains two parts:

1 a *graph representation* of client configurations

2 an *association vector* for each client configuration that relates HTML tags to web application failures. This section describes the details of these two parts.

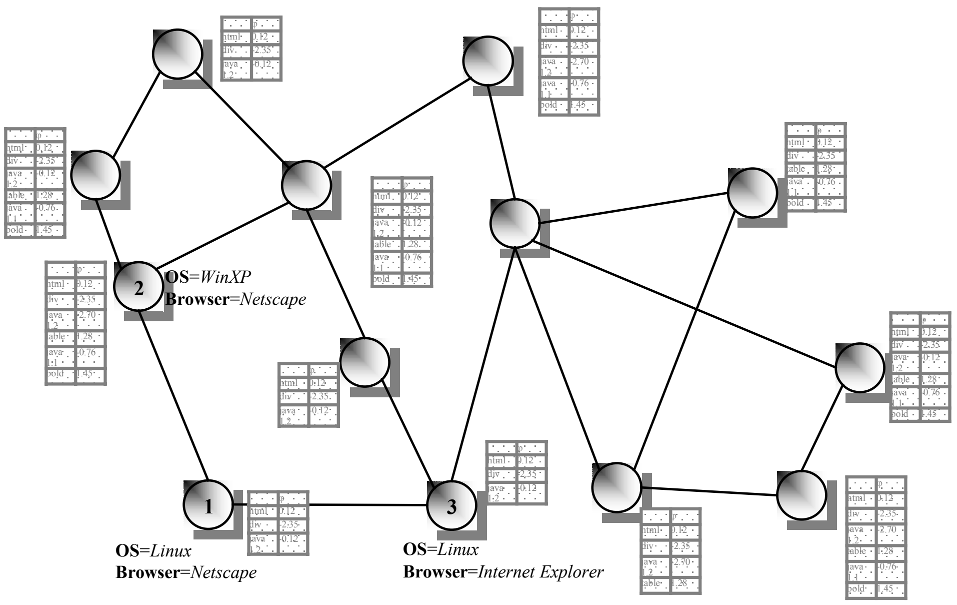
4.1 Modelling client configurations

Each client is described in terms of *options*, such as operating system installed, browser, browser settings, network speed and geographical location. Each option takes its value from a discrete set of *settings*. For example, the Operating System option (called OS) in our empirical study 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.*, the (Browser=Web TV) and (Version=2.0) feature is not supported when (OS=Linux)). Therefore, we allow *inter-option constraints* which limit the allowable settings of one option based on the settings of others. We represent constraints as $(P_i \rightarrow P_j)$, meaning '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, –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. An example of a constraint is ((Browser=IE) and (Version=6.0)) → (OS=Windows XP) which means that if the Internet Explorer 6.0 browser is used in a run-time environment, then the operating system must be Windows (since version 6.0 is not available for other operating systems). A *valid configuration* is a configuration that violates no inter-option constraints.

Figure 4 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 by one option setting (`Browser=Netscape` vs. `Browser=InternetExplorer`). 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 4.3). Without loss of generality, we assume that the client configuration space is connected (*i.e.*, it is one connected graph; 'dummy' nodes are used to connect disjoint parts).

Figure 4 An example of a client configuration space



4.2 Modelling 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, our empirical study (Section 5) showed that the `<blink>` tag does not work in Netscape browsers; the association vector for each client configuration that has `Browser=Netscape` should link 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. The first step in deriving tag support knowledge is to evaluate the correlation coefficient, ϕ, of each discovered tag (Yang and Pedersen, 1997; Ng *et al.*, 1997). Intuitively, ϕ uses observance of positive and negative phenomena to estimate the association of an element to one category or another. Note that the association vector is essentially a collection of ϕ values for all of the tags in a client configuration. Since the set of ϕ values are generally 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*:

$$\varphi(t,c) = \begin{cases} \dfrac{\sqrt{N} \times (AD - CB)}{\sqrt{(A+C)\times(B+D)\times(A+B)\times(C+D)}} \\ 0, \qquad \text{if } A+C=0; B+D=0; A+B=0; C+D=0 \end{cases} \tag{1}$$

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*. 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 ($B + D = 0$), no occurrence 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 measure centres on 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 characterised. Subsequently, ϕ allows us to predict the risk that a tag is unsupported in a given configuration. We essentially want to determine the tags most responsible for the classification of a page as a negative instance by identifying tags that have a high association with faulty web pages. Note that 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 incompliant environment.

For a concrete example, consider the set of web applications classified as positive or negative for an arbitrary client configuration shown in Figure 5. Note that 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 configuration. 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 shown in Figure 5 and their corresponding ϕ values is shown in Table 1.

Figure 5 Set of web applications classified as positive or negative

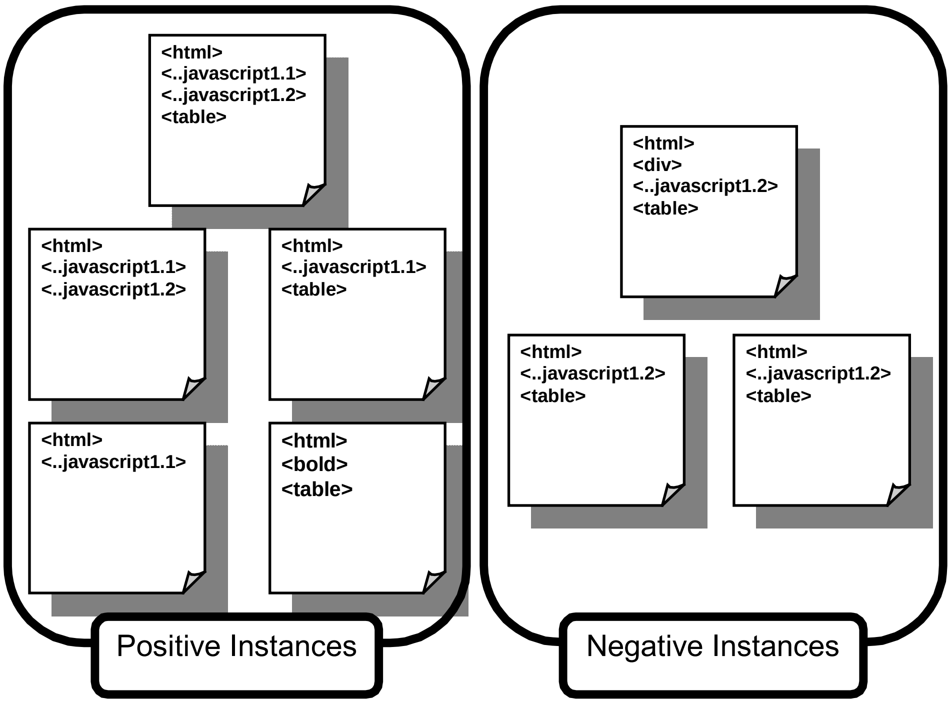


Table 1 Values for all tags in the example in Figure 5

Tag	*<html>*	*<div>*	*<javascript 1.2>*	*<table>*	*<javascript 1.1>*	*<bold>*
ϕ	0.00	–1.38	–1.70	–1.26	2.19	0.83

In the next section, we describe algorithms to (a) create/update an association vector when new positive/negative instances of web applications are available and (b) use the vector to test a given web application.

4.3 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 6 is invoked (via the two processes *Knowledge Acquisition* and *Automated Acquisition* shown in Figure 3) *for each web page* in the instance.

Figure 6 The `updateVector()` algorithm

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

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

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

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

As shown in Figure 6, `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

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* in 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 recognised as an 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 that in this approach, we are using ϕ as a predictive measure of tag support; as the values of ϕ change, the tag can be reclassified as supported/unsupported by our system. However, whether the tag is truly supported or unsupported in a given environment does not change. We note that although we have described an ‘aggressive’ algorithm that updates ϕ values each time a new positive/negative instance is available, 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.

4.4 Algorithm to use the inductive model

The *Web Application Evaluation* process shown in Figure 3 queries the inductive model to determine the set of configurations in which a given web application will execute incorrectly. The process invokes an algorithm called `queryData()`, shown in Figure 7. 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 is terminated. Again, note that for simplicity, we have not described details that improve the efficiency of the overall process. For example, ICE only examines configurations whose association vectors contain tags relevant to the application being tested. In this case, tags are relevant to the application if they are actually contained within the source code.

Figure 7 The `queryData()` algorithm

```
Algorithm::queryData(W /*WebApplication*/){
    FORALL config ∈ clientConfigurations DO {
        /*get the list of unsupported tags associated with the current configuration*/
        unsupportedTags = config.unsupportedTags;
        FORALL w ∈ W DO {
        /*get the list of tags in the current Web page*/
        currentTags = getTags(w);
        /*check to ensure that the current Web page does not include
                any of the unsupported tags*/
        FORALL f ∈ unsupportedTags DO {
            IF (f ∈ currentTags) {
                RETURN_FAULTY(config,f, w); break}}}}
```

5 Empirical study

Having described the inductive model and the processes/algorithms that create/update and use the model, we now present details of an empirical study conducted to determine the feasibility and utility of the overall approach. The major research questions we are attempting to address centre around the ability of the correlation coefficient, ϕ, to distinguish between supported and unsupported tags and the impact of sample size on the results.

5.1 Infrastructure

In order to conduct the study, the algorithms listed in Figures 5 and 6 were implemented in Java. The 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, Javascript 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 2. We chose this sample set in order to reflect the wide diversity of usage environments. To analyse the results, we modelled the gold standard or actual knowledge of tag support rules in Microsoft Excel and developed several Visual Basic scripts to summarise the data.

Table 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>

5.2 Empirical method

5.2.1 Research questions and evaluation strategy

Our empirical method was designed to answer the following questions:

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

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 because it justifies the purpose of our study. In the same vein, the second question was designed to analyse the viability of the approach we discuss in this paper. The third question was posed because users play a key role in the application of our approach; ease of use, therefore, is an important consideration. The fourth question, on the other hand, addresses the ability to utilise user input to expand and improve the model. To address the spirit of the fifth question, we expect the misclassification of tags, in terms of False Positives (*FP*s), to have a large impact on the feasibility of the approach. To be clear, we consider negative classification of a tag to mean that the tag is not supported in a given environment and positive classification to indicate that the tag is supported. Subsequently, a false positive is an unsupported tag labelled 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 we use to evaluate the approach penalise techniques that allow for more false positives. As a result, we have defined the measure *FPR* and calculate its value as follows:

$$FPR = \frac{FP}{Total\ number\ of\ tags} \tag{2}$$

We pose the sixth and final question to observe whether *FPR* improves as more information is obtained.

5.2.2 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 our experimental design, the client configuration is neither an independent nor a dependent variable.

5.2.3 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_{cx}, of positive and negative web pages.

Step 3 Parse web application source code, extract the HTML tags, and abstract the tags using the conditioning technique, *TC*. This will produce $P_{cx,\,TC}$, 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 Model 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 $P_{cx,\,TC}$.
- c Calculate the corresponding *FPR* value.
- d For five iterations...
  - 1 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.)

2 Generate the inductive model by calculating ϕ for $P_{cx, TC}$.

3 Calculate the *FPR* value for the inductive model.

A detailed account of each step follows in subsequent sections.

5.2.4 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 criterion in mind is shown in Table 2.

5.2.5 Step 2: training set selection

Each of the 16 configurations listed in Table 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 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. We used the Google search engine 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, since 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 ICE for automated analysis took a few seconds per web page. Note that submission to ICE 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 negative 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 five positive instances and eight negative instances. The inductive model did not contain sufficient information about these tags to be useful for analysis. We then started the automated acquisition process using the Google search engine. More specifically, we posed queries 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 last two 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 labelled as a positive instance.

Table 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}

5.2.6 Step 3: tag extraction/abstraction

ICE 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 the folding of 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.

More specifically, certain HTML attributes such as `width`, `href` and `summary` have numbers, URLs and strings of text as values. We do not want to discriminate between these tags based on their specific values. To understand how we handle this problem, consider the following:

`<table summary="XYZ">` and

`<table summary="ABC">`.

We are only interested in 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, we have designed an abstraction strategy 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, ICE automatically conditions them.

5.2.7 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. We manually define our ground truth with the help of a website that provides tag support data. The gold standard can be modelled 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 (3):

$$GS(t, c_x) = \begin{cases} yes, \text{ if tag } t \text{ is supported in configuration } c_x \\ \qquad\qquad no, \text{ otherwise} \end{cases} \tag{3}$$

5.2.8 Step 5: tag classification and evaluation

Recall that we are using the correlation coefficient, ϕ, 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, we use the function *AR* (actual results), which is analogous to the *GS* calculation shown in Equation (3). 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 modelled as shown below:

$$AR(t, c_x) = \begin{cases} yes, \text{ if } c_x \text{ is expected to support } t \\ \qquad no, \text{ otherwise} \end{cases} \tag{4}$$

Subsequently, a false positive occurs when $GS(t, c_x) = no$ and $AR(t, c_x) = \text{yes}$. In this step, we compare the predicted classification of the tag with that of the gold standard and use the *FPR* equation introduced earlier to determine how well the classification strategy performed.

5.3 Threats to experimental validity

5.3.1 Internal validity

The internal validity of experimental results is threatened when results of the dependent variable could be tainted by modelling and measurement errors. In each of the questions we address, *FPR* is the primary dependent variable. Hence threats to internal validity, in this context, include possible errors in measuring/designating the training set and modelling/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 be evaluated to a higher value than it actually should.

5.3.2 External validity

Threats to external validity, on the other hand, limit the ability to generalise results. Several candidates for this constraint apply. For one, we are currently only considering 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. 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 our expectation that inductive model accuracy will improve as more examples are submitted, the volume of data provided is important. We have acknowledged this and attempted to include an adequate number of pages in our experiment; similar considerations must be made to ensure the success of the tool in practical settings.

5.4 Results and discussion

Recall that we had six major questions we wanted to address in this study. In addressing the first question, we observed quite a few web applications that rendered and behaved properly in one environment yet were faulty in another (examples of this follow in Figure 9). In addition, we have spoken to several individuals who have run across such problems in very frustrating situations. 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, our 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, we learned that it takes a few seconds to report a problematic web application and associated client configuration, indicating minimal manual effort. Note that end-users are not responsible for indicating why an error occurred; they merely submit faulty pages to ICE, 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. With regard 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 occurrence of false positives in certain inductive models.

In the case of the fifth question, which addresses the performance of the technique, the inductive model that we created in this study yielded useful results. Given the data generated in our study, we have shown that our approach rarely labels an unsupported tag as supported in a given environment. While our model is promising, in principle, it is not complete. This is largely because of two issues:

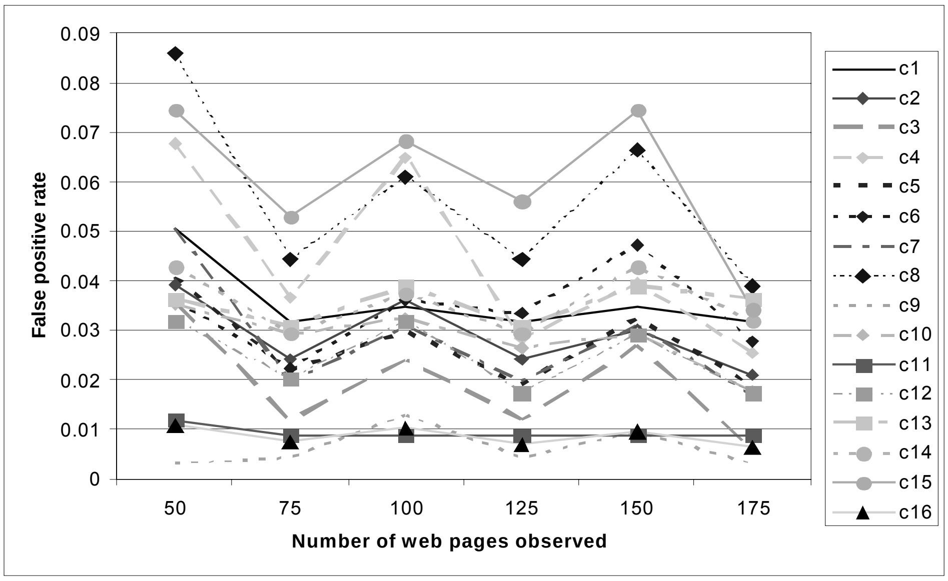
1 Information for every possible tag is not included in the association vector.

2 The tag information represented is not extensive.

Consequently, we observed the presence of false positives. We attribute these errors to incomplete/inaccurate information. It is necessary that observations of positive and negative web applications be large enough to draw accurate conclusions based on the data seen. More web page instances containing new and previously discovered tags must be identified and analysed.

The graph shown in Figure 8, which plots the rate of false positives as the training set size grows, was generated to address the sixth and final question. 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 labelling 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. We attribute this to the fact that there are 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 false positive rate generally decreases from 50 to 100 to 150 and from 75 to 125 to 175. Subsequently, we can conclude from the data that the results improve as the training set grows and that the false positive rate is best when there are more negative examples than positive examples.

Figure 8 False positive rate with respect to training set size



Web application evaluation

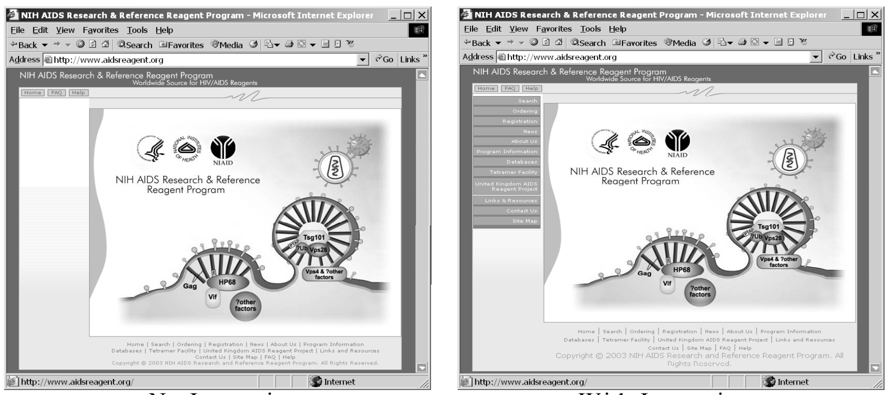
Finally, we were ready to use the inductive model to evaluate a new set of web applications. We chose 12 popular web applications as our subject applications. Note that these applications were not part of our application pool used to create the model. They are shown in Column 2 of Table 4. We found several problems with these applications. The client configuration in which these applications did not execute correctly is shown in Column 3 of Table 4; Column 4 shows one of the problematic tags for each.

Table 4 Evaluation results

#	*URL*	*Client*	*Problematic tags*
1	www.aidsreagent.org/	$c_{2,4,6,\ldots,16}$	`<script language = "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	$c_{2,4,6,\ldots,16}$	`<div onmouseover() = #>`
5	members.dcn.org/ez112654/html/h51.html	c_{5-16}	`<blink>`
6	www.musiciansunion.org.uk/html/index.php	$c_{1,\ldots,4,9,\ldots,16}$	`<a accesskey = #>`
7	www.execlangser.com/	$c_{1,\ldots,4,9,\ldots,16}$	`<basefont face = #>`
8	www.koko.gov.my/CocoaBioTech/Southern.html	$c_{1,\ldots,4,9,\ldots,16}$	`<marquee>`
9	www.sltrib.com/	c_{5-16}	`<ilayer bgcolor = #>`

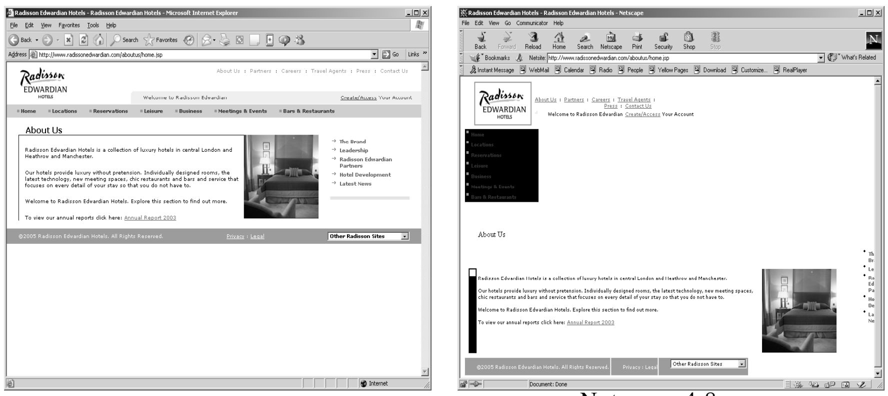
Screen-shots of some of these applications are shown in Figure 9. Each of the examples featured in Figure 9 have visibly evident errors. Yet, as noted before, while an image of web applications functioning in varied environments indicates the existence of problems, the causes of such problems can only be identified with deeper analysis. Using our 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 (number 6 in the table), highlights the problem when screen-shots are not enough to recognise errors. More specifically, the web application features 'hotkey' access to various tool components, yet this functionality is not available in Opera 6.0. Our technique allows us to diagnose 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, several tags caused problems in multiple client configurations. An example is the `<div onmouseover()=#>` tag that caused problems in configurations $c_{2,4,6,8,\ldots,16}$. We expect that in some specialised 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, we would like to explore the nature of such instances. Recognising that such similarities exist can effectively prune the search space utilised by `queryData()` (Section 4.4) since the detection of such a tag would signal failure for all associated browsing environments.

Figure 9 Examples of configuration-specific errors found in our study


NIH AIDS Research & Reference Reagent Program - Microsoft Internet Explorer
NIH AIDS Research & Reference Reagent Program
Worldwide Source for HIV/AIDS Reagents
NIH AIDS Research & Reference Reagent Program

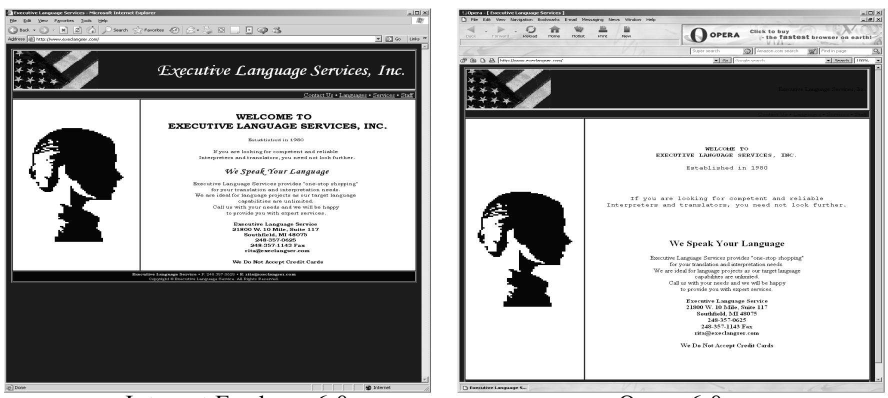
No Javascript With Javascript

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


Radisson Edwardian Hotels - Radisson Edwardian Hotels - Microsoft Internet Explorer
Radisson Edwardian Hotels - Radisson Edwardian Hotels - Netscape
About Us

Internet Explorer 6.0 Netscape 4.8

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


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

Internet Explorer 6.0 Opera 6.0

`http://www.execlangser.com`

6 Conclusions and future work

Successfully leveraging the ability of the web to reach a wide audience is complicated by the varied client configurations used to interact with web content. Providing tools capable of detecting client-configuration-induced faults is extremely important for supporting universal reliability considering the gamut of browsing platforms in use. This paper introduces an effective way to identify client-configuration-related problems that could hinder users from fully exploring and interacting with the information and services featured on a given web application. A new empirical technique based on association vectors, each mapped to a formally defined client configuration, was presented. Vectors evolve for specific environments as additional empirical data in the form of correctly/incorrectly executing web applications on specific platforms become available. The results of an empirical study showed that the approach is feasible and useful. Several client configuration specific problems in fielded web applications were discovered. Indeed there are, perhaps, several different approaches to this problem. This paper only features one. In the future, we plan to evaluate other techniques to assess their strengths and weaknesses.

Our main focus in this work was to derive knowledge of the tags that are and are not supported in a given environment using examples of the pages themselves. The results of this can be applied to a variety of situations. Doctor HTML as an example, or a general parser, could be calibrated for a given environment that is able to assess the compliance of a corresponding source code. However, this is not the crux of our work. Our main goal has been to define an approach that would derive knowledge of tag support rules from fielded examples. In particular, our objective was to generate these rules so that parsers, like the one mentioned, would have an accurate, comprehensive knowledge base during evaluation.

There are several directions for future research. For one, this paper provides an exploratory look at this topic. In this particular phase of our research, we have explored the effects of browser, browser version and operating system on web application execution. In future endeavours, we wish to explore richer client platform descriptions as well as identify causes for performance imbalance outside of HTML tags. The research presented here establishes a basis for our work on an extensible foundation.

In addition, we intend to study approaches that guarantee that the sampling of web applications is statistically fair so that we can rely on the reported frequency of rare events. Second, the algorithms presented here represent our initial attempts to effectively identify tag-related hindrances to web application correctness and to devise a plan for inductively determining the tags that are unsupported in associated client configurations. While we are fairly confident in the design we have incorporated for evaluating compliance for individual web pages, we plan to expand the scope of the learning algorithm with further investigation. For example, we want to be able to analyse the effect of tag interactions on the classification of web applications as positive or negative instances. More specifically, in some cases an unsupported tag may be offset by a supported tag, causing an application that would otherwise be a negative instance to render and execute properly. We would like to establish a reliable measure of such occurrences and incorporate such information in the association vector. Consequently, we will observe techniques employed in case-control studies (Lewallen and Courtright, 1998; Pfeiffer and Morris, 1994; Seaman and Richardson, 2001; Holmes, 1997) in addition to specific techniques for determining the most significant features

(Lashkia, 2002; Smillie, 1976; Cox and Snell, 1974), alternative techniques for determining causation (Martel, 2000; Vineis, 2003), and methods for handling conflicts among data sets in order to identify other, perhaps more appropriate approaches to this problem. Third, many of today's web applications use *Cascading Style Sheets* (CSS) to render content. We are extending our model to incorporate CSS. More specifically, the gold standard that we are currently using for HTML tags also features CSS element compliance data. Subsequently, we could apply CSS conditioning rules (analogous to the HTML tag conditioning rules) and calculate coefficient correlation values for these features in order to derive knowledge of CSS support in client configurations.

As we continue to study an increasing number of web applications and execute our algorithms on larger data sets, we intend to address some 'engineering' issues. While, in principle, our technique is able to handle dynamically generated web pages (*i.e.*, a user has to traverse the page via a browser, extract its source HTML and submit it to ICE), our automated crawler handles only static pages (or those generated using parameters already encoded in a URL).

Finally, while building knowledge bases by mass collaboration can greatly reduce the time and cost of developing large knowledge bases, several issues surface as a result including quality, consistency and relevance in submitted information; scalability of learning algorithms; and motivation level of participants (Richardson and Domingos, 2003). Since the knowledge derived is heavily dependent on examples, it is important that users categorise web applications accurately, that examples provide relevant bases for inference, that the algorithm can handle a large number of contributions from thousands of participants, and that users are properly motivated to participate. We will address such issues in the future in order to maximise the potential of this system as a whole, and the learning mechanism in particular.

Our long-term research plans include:

- Studying the effect of diverse *server* configurations on web application correctness; for example, if the configuration of a server hosting a web application changes, then what (if any) is the impact on the correctness of the hosted application?
- Studying the effect of client/server configurations on sequences of user interactions with the application.
- Classifying the defects that are found by our technique; currently we cannot detect non-tag related defects. For example, we cannot detect if a Javascript program in a web application executes incorrectly in certain configurations.

All in all, by addressing the issues we have identified as future work and refining our model, we believe that we can provide an extremely effective approach for detecting the browsing-environment-specific issues that threaten universal usability.

Acknowledgements

The authors thank the anonymous reviewers of this paper whose feedback and comments played an important role in reshaping the experimental results and analyses. This work was partially supported by the David and Lucile Packard Foundation, the American Association for the Advancement of Science (AAAS), the US National Science Foundation under NSF grant CCF-0447864, and the Office of Naval Research grant N00014-05-1-0421.

References

Bellettini, C., *et al.* (2005) 'TestUml: user-metrics driven web applications testing', *SAC '05: Proceedings of the 2005 ACM Symposium on Applied Computing*, New York, NY, USA: ACM Press, pp.1694–1698.

Berghel, H. (1996) 'HTML compliance and the return of the test pattern', *Communications of the ACM*, Vol. 39, No. 2, pp.19–22.

Brajnik, G. (2004) 'Using automatic tools in accessibility and usability assurance processes', *LNCS Proceedings of the 8th ERCIM Workshop on User Interfaces for All*, Springer, pp.219–234.

Cox, D.R. and Snell, E.J. (1974) 'The choice of variables in observational studies', *Applied Statistics*, Vol. 23, No. 1, pp.51–59.

Dávila-Nicanor, L. and Mejía-Alvarez, P. (2004) 'Reliability improvement of web-based software applications', *4th International Conference on Quality Software (QSIC 2004)*, IEEE Computer Society, pp.180–188.

Deshpande, Y. and Murugesan, S. (2001) 'Summary of the Second ICSE Workshop on Web Engineering', *SIGSOFT Software Engineering Notes*, Vol. 26, No. 1, pp.76–77.

Eaton, C. and Memon, A. (2004a) 'Evaluating web page reliability across varied browsing environments', *Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04)*, Saint Malo, Bretagne, France.

Eaton, C. and Memon, A. (2004b) 'Improving browsing environment compliance evaluations for websites', *Proceedings of the International Workshop on Web Quality (WQ'04)*, Munich, Germany.

Gaur, N. (2000) 'Assessing the security of your web applications', *Linux Journal*, Vol. 2000, Issue 72es, Article no. 3; ISSN 1075-3583.

Hansen, S. (2002) 'Web information systems: the changing landscape of management models and web applications', *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, ACM Press, pp.747–753.

Holmes, J.H. (1997) 'Discovering risk of disease with a learning classifier system', *International Conference on Genetic Algorithms*, Morgan Kaufmann, pp.426–433.

Huang, Y-W., *et al.* (2003) 'Web application security assessment by fault injection and behavior monitoring', *Proceedings of the Twelfth International Conference on World Wide Web*, ACM Press, pp.148–159.

Kallepalli, C. and Tian, J. (2001) 'Measuring and modeling usage and reliability for statistical web testing', *IEEE Trans. Softw. Eng.*, Vol. 27, No. 11, pp.1023–1036.

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

Lashkia, G.V. (2002) 'Learning with relevant features and examples', *International Conference on Pattern Recognition*, pp.68–71.

Lewallen, S. and Courtright, P. (1998) 'Epidemiology in practice: case-control studies', *Community Eye Health Journal*, Vol. 11, No. 28, pp.57–58.

Lucca, G.A.D. and Penta, M.D. (2003) 'Considering browser interaction in web application testing', *Proceedings of the 5th International Workshop on Web Site Evolution*, IEEE Computer Society, pp.74–81.

Martel, I. (2000) 'Probabilistic empiricism: in defence of a Reichenbachian theory of causation and the direction of time', *Thesis*, University of Colorado.

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

Offutt, J. (2002) 'Quality attributes of web software applications', *IEEE Software*, Vol. 19, No. 2, pp.25–32.

Pfeiffer, D. and Morris, R.S. (1994) 'Comparison of four multivariate techniques for causal analysis of epidemiological field studies', *Proceedings of the 7th International Symposium on Veterinary Epidemiology and Economics*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp.165–170.

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

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

Richardson, M. and Domingos, P. (2003) 'Building large knowledge bases by mass collaboration', *K-CAP '03: Proceedings of the International Conference on Knowledge Capture*, ACM Press, pp.129–137.

Seaman, S.R. and Richardson, S. (2001) 'Bayesian analysis of case-control studies with categorical covariates', *Biometrika*, Vol. 88, No. 4, pp.1073–1088.

Smillie, K.W. (1976) 'Regression analysis: theory and computation', *Proceedings of the Eighth International Conference on APL*, pp.401–407.

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

Tonella, P. and Ricca, F. (2004) 'A 2-layer model for the white-box testing of web applications', *Proceedings of the 6th International Workshop on Web Site Evolution*, IEEE Computer Society, pp.11–19.

Vineis, P. (2003) 'Causality in epidemiology', *Soz Praventiv Med*, Vol. 48, No. 2, pp.80–87.

Xu, L., *et al.* (2003) 'A browser compatibility testing method based on combinatorial testing', *International Conference on Web Engineering*, Springer, pp.310–313.

Xu, L., *et al.* (2005) 'Testing web applications focusing on their specialties', *SIGSOFT Softw. Eng. Notes*, Vol. 30, No. 1, p.10.

Xu, L.X.B. (2004) 'A framework for web applications testing', *International Conference on Cyberworlds*, pp.300–305.

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

Notes

1 http://www.netmechanic.com/browser-index.htm

2 http://www2.imagiware.com/RxHTML/

3 http://www.watchfire.com/products/Webxm/bobby.aspx

4 http://en.wikipedia.org/wiki/Web_application

5 http://www.w3.org/QA/Tools/

6 http://www.w3.org/2001/di/