Improving Web Application Testing Using Testability Measures

Nadia Alshahwan[1],**Mark Harman**[1],**Alessandro Marchetto**[2] and **Paolo Tonella**[2]
[1]Kings College London–CREST Centre,Strand, London, UK
[2]Fondazione Bruno Kessler–IRST, Trento, Italy
{Nadia.Alshahwan,Mark.Harman}@kcl.ac.uk,{Marchetto,Tonella}@fbk.eu

Abstract

One of the challenges of testing web applications derives from their dynamic content and structure. As we test a website, we may discover more about its structure and behaviour. This paper proposes a framework for collection of testability measures during the automated testing process (termed 'in-testing' measure collection). The measures gathered in this way can take account of dynamic and content driven aspects of web applications, such as form structure, client-side scripting and server-side code. Their goal is to capture measurements related to on-going testing activity, indicating where additional testing can best lead to higher overall coverage. They denote a form of 'web testability' measures. The paper reports on the implementation of a prototype Web Application Testing Tool, WATT, illustrating the in-testing measure collection approach on 34 forms taken from 14 real world web applications.

Keywords: Web Testing, Page Coverage, Testability Measures, Automated Testing, Crawlers.

1. Introduction

As web systems evolve there is a need to test the revisions made to the web site. The available test cases need to evolve as the website evolves in order to provide robust mechanisms for quality assurance and trust. However, unlike more traditional systems, in the evolution of web systems and their test cases, the task of generating test cases to cover the existing and new functionality and the exploration of the web system's structure are closely interwoven activities.

That is, the process of understanding the structure of a website (through, for instance, spidering the system) is closely related to the task of seeking to cover the system for achieving test coverage. Furthermore, the task of achieving coverage while testing web based systems presents many important challenges not found in the testing of conventional systems [2, 18]. Web systems are highly dynamic; testing must take account of coverage of form structure, client-side scripting and server-side code, all of which have a bearing on the behaviour of the web system under test.

Previous work on testing web based systems, has focused on the task of generating test cases to achieve coverage [8, 15]. While this is important, the special features of web based systems tend to suggest that the coverage to be expected from automated testing will not be complete; it can be expected to be no better than (and perhaps somewhat worse than) those levels of coverage obtainable for more conventional systems. Indeed, there is recent evidence that, even state–of–the–art test data generation systems for conventional 3^{rd} generation code are only capable of achieving relatively modest levels of coverage at the interprocedural level [12].

Therefore, this paper takes as a starting point, the assumption that automated test data generation for web systems, though it will be a useful tool in the overall toolbox, will not provide a complete solution to the testing of web based systems. As a result, there will be a need for on–going testing activity after an initial automated phase of test data generation has 'done the best it can'. This underlying assumption motivates our 'in–testing' approach, in which we seek to measure the system as we perform test data generation. We call this approach an 'in testing' measure collection approach.

The 'in testing' approach collects a set of facts concerning the properties of the web based system as it attempts to cover it using automated test data generation. These facts are stored to a database, forming the building blocks for the computation of measurements concerning the system under test. We term the approach 'in testing' because the measurements are computed during the progress of testing, rather than before the testing starts (as with predictive measurements of conventional systems) or as a measurement of the effectiveness of the completed testing process.

We set out a framework for collection of facts, on which metrics can be built (strictly speaking the values we compute in this paper are measurements, not metrics, since they are not normalized). The collection of these facts can be used to support the computation of a wide range of metrics, not merely those associated with testing. However, in

this paper we focus on testability and test–related measurements.

We report on a tool WATT, which provides an initial implementation of the proposed 'in testing' approach. WATT spiders a website, collecting facts as it goes, storing them to a database. The spidering process uses random test data generation to fill in the forms it encounters. The database also provides default values that can be used as entries to web forms to assist the spidering process in maximizing coverage.

However, like any automated coverage based web testing tool, WATT cannot be expected to cover the entire structure of the website. Where WATT differs from previous web testing approaches [4, 15, 16] is in the collection of facts which can be used to compute metrics to guide additional testing towards those parts of the website that may have remained uncovered. In this paper we define five simple measurements that can be computed from these facts to illustrate the way in which WATT supports in-testing measurement of testability.

We configured WATT to compute these 5 measures and applied it to 34 forms drawn both from currently popular websites, such as Facebook, Google, YouTube and also from commercial downloadable web applications such as osCommerce, softslate and Open Project Manager. As a preliminary study of these measurements, a human tester (one of the four authors, who was least closely involved with the measures' definition) categorized the web systems' forms according to whether they were easy or hard to test. The human categorization was based purely on expert judgement and was not informed in any way by the measures. We used the combination of automatically computed in-testing measurements and this human judgement to pinpoint some interesting observations concerning the potential application of the WATT tool and the in-testing approach.

The primary contributions of the paper are as follows:

1. We introduce the in-testing approach to integrating and interweaving the process of test data generation and the computation of testability measurements for web application testing.
2. We introduce a preliminary prototype tool, WATT that implements this approach. Currently WATT is implemented only for client side testing. We are in the process of extending it to handle server side testing and testability assessment.
3. We introduce five simple client side testability measurements and report on our experience in applying WATT to compute these measurements on several widely used websites and web applications. Though the results are preliminary at this stage, they provide some evidence to suggest that the in-testing approach can collect useful information during automated web testing that can be used to define measurements that can inform the on-going testing process.

The rest of the paper is organized as follows: Sections 2 and 3 present the proposed testing framework and the aspects we want to measure. Section 4 defines the testability measures, whilst Section 5 presents our experimentations with the measures together with a discussion of the results. Section 6 presents related work and Section 7 concludes.

2. Proposed Testing Framework

In dynamic web applications, the content is generated and formed based on user choices and input values mainly provided by means of HTML forms. To explore larger parts of the application and to achieve more coverage during testing, each form is provided with a set of input values that aim to generate pages from which user interaction can proceed. As a consequence, meaningful input values are important to produce during testing.

Testing tools that aim at page coverage are typically based on web crawling. Given a base URL, they automatically navigate links starting from that URL and use automated input generation techniques (e.g. predefined database of inputs) to process forms. The primary three problems with these tools are that: (i) not all forms can be automatically filled by such tools; and (ii) after running them we do not precisely know how much application coverage has been achieved; (iii) it is not clear which application inputs should be changed in order to increase such a coverage. Moreover, during testing the test tool in certain areas of the application makes decisions that could affect page coverage. Directing the user to these areas could be useful for improving the quality of testing and achieving higher coverage while saving time and resources.

We propose a framework that provides the tester with this kind of information. The framework points the tester to certain areas (pages and forms) of the application that could possibly lead to new unexplored pages. Furthermore, it guides the user by better focusing and limiting the manual effort spent during the testing activity. This information consists of a set of proposed testability indicators that measure specific aspects of each application form that could impact application coverage and testing effort.

Figure 1 summarizes the most relevant activities of the proposed framework. The URL for the starting page of the web application under test is provided by the user and crawling starts from that point. The crawler will download the web page and identify any forms it contains. Input values are automatically generated (by mixing random values and input taken from a predefined database, when available) for these forms and test cases are created. The crawler then resumes crawling using those created test cases and the process is repeated for each encountered page.

The identified forms are analyzed and testability indicators are calculated for each form. A ranking of forms is

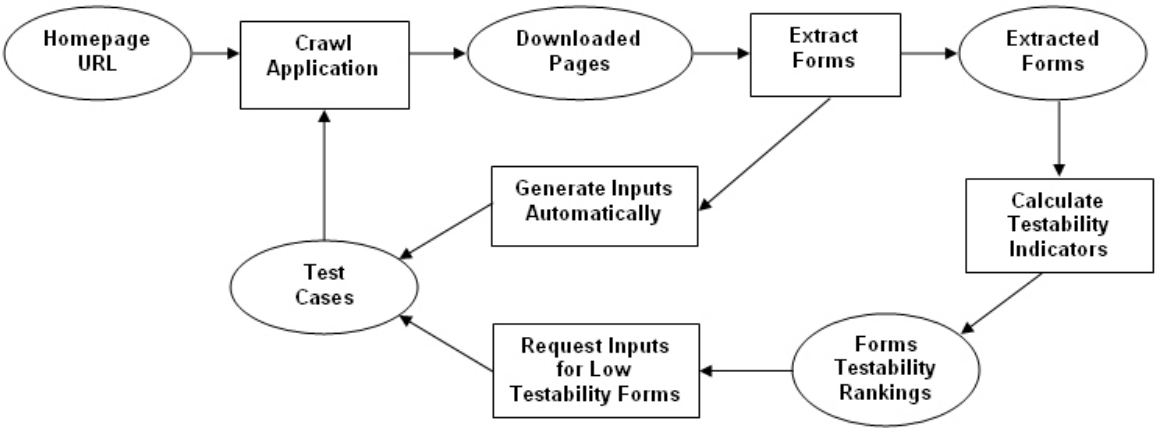

Figure 1. The proposed framework showing the sequence of testing steps taken by the tester

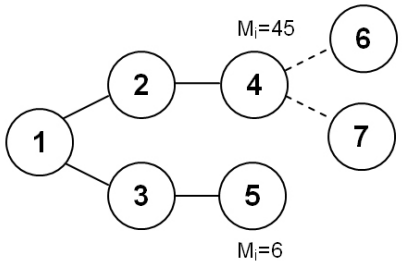

Figure 2. A simple demonstration of how measures can help achieve more coverage

then generated and displayed to the user. Some indicators might prove to be more accurate than others in a given application. This could depend for example on the languages used to implement the application (JavaScript/AJAX rather than JSP/Java Servlet). However, this ranking approach is assumed to give an initial idea of which forms are more suitable candidates for further/manual investigation.

The user is supported in choosing which forms to concentrate on that can lead to higher page and application coverage. Testability measures could indicate that more accurate analysis may be required. The measures also provide some insight into which aspects of these forms need more care. For example, a testability indicator related to a high number of client side validation functions may lead the user to provide more input values for fields that trigger these functions.

The input values provided by the user are then used to generate additional test cases that could lead to new target pages being covered. These pages are processed in the same way returning control to the user when more manual intervention is needed. The process can continue until the user is satisfied or no new pages are encountered.

Figure 2 is a simple demonstration of the result of the approach. Nodes represent pages in the application and edges represent dependencies (due to links and forms) between these pages. A testability measure M_i is calculated for pages with forms such as nodes 4 and 5 in the figure. When this measure indicates low testability (node 4), the tester's attention is directed to this part of the application leading to new pages as indicated by the dashed lines.

3. Aspects Affecting Testability in Web Applications

3.1. Forms

Form inputs affect the generation of dynamic content and sometimes the target pages that can be navigated, thus affecting page coverage. Though a form could have several types of input fields, we can divide those types into two categories: enumerable and unbounded.

Enumerable fields are those fields that in the form definition itself have a list of all possible input values from which the user can choose. Examples of such fields are drop down lists, radio buttons and check boxes. Other types that are slightly different but can be included in this category are buttons and hidden fields since the choices of inputs are limited as well.

A form with enumerable fields has a number of possible input combinations. Different combinations can lead to different target pages and thus if all possible combinations are explored more page coverage could be reached. However, even a simple form with a few enumerable fields could have a large number of possible combinations of inputs. Figure 3 shows a simple form taken from Facebook. Although it has only two inputs, the number of possible choices for the *country* field are 251 and for *estimated budget* are 4, making the number of possible combinations $1,004$.

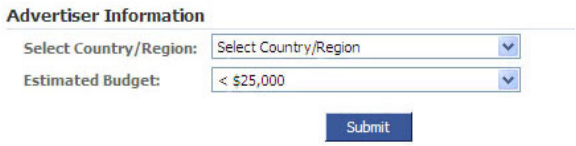

Figure 3. Facebook Form: there are 1,004 possible input combinations for this simple form

It is hard to determine which input combinations lead to new pages that have not been covered. Covering all combinations in forms that have millions of possible combinations could be time consuming. Therefore, a form with a higher number of input combinations could be considered to have lower testability.

Unbounded fields are text based fields that the user has to enter. The possible values are unbounded and therefore are more challenging to automatically generate. Examples of these types are text and password fields and search boxes.

A form with a larger number of unbounded fields could be considered to have lower testability, because generating meaningful values for these fields may be quite hard.

Moreover, these fields could be dependent upon each other. In this case values need to be generated that are both meaningful and properly related. Examples of such hard–to–fill forms are login and credit card details forms.

Clearly, unbounded fields could be sometimes easy to fill such as description or comment fields. Therefore, additional measures about server side validations, database queries and client side validations performed on fields should be defined. These measures could help to better characterize the difficulty of finding meaningful values for unbounded fields. However, the raw number of unbounded fields is itself a testability indicator that we will consider.

3.2. Client Side Validation

Forms or form fields can have a client side script such as JavaScript functions attached to them. These scripts are triggered by certain events such as changing the value of the field or moving the cursor over it.

These scripts could display text to help the user fill out the field or highlight a field to show an invalid input. Other scripts could automatically fill a dependent field when a certain field is filled or can change the possible choices for an enumerable field. Figure 4 shows an example of such a form taken from the ASOS website where changing the selected colour triggers a JavaScript function that changes available sizes for that colour. Dynamic modification of DOM is a more advanced type of client side scripts which may affect also the forms in the page. This happens for instance in AJAX applications, where form elements can be added or removed or new forms can be created based on user actions.

Figure 4. ASOS Form: changing the colour triggers a JavaScript function that changes available sizes

Forms with a higher number of each of these script types can be considered harder to test. We will consider each of these JavaScript function types separately when measuring a form's testability. However, in this work we do not analyze plug-ins and extensions such as applets, flash and ActiveX controls, which are indeed quite hard to test.

3.3. Server Side Manipulation

Inputs Affecting Execution Paths: Server side code generates dynamic pages based on choices made by the user. Input fields can be directly or transitively used in page generation statements. This can be along a high number of different paths in the server code which indicates that pages can be generated in a number of different ways. Each of these paths may need separate test inputs to satisfy.

Inputs Used in Queries: Inputs that are used to query the database need meaningful values imbued with domain knowledge, so that valid results can be returned. Hence, the presence of dependencies between form inputs and database queries is also an indicator of hard to test forms.

Non–User Inputs: Some inputs used on the server side are not directly entered by the user, such as date or location. These parameters can be used to display different pages at different times or customize pages based on the user's location, hence affecting the web application's testability.

In this paper we do not define server side measures more formally, as we do with client side measures. Further investigation of server side testability measurements remains a topic for future work.

4. Defining Testability Measures

In order to define the proposed testability measures unambiguously, we first introduce a web application model, structured into components and relationships between them.

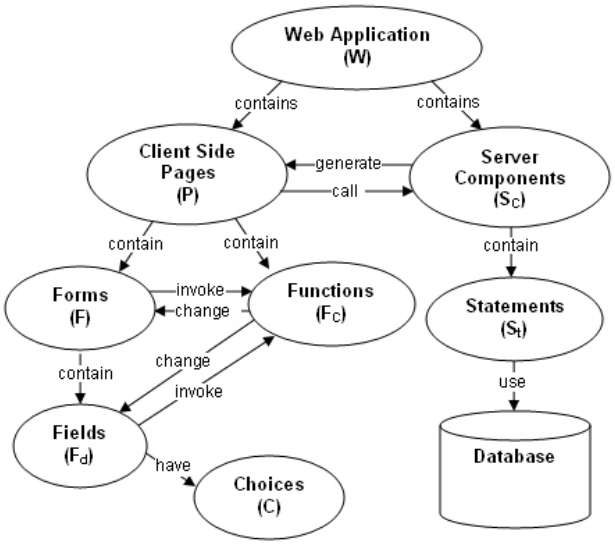

Figure 5. Model of typical web application structure

Measures will refer to these entities. We also introduce auxiliary functions defined upon the model entities, which simplify successive measures definition.

Figure 5 shows a generic web application model. In this model, a web application (W) can be looked at from 2 different perspectives: client side and server side. Client side pages (P) are those that the browser displays to the user navigating the application. Server side components (S_c) are the software components that the web server invokes to generate the client pages. They are also responsible for retrieving persistent data from the database. A server component is made of statements (S_t), these statements in our context can have different types: HTML generating statements, database access statements, control statements and other statements. Client side pages consist of forms (F) and client side scripts such as JavaScript functions (F_c). Forms consist of input fields (F_d). Each field of enumerated type has a finite number of possible input choices drawn from a set C. Both forms and fields can invoke JavaScript functions, which may in turn change existing fields and forms or create new fields or forms, due to the possibility of DOM manipulation available to JavaScript functions.

To improve readability, we use uppercase letters to represent sets and lowercase to represent instances. We first define a web application as a set of client side pages and server components:

$$W :< P, S_c >$$

We define a function forms, that retrieves all forms in a client side page:

$$\mathsf{forms} : P \rightarrow 2^F$$

We define a function fields, which maps forms to the set of fields that occur in the form:

$$\mathsf{fields} : F \rightarrow 2^{F_d}$$

We define a function choices that returns the set of possible input values defined in the HTML code for a field or $\perp$ if the field is unbounded. The function is defined as follows:

$$\mathsf{choices} : F_d \rightarrow 2^C \cup \{\perp\}$$

$$\mathsf{choices}(f_d) = \begin{cases} \{c_1, \ldots, c_n\}, & \text{if } f_d \text{ is enumerable} \\ \perp, & \text{if } f_d \text{ is unbounded} \end{cases}$$

Where $\{c_1, \ldots, c_n\}$ is the set of possible choices for f_d.

We define a function unbounded to return those fields which have an unbounded number of possible input values (i.e. free text inputs such as password, text fields and search boxes):

$$\mathsf{unbounded} : F \rightarrow 2^{F_d}$$

$$\mathsf{unbounded}(f) = \{f_d \in \mathsf{fields}(f) \mid \mathsf{choices}(f_d) = \perp\}$$

We define a function enumerable to return those fields F_d which are enumerable (i.e. not unbounded):

$$\mathsf{enumerable} : F \rightarrow 2^{F_d}$$

$$\mathsf{enumerable}(f) = |\mathsf{fields}(f)| - |\mathsf{unbounded}(f)|$$

We define functions to be a function that returns the set of JavaScript event handler functions, associated with a restricted set of relevant events, which are attached to form elements (e.g. input fields, buttons, etc.), excluding special cases, such as handlers performing unchecked form submission in response to mouse click:

$$\mathsf{functions} : F \rightarrow 2^{F_c}$$

4.1. Testability Measures

We now introduce the proposed testability measures, to be calculated for each form in the application. We shall define each measure by giving it a name, an acronym, a short description, a formula and a usage example. Table 1 is a simple form example that we will use to illustrate our measures. The code represents a simple form that allows the user to enter his/her name and allows him/her to provide information about their sex and marital status. Selecting *Male* as sex will reset (by a JavaScript function) the maiden name text box to *none*.

4.1.1 Field values

Unbounded fields can be more challenging for automated input generation. We define a function NUF (Number of Unbounded Fields) that returns the number of unbounded fields in a form as follows:

$$NUF : F \rightarrow \mathbb{N}$$

$$NUF(f) = |\mathsf{unbounded}(f)|$$

```
1 <SCRIPT TYPE="text/javascript">
2 <!--
3 function checksex()
4 {if(document.form.sex[0].checked)
5 {form.maidenname.value = none;}}
6 --></SCRIPT>
....
7 <form name="form" method="GET" action="Srv">
8 <input type="text" name="name">
9 <input type="text" name="maidenname">
10 <input type="radio" name="sex" value="M"
 checked onclick="checksex()"> Male <br>
11 <input type="radio" name="sex" value="F" >
 Female<br>
12 <input type="checkbox" name="Married"
value="Married"> Married<br>
13 <input type="submit" value="Submit"><br>
14 </form>
....
```

Table 1. Simple form used to demonstrate testability measures

For our example the number of unbounded fields is 2, so the value for NUF would be calculated as follows:

$$NUF(form) = |\mathsf{unbounded}(form)| = 2$$

A form with a larger number of possible combinations of inputs is harder to test exhaustively or in a way that would ensure high coverage of t-way combinations. We define a function EFC (Enumerable Field Combinations) that calculates the number of combinations of choices for enumerable fields. We define the function as follows:

$$EFC : F \to \mathbb{N}$$
$$EFC(f) = \prod_{f_d \in \mathsf{enumerable}(f)} |\mathsf{choices}(f_d)|$$

In our example there are two enumerable fields, *Sex* and *Married*. The number of possible choices for *Sex* is 2 (*M* and *F*) and for *Married* is also 2 (*checked* and *unchecked*). EFC for this form will be calculated as follows:

$$EFC(form) = \prod_{f_d \in \mathsf{enumerable}(form)} |\mathsf{choices}(f_d)|$$
$$= 2 \times 2 = 4$$

4.1.2 Client Side Validation

Forms and form fields that invoke JavaScript functions that change the interface need more attention in input generation, to fully test these functions. We differentiate between the following types of JavaScript functions:

- *Type 1* JavaScript functions that modify only elements related to text and display.
- *Type 2* JavaScript that modifies existing fields in existing forms and possibly elements related to text and display.
- *Type 3* JavaScript that modifies the structure of forms by adding or changing fields or forms and possibly elements related to text and display and existing fields.

JavaScript functions that do not manipulate the interface such as logging a request are not considered.

We define a function *CSS* (Client Side Scripts) that measures the testability in terms of JavaScript functions. We define the function as follows:

$$CSS : F \to \mathbb{N}$$

More specifically, we define this measure for each type of JavaScript code mentioned above as follows:

$$CSS_1(f) = |\{f_n \in \mathsf{functions}(f) \mid f_n \; of \; type_1\}|$$
$$CSS_2(f) = |\{f_n \in \mathsf{functions}(f) \mid f_n \; of \; type_2\}|$$
$$CSS_3(f) = |\{f_n \in \mathsf{functions}(f) \mid f_n \; of \; type_3\}|$$

In our example we only have one JavaScript function of *Type 2* where function *checksex()* changes the value of an existing field *maidenName*. Therefore the value of the three measures will be as follows:

$$CSS_1(form) = |\{f_n \in \mathsf{functions}(form) | f_n of type_1\}| = 0$$
$$CSS_2(form) = |\{f_n \in \mathsf{functions}(form) | f_n of type_2\}| = 1$$
$$CSS_3(form) = |\{f_n \in \mathsf{functions}(form) | f_n of type_3\}| = 0$$

5. Experimentation with the Approach

With the aim of conducting a preliminary framework evaluation and refinement, we performed the following steps:

1. Selected applications and forms to be evaluated.
2. Applied WATT to those forms to collect the presented measures: number of unbounded fields (NUF), number of possible combinations (EFC), and the number of each of the three types of JavaScript functions attached to form elements (CSS_1, CSS_2 and CSS_3).
3. An expert tester (a different author not involved in the framework definition) tested and evaluated each form as 'hard' or 'easy' to test (see below).
4. Observed and compared the obtained measures to the expert evaluation.

5.1. WATT Tool

We developed a prototype of the Web Application Testability Tool (WATT). The tool supports our framework by crawling an application automatically, collecting data about the application's client-side structure, and calculating testability measures.

The tool has the ability to automatically fill and submit forms using a mixture of random input values and user provided values. The tool also parses JavaScript and extracts links. It also manages cookies and user sessions. While crawling the site, the tool generates a parse tree of the downloaded HTML files and extracts different components and their structure such as forms and JavaScript and stores them to the database. The structure of the application and how its pages are connected is also stored and can be used for further analysis later. The tool can also determine which encountered links/forms failed by capturing the response code generated by the server. While crawling, JavaScript functions attached to form elements are parsed for keywords to determine their type. The result is then manually reviewed to insure accuracy. A filter can be provided to limit the crawler to a certain path and sub directory.

The tool was developed and tested based on the completeness criteria and the categorization of Hypertext links proposed by Girardi et al. [7].

In this experiment we limit the WATT tool to the application pages that contain the selected forms.

5.2. Form Selection and Classification

To apply the tool, we selected 34 forms taken from 14 real world web applications. The forms were taken from a number of well known sites such as Facebook, Yahoo and YouTube and commercial downloadable applications such as Softslate, osCommerce and Zen Cart. These forms have been selected to cover different types of forms such as login, registration, search and order forms. They were also selected to have different structural sizes and complexity.

An expert tester has been asked to classify these forms as 'easy' or 'hard' to test. The tester analyzed and tested each form then annotated the main activities applied during this task as follows:

- Analysis of the possible combinations of the form fields; each form has been exercised by using ad-hoc coverage criteria for field combinations with the purpose of discovering new application parts/behaviours.
- Inspection and evaluation of field-attached actions, in both client and server-side code (e.g. JavaScript checks related to fields, links to server components). Application components that impact the form/page have been manually inspected. This was done to understand whether their behaviour on the form impacts the application exploration.
- Evaluation of the random exploration of the form and the required manual intervention. In this case, each form has been exercised by using only random inputs with the aim of evaluating how much the form can be automatically explored.
- Analysis of documentation, help, and information contained in the page about how to fill and activate the form.
- Evaluation of the overall time required to study the form fields with the aim of selecting an adequate set of inputs required to exercise the form during testing.

According to this evaluation each form has been classified as 'easy to test' when it requires little testing effort to be 'reasonably' tested and 'hard to test' if a lot of effort has been required. This classification is made entirely by human intuition, based on the above guidelines.

5.3. Results

We ran the tool on our selected forms and collected the results presented in Table 2. The results show a somewhat consistent relation between the number of unbounded fields and human classification of the forms. One exception is Web Calculator, where the form was classified as hard while it only contains one unbounded field. This is because the form has a large number of buttons that operate on this field. However, this is captured in the number of JavaScript functions attached to this form which is 35 (CSS_2). Most of the forms rated easy to test have a low number of unbounded fields. *Google's Advanced Search* form is an exception. For a human tester it is easy to classify this form as easy to test since any random data could be used to submit the form successfully. However, this observation requires additional domain knowledge that can not be available to a measurement.

The *Venere Hotel Search* form has a number of events that perform client side JavaScript validations on fields. One example is to check if the selected 'From Date' is less than or equal to the selected 'To Date'. This was captured by the CSS_2 measure.

The possible number of input combinations seems to have not affected the human rater's judgement of 'subjective testability'. For instance, *Softslate's Address Settings* form has the largest number of possible combinations in the set. However, the input fields consist of 92 check boxes and this is what causes the possible number of combinations to be high. Choosing combinations that represent the typical application behaviour we can encounter is sufficient to achieve high coverage in this case. However, this is not always an easy task especially when testers have no special knowledge about the application under testing. Therefore, this measurement could indicate to the tester that due to the large number of possible combinations, the form needs to be examined to determine which combinations need to be tested. This leads us to conclude that this measure needs to be combined with other measures to be a better indicator of testability. One way to refine this measure could be analysis of the server side code to determine how many different combinations lead to different behaviour.

We observe in Table 2 that JavaScript is not as frequently attached to the form fields as we expected. This could be caused by the fact that JavaScript is a client side scripting language. Checks on input values in forms on the client side

Website	Form Name	# Fields	NUF	EFC	CSS_1	CSS_2	CSS_3
Easy to test for human testers (as judged subjectively)							
Google	Advanced Search	17	7	1.5×10^8	0	0	0
Last Minute	Hotel Search	20	0	5.8×10^{18}	0	4	14
BBC	Display Options	39	0	1.7×10^4	0	0	0
You Tube	Search	1	1	1	0	0	3
Facebook	Login	5	3	2	1	0	0
Facebook	Registration	10	4	1.4×10^5	1	0	3
Venere	News letter	1	1	1	0	1	0
Venere	Hotel Search	7	1	2.1×10^7	0	6	0
Wikipedia	Login	5	2	2	0	0	0
Wikipedia	Search	3	1	1	0	0	0
Softslate	Search	2	1	1	0	0	0
Yahoo	Login	4	2	2	0	0	0
Softslate	Add to Cart	10	9	1	0	0	0
Softslate	Register	4	3	1	0	0	0
Softslate	Product Settings	28	0	1.3×10^8	0	1	0
Softslate	Discount Settings	10	0	32	0	0	0
Softslate	Address Settings	92	0	5×10^{27}	0	0	0
Open Project	Add Task	12	0	5.3×10^6	0	2	0
Open Project	Button Settings	30	0	5.4×10^8	0	0	0
Open Project	Create Button	10	0	2,400	1	0	0
Open Project	Edit States	34	0	5.4×10^{11}	0	0	0
Zen Cart	Manufacturers Form	1	0	8	0	0	0
Zen Cart	Filter Search Results	2	0	111	0	0	0
Zen Cart	Shopping Cart	4	1	2	0	0	0
osCommerce	Checkout Address	3	0	2	0	0	0
osCommerce	Checkout Payment	7	2	240	0	0	0
Hard to test for human testers (as judged subjectively)							
TedLab	Web Calculator	36	1	35	0	35	0
Easy Chair	Registration	5	4	1	0	0	0
Softslate	Login	3	2	1	0	0	0
Softslate	Checkout	30	20	4.2×10^9	0	1	0
Softslate	Payment	7	3	480	0	0	0
Softslate	Checkout Settings	14	8	12	0	0	0
Zen Cart	Create Account	24	15	4.3×10^5	0	0	1
osCommerce	Create Account	19	14	960	0	0	0

Table 2. Measure values for the forms used in the experiment

can be easily bypassed causing a security threat. This could cause developers to prefer server side input validation while using JavaScript to enrich the application GUI.

We also notice that the form size and complexity is only partially tied with the judgement of the human tester. For instance, we can consider *Facebook Registration* and *Softslate Payment*. They are similar in terms of number of fields and JavaScript but differently classified as easy (*Facebook Registration*) and hard (*Softslate Payment*) to test. This is primarily because of different input validation checks applied to their fields. In fact, in the *Facebook Registration* form there are a few simple checks (e.g. email format) while in *Softslate Payment* strong checks related to credit card validation are performed (in both client and server side).

Another example is *Softslate Checkout* compared to *Softslate Payment*. In this case, we see that both forms were classified as hard to test by the human while they are different in terms of number of fields and their combinations. In particular, *Checkout* has 30 fields and 4.2×10^9 combinations while *Payment* only has 7 fields and 480 combinations. The difference lies in the validation checks applied to each field of these forms. For *Checkout*, there are a few checks (e.g. email format, ZIP code format) while in *Payment* there are some more complicated checks related to credit card validation.

Furthermore, we can compare *Softslate Register* and *Easy Chair Registration*. Although their measures are comparable, they were classified as easy and hard to test by the human rater respectively. The main reason is that in *Easy Chair Registration* there is a 'CAPTCHA' field that must be manually transcribed by the tester.

Of course, the obtained results are preliminary in nature and cannot be generalized. However, they are encouraging and indicate that with more refinement, measures could be proved to be helpful in testing. In future work, a larger scale more developed evaluation of the approach and analysis of server side related measures will be conducted.

6. Related Work

Several testing web techniques and tools have been proposed as a result of the increased pervasiveness of web applications. This increased pervasiveness demands high quality applications with low defectiveness.

Functional testing is the most widely applied testing approach. Existing tools for web applications (e.g. LogiTest, Maxq, Badboy), are based on capture/replay facilities: they record the interactions that a user has with the graphical interface and repeat them during regression testing. An alternative is based on tools such as HttpUnit. HttpUnit is a Java API that provides the building blocks required to emulate the browser's behaviour. When combined with a framework such as JUnit, HttpUnit allows testers to create test cases to verify Web Application behaviour [10].

Model-based testing of web applications was initially proposed by Ricca and Tonella [15] and then refined by others. This testing approach performs a preliminary analysis of the application under test with the aim of describing it by means of a model (often, it describes web pages, links and forms). Coverage criteria are applied to the model to extract test case suites. Recently, model-based testing approaches have been applied to web 2.0 applications [13, 14]. These approaches extract test cases by studying the client-side behaviour of the application.

Elbaum et al. [8] propose a web testing a web testing approach that uses data captured in user sessions to create test cases automatically. Alshahwan and Harman [1] proposed a session data repair approach to be used in regression testing.

A test case for a web application can be viewed as a sequence of web pages, enriched by inputs and user actions performed through the GUI. The goal of a test case is to emulate an execution in which the expected and real application behaviour are compared. One of the most relevant difficulties in web testing is that a lot of manual intervention is often required to fully test the application. In fact by applying only automatic testing criteria there is no way to guarantee that a web application is 'completely' covered (e.g. all links, pages and forms).

Testability metrics have been used and defined for traditional software such as Object Oriented software for predicting testing effort. For instance, Bruntink and Deursen [5, 6] evaluated and defined a set of testability metrics for Object Oriented programs and analyzed the relation between classes and their JUnit test cases. Jungmayr [11] suggests that testability metrics can be used to identify parts of the application causing a lower testability level by analysing test critical dependencies.

In web applications, metrics have been used especially for maintainability and evolution. Emad Ghosheh et al. [9] compare a number of papers that define and use web maintainability metrics. These are mostly source code metrics that predict maintainability of web applications. Warren et al. [17] created a tool to collect a number of metrics to measure web application evolution over an interval of time. In the context of using metrics to aid testing, Bellettini et al. [3] created a tool TestUML that combines a number of techniques to semi-automatically test a web application. Metrics such as number of pages or number of objects were used to define coverage level and then be used for stopping the testing process based on user criteria.

The framework we propose combines the existing web testing approaches with testability measures. This would help to prioritize parts of the application with low testability during the testing process. Combining testing with testability measures would help the tester identify areas of the application that need more attention thereby decreasing the testing effort.

The primary novelty of the in-testing approach reported

on in this paper is that measures are used during the testing process to guide the tester to areas where efforts could be most effective.

7. Conclusion and Future Work

In this paper we have introduced an approach to testing web based systems that is integrated and interwoven with the process of computing measurements of the system under test. The approach collects facts about the system under test as it spiders over the site, entering data for forms and seeking to cover structure. These facts are used to compute measurements. While the measurements collected during testing can report on any aspect of the system, it is natural to focus on testability, especially since web based systems pose particular challenges for testing.

We report on an implementation, WATT, a system that implements the 'in testing' computation of testability measures that we advocate. We present results from initial experiments with five simple measurements, computed using WATT for 14 web based systems, comprising of well-known web sites and downloadable web applications. Although the results are preliminary, they highlight some interesting features of the systems under test and the way in which measures can be used to draw the tester's attention to them.

Much remains to be done to develop this research agenda. Future work will focus on the development of additional measurements at the server side to complement those presented for client side testing in this paper. More work is also required to evaluate testability transformation measurements for web based testing. This work will explore the correlation between testability measurements and the behaviour of and effort required by test data generation approaches. We plan to further develop the WATT testing tool to achieve these goals for future work and to release it for wider research use.

References

[1] N. Alshahwan and M. Harman. Automated session data repair for web application regression testing. In *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 298–307, Washington, DC, USA, 2008. IEEE Computer Society.

[2] J. O. Anneliese A. Andrews and R. T. Alexander. Testing web applications by modeling with fsms. *Software and Systems Modeling*, 4:326–345, 2005.

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

[4] M. Benedikt, J. Freire, and P. Godefroid. Veriweb: Automatically testing dynamic web sites. In *Proceedings of 11th International World Wide Web Conference*, Honolulu, Hawaii,USA, 2002.

[5] M. Bruntink and A. van Deursen. Predicting class testability using object-oriented metrics. In *SCAM '04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, pages 136–145, Washington, DC, USA, 2004. IEEE Computer Society.

[6] M. Bruntink and A. van Deursen. An empirical study into class testability. *J. Syst. Softw.*, 79(9):1219–1232, 2006.

[7] F. R. Christian Girardi and P. Tonella. Web crawlers compared. *International Journal of Web Information Systems*, 2:85–94, 2006.

[8] S. Elbaum, S. Karre, and G. Rothermel. Improving Web application testing with user session data. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 49–59, Portland, USA, May 2003. IEEE Computer Society.

[9] E. Ghosheh, J. Qaddour, M. Kuofie, and S. Black. A comparative analysis of maintainability approaches for web applications. In *AICCSA '06: Proceedings of the IEEE International Conference on Computer Systems and Applications*, pages 1155–1158, Washington, DC, USA, 2006. IEEE Computer Society.

[10] E. Hieatt, R. Mee, and G. Faster. Testing the web application engineering internet. *IEEE Software*, 19(2):60–65, March/April 2002.

[11] S. Jungmayr. Testability measurement and software dependencies. In *Proceedings of the 12th International Workshop on Software Measurement*, pages 179–202, Aachen, 2002. Magdeburg,Shaker Publ.

[12] K. Lakhotia, P. McMinn, and M. Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In 4^{th} *Testing Academia and Industry Conference — Practice And Research Techniques (TAIC PART'09)*, Windsor, UK, 4th–6th September 2009. To appear.

[13] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of ajax web applications. In *International Conference on Software Testing Verification and Validation (ICST)*, Lillehammer, Norway, April 2008. IEEE Computer Society.

[14] A. Mesbah, , and A. van Deursen. Invariant-based automatic testing of ajax user interfaces. In *31st International Conference on Software Engineering (ICSE)*. IEEE Computer Society, May 2009.

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

[16] F. Ricca and P. Tonella. Building a tool for the analysis and testing of web applications: Problems and solutions. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 373–388, London, UK, 2001. Springer-Verlag.

[17] P. Warren, C. Boldyreff, and M. Munro. The evolution of websites. In *IWPC '99: Proceedings of the 7th International Workshop on Program Comprehension*, page 178, Washington, DC, USA, 1999. IEEE Computer Society.

[18] Y. Wu and J. Offutt. Modeling and testing web-based applications. Technical report, George Mason University, 2002.