Int J Softw Tools Technol Transfer (2011) 13:131–149
DOI 10.1007/s10009-010-0177-3

WSE 2009

Crawlability metrics for automated web testing

**Alessandro Marchetto · Roberto Tiella ·
Paolo Tonella · Nadia Alshahwan · Mark Harman**

Published online: 22 October 2010

Abstract Web applications are exposed to frequent changes both in requirements and involved technologies. At the same time, there is a continuously growing demand for quality and trust and such a fast evolution and quality constraints claim for mechanisms and techniques for automated testing. Web application automated testing often involves random crawlers to navigate the application under test and automatically explore its structure. However, owing to the specific challenges of the modern Web systems, automatic crawlers may leave large portions of the application unexplored. In this paper, we propose the use of structural metrics to predict whether an automatic crawler with given crawling capabilities will be sufficient or not to achieve high coverage of the application under test. In this work, we define a taxonomy of such capabilities and we determine which combination of them is expected to give the highest reward in terms of coverage increase. Our proposal is supported by an experiment in which 19 web applications have been analyzed.

Keywords Web applications · Crawlers · Crawlability and testability · Measures and metrics · Web testing

A. Marchetto (✉) · R. Tiella · P. Tonella
Fondazione Bruno Kessler, IRST, Trento, Italy
e-mail: marchetto@fbk.eu

R. Tiella
e-mail: tiella@fbk.eu

P. Tonella
e-mail: tonella@fbk.eu

N. Alshahwan · M. Harman
Kings College London, CREST Centre, Strand, London, UK
e-mail: Nadia.Alshahwan@kcl.ac.uk

M. Harman
e-mail: Mark.Harman@kcl.ac.uk

1 Introduction

Web systems evolve rapidly and there is a strong need for providing robust mechanisms for quality assurance and trust. Testing is one of the most wide spread approaches to verify the quality of a Web application. In the literature, several Web testing techniques have been proposed (e.g., [11,32]).

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 navigation structure are closely interwoven activities. Understanding the structure of a Web application, usually performed by means of crawling, is closely related to the task of seeking to cover the system for achieving test coverage. This task is quite challenging, with respect to testing of conventional systems [3,42]. For instance, Web systems are highly dynamic; testing must take into account coverage of the form structure, client-side scripting and server-side code, all of which have a bearing on the behavior of the Web system under test.

Previous works on testing of Web applications have been mainly focused on the generation of test cases to achieve coverage [11,32] in a manual or only partially automated way. The special features of Web applications suggest that the coverage to be expected from fully automated testing (e.g., the one realized by means of automated crawlers) will not be complete. Automatic crawlers may leave large portions of the application under test unexplored. Indeed, there is recent evidence that even state-of-the-art test data generation systems for conventional third generation code are only capable of achieving relatively modest levels of coverage at the interprocedural level [26].

This paper takes as starting points the assumptions that (1) modern Web applications have a limited crawlability degree (intuitively, the crawlability is the property of a Web application to be traversed by an automatic—e.g., random—crawler); and (2) automated test data generation for Web applications (e.g., by means of random-based strategies) will not provide a complete solution to the testing of Web applications.

As a result, there will be the need for further, manual 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 application under test for guiding the user in improving the test data generation only focusing where it is actually needed (e.g., on those forms that can be hardly explored by the automatic crawler). Our IN-TESTING approach collects a set of measures concerning the properties of the Web system that can impact its crawlability as it attempts to cover it using automated test data generation.

In this paper, we propose the use of three structural metrics for predicting whether an automatic (e.g., random) crawler will be sufficient or not to achieve high coverage of the application structure, i.e., to explore all the pages of an application. Furthermore, when the crawler is insufficient, user intervention may be required or more advanced crawling capabilities must be added before asking for the user's help. In this paper, we provide an overview of the most relevant crawler capabilities connected with the application crawlability and we determine which combination of such capabilities is expected to give the highest reward in terms of coverage increase.

An experiment has been conducted on 19 realWeb applications to evaluate (1) the relationship between the proposed metrics and the runtime generated pages to be crawled, so estimating their ability of predicting such number of pages; and (2) the impact of different crawler capabilities on the page crawlability. Overall, we have analyzed a set of 58 HTML forms contained in 51 Web pages (mainly HTML and PHP pages) of the Web applications studied that generate, at runtime, 189 different client-side pages to be crawled during testing. The obtained results are encouraging for our IN-TESTING approach. Some of our metrics are considered strong indicators of crawlability and a limited set of crawler capabilities (i.e., 3/4 well known and simple capabilities) can be used to cover a lot of dynamically generated pages.

In detail, this work extends our previous work [2] with the following contributions:

1. We introduce, for the first time to the best of our knowledge, the notion of *crawlability* of a Web application and the structural aspects of an application that impacts it.
2. We introduce three metrics (with some additional variants for each one) that are proposed to estimate the crawlability of an application and predict whether a crawler with given capabilities is sufficient to explore the application.
3. We present a taxonomy of the possible crawler capabilities expected to be related to the ability of the crawler of exploring the pages of the application.
4. We present our idea of the IN-TESTING framework, integrating and interweaving test data generation, computation of crawlability metrics and human intervention.
5. We present an experiment in which 19 Web applications have been analyzed to evaluate the relationship between the set of pages dynamically generated from the applications with respect to: (i) our metrics, and (ii) the crawler capabilities. Although further investigations are required to strongly confirm the obtained results, they provide some evidence to suggest that our IN-TESTING approach can collect useful measurements during automated Web testing that can be used to inform the ongoing testing process.

The rest of the paper is organized as follows: Section 2 presents the proposed testing framework to give the reader a high-level picture of the research context of this work. Section 3 formally introduces the notion of application *crawlability* and relevant aspects impacting it. Section 4 details the metrics that we propose to evaluate the crawlability during application testing. Section 5 summarizes the results of an experiment performed to evaluate the proposed suite of metrics. Finally, Sect. 6 presents related work and Sect. 7 concludes the paper presenting also our future work.

2 Framework

Testing tools that seek to achieve page coverage are typically based on the Web crawling. Given a base URL, they automatically navigate links starting from that URL and use automated input generation techniques to process forms.

In our previous paper [2], we presented a framework that tries to overcome the three primary problems of these tools: (1) not all forms can be automatically filled by such tools; and (2) after running them we do not precisely know how much application coverage has been achieved; (3) it is not clear which application inputs should be changed to increase such a coverage.

Tooling is very important for the automation of testing-related activities. Without automation, the process of testing becomes slow, costly and error prone. In our work, reported in this paper, we used our the Web Application Testing Tool (WATT), which is described in more detail in the conference version of the present paper [2].

Figure 1 summarizes the most relevant activities of the IN-TESTING 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 downloads the Web

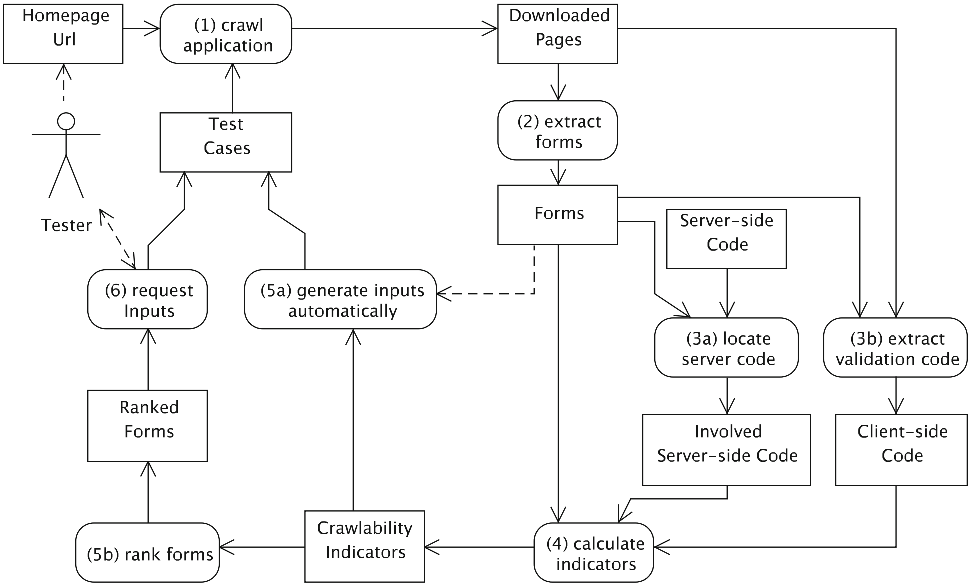


Fig. 1 The IN-TESTING framework showing the sequence of testing steps to be taken by the tester. In the figure: *rectangles* indicate artifacts (e.g., pages downloaded by the crawler); *ovals* represent the steps performed by means of the framework for supporting the application testing; and *arrows* represent the flow of the step execution

page (step 1 in Fig. 1) and identifies any forms it contains (step 2).

During this step, typical Web application testing tools, following the dashed arrow, automatically generate input values (e.g., using random values; using input taken from a predefined database or from previously logged user sessions) for these forms and test cases are created (step 5a). The crawler then resumes crawling using those created test cases and the process is repeated for each encountered page.

In the framework, we are proposing, forms, client-side code (extracted from downloaded pages, step 3b) and involved server-side code (identified in step 3a) are analyzed and crawlability indicators are calculated for each form (step 4). These indicators are then used both for generating a ranking of forms (step 5b) and for improving the automatic generation of inputs (step 5a).

Forms, ranked by estimated crawlability, are displayed to the user, who is asked to fill the lowest crawlability forms with inputs from equivalence classes not considered so far (step 6). 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 automatic input generator can effectively use crawlability indicators to tune the crawling strategy, generating more test cases and making use of more advanced generation capabilities for forms with low values of crawlability.

The following sections provide experimental evidence to support the claim that our approach can be useful in both cases. Of course, it is, like all software engineering metrics measured on any scale between the nominal and absolute scales [36], naturally and inherently an approximate approach. However, as the study illustrates, our metrics, although approximate, can still be useful.

3 Crawlability

Intuitively, the crawlability of a Web application measures how difficult it is for a crawler to explore all the pages reachable from the home page. When such exploration involves the submission of forms, it depends on the ability of the crawler to generate inputs that explore all different pages possibly generated in response to form submission. Therefore, it depends on the input generation capabilities of the crawler. It also depends on the notion of the “difference” between pages generated in response to a form submission; different concrete pages may be merely instances of the same “conceptual” page, differing only in minor irrelevant display details such as the date and time information shown at some location within each instance of the conceptual page.

It is important to distinguish between these minor irrelevant differences and real and substantive differences. Without such distinction, the measurement of crawlability (and coverage) can become significantly skewed. Hence, instead of an absolute definition of crawlability, we give a “parameterized” definition which is relative to two parameters: (1) the conceptual model of the pages to be crawled; and, (2) the crawler’s capabilities:

– *Conceptual Web application model*: we consider a conceptual Web application model containing (at least) the set of conceptual client pages the Web application can provide to the user. A conceptual client page represents an equivalence class of all concrete pages that are conceptually regarded as equivalent. The model may also include the server-side components that generate the conceptual client pages, as well as the *submission* relationship between forms contained in client pages and associated server-side actions.

The UML Conallen model [8] of a Web application is an example of a conceptual model pretty close to the one referred to in this work. Client pages are not distinguished by the concrete content, which may vary from time to time. Rather, they are characterized by their conceptual role in the application. Multiple different concrete pages may be actually instances of the same conceptual page. For example, the concrete content of the result of a search on the Web may vary from time to time, but conceptually there is just one client page showing the *search result*.

- *Crawler's capabilities*: we characterize a crawler by its specific crawling capabilities and such capabilities include the algorithm used to generate input data when forms are to be filled-in to continue exploration of the Web application and the algorithm used to recognize equivalent pages that should not be crawled separately. Examples of common crawler's capabilities, related to input data generation, are random generation of strings or getting input values from existing databases. Common criteria used by crawlers to decide whether a page has already been explored or not (i.e., whether two pages instantiate the same conceptual page) are based on the page name/title or page HTML structure comparison. Crawler's capabilities are further analysed in Sect. 3.3.

3.1 Definition of crawlability

- *DEF 1 [Crawlability of a Web application*]: The crawlability of a Web application is the degree to which a crawler with given capabilities is able to explore all conceptual client pages in the conceptual model of the Web application.
- *DEF 2* [*Crawlability of a form*]: Assuming the conceptual Web application model contains information to associate each form with the conceptual client pages produced in response to form submission, the crawlability of a form is the degree to which a crawler with given capabilities is able to explore all conceptual client pages produced in response to form submission, according to the conceptual model of the Web application.

  Given a conceptual model (see Fig. 2) containing a page with a form, a set of pages reachable from such a form, and an arc connecting the form with each page, we can assign a number p_i to each arc representing the probability of the event E_i of having the crawler traversing that arc. Being p_i a discrete probability distribution, we have $0 \leq p_i \leq 1$ and $\sum_{i=1}^{N} p_i = 1$. The crawlability of a form is related to values assigned to probabilities p_i. Ideally a form is completely crawlable if all arcs have a non-negligible probability of being traversed ($p_i \gg 0$) and in general the crawlability of a form will depend on the number of arcs for which $p_i \gg 0$, i.e. the higher this number, the higher the crawlability.

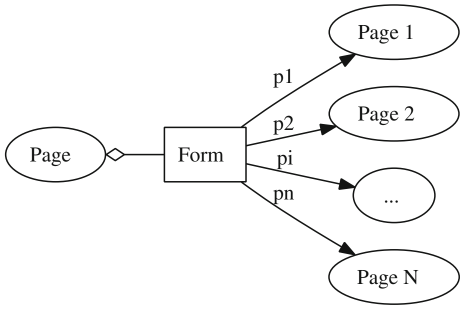


Fig. 2 Conceptual model augmented with arc traversing probabilities

3.2 Aspects affecting crawlability

Three main aspects of Web applications affect crawlability: *forms*, *client-side validation* and *server-side manipulation*.

- *Forms*. Form inputs affect the generation of dynamic content and sometimes the target pages that can be navigated, thus affecting page coverage. Although a form could have several types of input fields, we can divide those types into two categories: enumerable (often, those fields that in the form definition itself have a list of all possible input values from which the user can choose, e.g., drop down lists, radio buttons and check boxes) and unbounded (often, those fields that the user has to enter, e.g., text and password fields and search boxes). Given a form with different bounded/unbounded fields (e.g., Fig. 3 shows a simple form taken from Facebook) it is hard for a crawler to determine which field combinations and inputs lead to new dynamically generated pages that have to be covered during application testing. The crawler can randomly run the form with random combinations of fields selection and inputs, however, it is hard to detect/explore in such a way all the possible pages in reasonable time. In other terms, it could be hard for the crawler to generate meaningful values for the form fields.
- *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 typing a value into a text field or submitting the data in the form to a server component. These scripts could display text and/or automatically fill a dependent field when a

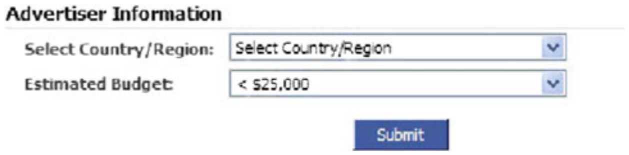


Fig. 3 Facebook Form: there are 1,004 possible input combinations for this simple form

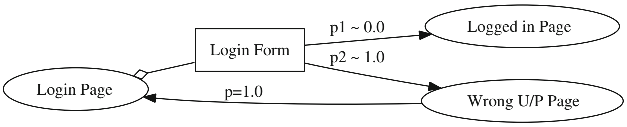


Fig. 4 Conceptual model for a login form

certain field is filled or highlight invalid inputs. This is done to help the user in filling out fields and validate the input value (in terms of format and content). Forms with complex scripts attached can be considered harder to explore for an automatic crawler since the crawler needs to generate meaningful values that overcome the validation check and proceed with the expected validation (avoiding error and redirection pages). Note that, in this work, we do not consider plug-ins and extensions such as applets, flash and ActiveX controls.

- *Server-side manipulation*. Several aspects regarding the server-side components can impact the application crawlability 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. In addition, inputs that are used to query the database also 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 can make an application hard to crawl for an automatic crawler. Furthermore, 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 application's crawlability. In this work, we mainly focus on the first aspect (multiple page generation paths). Further investigation will consider crawlability indicators for the other server-side aspects introduced here.

3.3 Capabilities of crawlers

If given infinite testing time, a random crawler is exhaustive, i.e., it fills unbounded textual input fields with random strings which eventually crawl every page that could be generated from a form, as a result of its ability to generate all possible inputs. Depending on the constraints imposed by the logic of the application, the probability p_i of visiting a page in response to a form submission could vary a lot. As an example, Fig. 4 shows the conceptual model of a typical Web application where a random crawler starting from a login page will have a probability p_1 (close to 1) of reaching the "Wrong user or password page", and a probability $p_2 = 1 - p_1$ (close to 0) of visiting the "Logged in page".

To increase the number of reached pages in a fixed number of attempts, a random crawler could be extended with capabilities of generating field values from some smaller "specialized" domains, e.g. numbers, emails, dates, and times. Figure 5 shows the set of capabilities we have considered in this work organized into a hierarchy rooted at the least specialized domain:

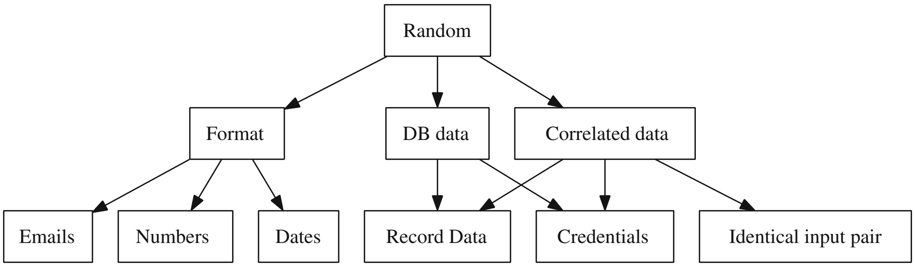
- *Random* random generation of character sequences, including the empty string;
- *Format* generation of properly formatted character sequences, which comprises: Emails, Numbers and Dates;
- *DB data* random values which are extracted from DB;
- *Correlated data* multiple inputs which are constrained to satisfy some relation;
- *Identical input pair*;
- *Record data* values from the same record;
- *Credentials* username and password are for a registered user.

A crawler enhanced with a given set of capabilities could implement some strategies attempting to improve its efficiency, e.g. by randomly selecting a domain for each field in a form and then generating random values from the selected domain. Continuing with the login example introduced above, if the crawler is provided with the "Credentials" capability which randomly (and uniformly) selects records from a username/password table and its strategy selects this capability with a probability of 0.5 whenever the crawler is facing a form with two fields, then $p_2 \sim 0.5$ (so $p_2 \gg 0$).

4 Metrics

We can distinguish between two cases: (1) the conceptual model of the Web application is known; (2) the conceptual model of the Web application is unknown. In case (1), crawlability can be measured directly in terms of model coverage:

Fig. 5 Crawling capabilities



- *PCOV* [*Conceptual client page coverage*]: proportion of conceptual client pages that are explored by a crawler with given capabilities.

Assuming the crawler's implementation is available, PCOV is determined by just running the crawler. For crawlers with long or unbounded execution times, this metrics may be parametrized with a time limit (i.e., pages covered within a given time bound). For crawlers with non-deterministic behaviors, this metrics may be interpreted statistically (considering average and standard deviation, or the probability of covering a page).

In case (2), crawlability cannot be measured directly. Internal metrics computed for the Web application are used to characterize its crawlability. These metrics are actually *indicators* (or indirect measures) of crawlability, the validity of which must be assessed empirically (e.g., by correlating them with PCOV, when this is available). It should be noticed that case 2 is the typical, normal case, when applying the IN-TESTING framework in practice, since the conceptual model of the Web application will be in general unknown. Case 1 is limited to controlled experiments conducted in a research setting (as the one we describe in the next section).

Crawlability depends on how input values are processed by the server-side and client-side code. On the server-side, input values might be validated before triggering the generation of new pages. Moreover, under different conditions, different (conceptual) pages may be produced. On the client-side, the presence of input validation is also an indicator of some filtering applied before allowing the user to navigate to the next page.

Some internal metrics may depend on the crawler's capabilities we are assuming. In such cases, we list the assumed capabilities explicitly.

We introduce two main categories of internal indicators of crawlability:

Server-side metrics: The server-side code generates the next client page depending on the submitted input. The existence of different page generation statements that are executed under different conditions is an indicator of a server-side component which produces different (conceptual) pages depending on the input values it receives upon form submission. To quantify such a variety of behaviors of the server code, a first rough indicator that we can use is McCabe's cyclomatic complexity, which measures the number of independent paths in a program [28]. Specifically, we need an interprocedural variant of the cyclomatic complexity, which includes also the contribution from transitively called functions. In fact, page generation statements can be inside functions transitively called by the main server component activated by form submission. The related metrics is

- *ICC* [*Interprocedural cyclomatic complexity*]: number of independent paths in the interprocedural control flow graph.

In practice, we can obtain the number of independent paths of each procedure as the number of conditional and loop statements (NCS), incremented by 1. However, for called functions, the increment should not be applied, since the base execution path is already accounted for in the main component. It should be noticed that ICC is indeed an indicator of *lack* of crawlability, in that low values of ICC indicate high crawlability, while high values of ICC indicate low crawlability. Hence, we use this metrics as an inverse indicator of crawlability, by interpreting low values as high crawlability and vice versa. The same holds for the following internal metrics.

A variant of ICC is *ICC-L0* (loops zero), which differs from ICC in that loops do not contribute to the measured cyclomatic complexity. The reason for ignoring loops is that the number of times they are traversed is usually not related to the number of distinct client pages that the server component can generate.

Consider the PHP program shown in Fig. 6 (reduced for readability). The program, invoked by pressing the submit button in a login form, is in charge of verifying user credentials, namely userid and password, and possibly to allow the user to login. Possible outcomes of such an invocation are (1) the home page is shown with the user logged in, (2) the login form is redisplayed to the user with an error message or (3) an error page is shown if an internal error occurs, e.g. a database connection fails. Upon being activated, the program verifies that expected input parameters are provided and match the

```
 1  <?php
 2  function checkMinAndMaxLength(...) { /* ICC = 2 */ }
 3  function emailCheck(...) { /* ICC = 4 */ }
 4  function registerLogin(...) { /* ICC = 1 */ }
 5  function checkMandatory (...) {
 6      if (! isset($_SESSION["{$formVars}"]["{$field}"]) ||
 7                  empty($_SESSION["{$formVars}"]["{$field}"])) {
 8          ...
 9          return false;
10      }
11      return true;
12  }
13  function authenticateUser(...) {
14      $password_digest = md5(trim($loginPassword));
15      $query = "SELECT password FROM users
16               WHERE user_name = '$loginUsername'
17               AND password = '$password_digest'";
18      $result = $connection -> query($query);
19      if(DB :: isError($result)) {
20          // return internal error page
21      }
22      if($result -> numRows() != 1)
23          return false;
24      else
25          return true;
26  }
27  function checkLogin(...) {
28      if (authenticateUser(...)) {
29          registerLogin(...);
30          // forward to home page
31      } else {
32          // back to login page
33      }
34  }
35  ...
36  if(checkMandatory(...))
37      checkMinAndMaxLength(...);
38  if(checkMandatory(...))
39      emailCheck(...);
40  checkLogin(...);
41  ?>
```

Fig. 6 Example of PHP program (reduced for readability)

proper syntax (lines 36–39), then the function *checkLogin* is called which in turn invokes the function *authenticateUser* to look for user credentials in the *users* database table. Functions whose body is not shown in the example have ICC reported in comments (e.g. line 2). ICC computation for this example develops as follows:

1. function *authenticateUser* has ICC of $1 + 2 = 3$ (two conditional statement, lines 19 and 22)
2. function *checkLogin* contains a conditional statement (line 28) and invokes *authenticateUser* and *registerLogin*, its ICC is $1 + 1 + (3 - 1) + (1 - 1) = 4$
3. function *checkManadatory* has ICC of 2 (one conditional statement at line 7)
4. the main program contains two conditional statements in its body and it calls *checkMandatory* (two times), *checkMinAndMaxLength*, *emailCheck*, and *checkLogin* its ICC is then $1 + 2 + 2*(2 - 1) + (2 - 1) + (4 - 1) + (4 - 1) = 12$.

Depending on designing and programming styles, server-side logic could account for various aspects of a Web application which dictates the complexity of the control flow:

- configuration/installation state: often Web applications are deployed using some kind of file transfer service (e.g. ftp) and then installed invoking some specific installation URL. User pages often contain code to check that such an installation has taken place;
- session state: applications often check whether a specific form action is invoked in the proper state (e.g. after the user has logged in);
- role management: applications could check whether an action is invoked by a user with some specific role, e.g. administrator, guest, etc.;
- action selection: a single server page may be devoted to generate content for different forms; in these cases, a mechanism is implemented to select which action to execute, e.g., via case statement on the value of a hidden input;
- internal errors management: pages access databases, send mails, or use other external services; all these activities may rise errors that do not depend on user inputs.

All the above aspects do not contribute to the number of pages generated in handling form submission and the complexity due to the code needed to implement them has to be ignored. One way to achieve this result is to measure the source code complexity on a slice instead of the entire code. The slice can be restricted to the code between input processing and page generation (technically, this is a variant of a slice called a *chop*).

As a further example, let us consider the code listed in Fig. 7. The code is in charge to handle all the administrative tasks for an web-based guestbook. Among the set of available features, the site administrator can create a list of (bad) words that are banned from users' messages. The administrator is provided with a form to add a new bad word which is handled by the shown code. Lines 15–22 check if the request is coming from an authenticated user with "admin" role, while lines 26–44 perform the selection of the requested action. Lastly, the code from line 34 to line 38 manages user inputs (checking, as an example, input word's length as shown at line 7), and generates the proper response.

The example above shows that to measure as exactly as possible the portion of the code which is responsible for mapping input under the control of the user to alternative resulting pages, we can take advantage of program slicing [38,41]. Specifically, we take a forward slice from all input receiving statements associated with input data which are submitted

```
 1  <?php
 2  function bad_word_add ($word) {
 3    ...
 4    if ($word === NULL || strlen ($word) === 0)
        {
 5      return FALSE;
 6    }
 7    if (strlen ($word) > $MAX_BAD_WORD_LENGTH) {

 8      die ("That word is too long.");
 9    }
10    ...
11  }
12
13  ...
14  session_start ();
15  if (! isset ($_SESSION ["username"]) || !
        isset ($_SESSION ["admin"])) {
16    relative_location ("../login.php");
17    exit ();
18  }
19  if ($_SESSION ["admin"] !== "TRUE") {
20    relative_location ("..");
21    exit ();
22  }
23  if (isset ($_REQUEST ["action"])) {
24    ...
25    $action = $_REQUEST ["action"];
26    switch ($action) {
27      case "delete":
28         ...
29
30      case "showbans":
31         ...
32
33      case "addbadword":
34        if (isset ($_POST ["word"])) {
35          bad_word_add ($_POST ["word"]);
36        }
37        ...
38        break;
39
40      ...
41
42      default:
43          die ("Invalid action.");
44      }
45  } else {
46      ...
47  }
48  ...
49  ?>
```

Fig. 7 Example of PHP page which needs slicing

by the form of interest, and we intersect it with the backward slice obtained from all output (client page generation) statements. Applying such a method to the above example, we have only to measure code defined at lines 2–11 and 34–38.

- *SL-ICC* [*Sliced ICC*]: interprocedural cyclomatic complexity computed for the intersection of the forward slice from input statements and backward slice from output statements.

The variant of SL-ICC with loops not contributing to the complexity can be considered for the sliced server code as well, giving the *SL-ICC-L0* metrics.

Client side metrics: We consider two metrics related to crawlability from the client-side point of view: (1) the number of fields of a form (NOF), and (2) the Javascript Validation Interprocedural Cyclomatic Complexity (JVICC).

- *NOF* [*Number of fields of a form*]: It indicates the number of enumerable and unbounded fields of a form. The composition of a form is expected to be relevant to its crawlability since often the data in input and output to/from the application is business data [25].

For efficiency reasons, input data validation is often moved to the client-side. In such cases, the form is not submitted to the server until it passes the client-side validation routines. The pattern to recognize client-side validation of form inputs is the following: an `onsubmit` event that calls a Javascript function is associated with the form tag or with one of the form fields by means of the `onclick` event. When the function, which performs the actual input data validation, returns false, form submission is blocked (and typically some error message is displayed to the user on the Web page or on a separate dialog window).

We identify the Javascript validation functions based on the form attribute `onsubmit` and we measure the interprocedural cyclomatic complexity [28] for them. In fact, more complex functions will require careful selection of the input values.

- *JVICC* [*Javascript validation interprocedural cyclomatic complexity*]: ICC computed for the Javascript input validation routines.

A variant of this metrics is *JVICC-B*, where the metrics is replaced by a boolean value (1 or 0), which just indicates whether any validation is performed or not. The reason for collapsing the cyclomatic complexity to a single boolean value is that the actual complexity of the input validation performed may not be reflected well in the number of independent paths that are counted in the validation routines.

For instance, let us consider the fragment of code shown in Fig. 8. In this code, two Javascript functions (*validate_email* and *validate_form*) are used in an HTML page to check the conformity between the content written by the user into the *email* input field of the page form *f* and the syntax of a typical email address (e.g., the presence of the character "@"). This check indicates that a Javascript-based validation of the form inputs is performed, i.e., *JVICC-B(f)=1*. Moreover, the two Javascript functions, *validate_email* and *validate_form*, have interprocedural cyclomatic complexity equal to 3 and 5, respectively. Hence, the overall Javascript validation interprocedural cyclomatic complexity of the form *f* is 5, i.e., *JVICC(f)=5*. We expect the JVICC metric to be related to the number of dynamic pages generated by the form when

```
....
<SCRIPT TYPE="text/javascript"><!--
function validate_email(field,alerttxt){
var valid=false;
if (field){
 apos=value.indexOf("@");
 dotpos=value.lastIndexOf(".");
 if (apos<1||dotpos-apos<2){
   alert(alerttxt);
   valid=true;}
 }
 return valid;
}
function validate_form(thisform){
if (thisform)  {
 if (validate_email(email,"Not a valid e-mail address!")==false){
   email.focus();
   return false;}
 }
}
//--></SCRIPT>
....
<form name="f" action="submit.htm" onsubmit="return validate_form(this);">
Email <input type="text" name="email" size="30"/>
<input type="submit" value="Submit">
</form>
....
```

Fig. 8 Simple example of Javascript-based validation

a validation is performed. For instance, in the example above, the form generates at least two conceptual pages according to the pass/fail of the *email* field validation.

5 Evaluation

An experiment has been conducted with the aim of investigating crawlability and, in particular: (1) the relationship that exists between our metrics and form crawlability, in other terms, we investigated whether our server and client side metrics are good surrogates for the quantification of crawlability; and (2) the impact of the crawler's capabilities on crawlability. To these aims, we formulate the following research questions.

RQ1 Do internal metrics correlate with a direct measure of crawlability?

RQ1 deals with the relationship that exists between our metrics (internal measures of a Web application) and form crawlability (an external property of a Web application). To evaluate this relationship, we manually build the conceptual model of the given Web application and we determine precisely which conceptual pages are explored by a crawler with given capabilities and which are not. Then, we run our tool to compute the internal metrics (e.g., *ICC*, *NOF* and *JVICC*) on the application code and correlate them with the fraction of unexplored conceptual pages reachable from each form (i.e., *1-PCOV*). We used the *Spearman's* correlation coefficients [24] for evaluating the relationship between metrics and crawlability. This coefficient reflects both the degree of linear and curvilinear relationship between two variables and range from $+1$ to -1. A correlation of $+1$ means that there is a perfect positive correlation between variables; -1 means that there is a perfect negative correlation. A coefficient of 0 indicates the absence of correlation. The p value associated with the coefficient gives the statistical significance of the evaluation, in other terms, it estimates how much the correlation is due to random reasons.

Finally, we performed a regression analysis to build prediction models for crawlability of forms from the collected measures and such a model could be used by the crawler to estimate the number of pages that are reachable from a particular form given the values of the metrics. Comparing the estimate value with the number of pages actually discovered behind a form and taking into consideration the number of attempts already made on the form, the crawler's strategy may decide which action to perform, e.g. generating a new input or asking the tester for help.

RQ2 Which types (or set) of crawler capabilities have major impact on crawlability?

RQ2 deals with the relationship that exists between the ability of the crawler of crawling Web pages with respect to its capabilities. To evaluate this relationship, for each page contained in the conceptual model of the Web application, we identified the set of crawler capabilities (among those previously introduced in the taxonomy, see Sect. 3) required to crawl the page. In other terms, given a Web page, we identified the capabilities (e.g., random string, dates, valid emails) that can significantly increment the chance of a random crawler to crawl (i.e., reach) some uncovered page in reasonable time. Finally, we evaluated the impact of the (set of) crawler capabilities by plotting them with respect to the number of pages they affect.

5.1 Objects

Table 1 summarizes some data about the objects taken into consideration in the experiment. To answer *RQ1* we evaluate the correlation between metrics and crawlability by analyzing 51 Web pages related to 19 Web applications (e.g., OScommerce2, Zencart, phpMyAdmin), which contain a set of 58 HTML forms generating HTML pages in response to the data submitted with the forms. With respect to *RQ2*, we analyzed all the 189 Web pages dynamically generated by the same set of 58 HTML to understand the capabilities required to reach each page.

Table 1 Analyzed applications

Application	URL	Number of considered pages	Number of forms
oscommerce2	http://www.oscommerce.com	2	2
campcaster	http://www.campware.org	8	8
ajchat	http://ajchat.sourceforge.net	1	1
shop	http://demo.webasyst.net/shop	1	1
hotelBooking	http://demo.onlinebookingmanager.com	3	3
zencart	http://www.zencart-italia.it	4	4
orangeHRM	http://www.orangehrm.com	3	4
spellCheck	http://www.netjs.com/speller	2	2
addressBook	http://www.corvalis.net/address	2	3
webChess	http://webchess.sourceforge.net	2	2
schoolMate	http://www.primateapplications.com	2	2
phpmyadmin	http://www.phpmyadmin.net	5	8
addressook	http://www.corvalis.net/address/	1	1
php-agenda	http://www.abeel.be/php-agenda	3	3
airalliance	http://blogs.sun.com/phantom/resource/php/AirAlliance.tar.gz	1	1
ecombill	http://sourceforge.net/projects/ecombill	1	1
dbguestbook	http://freshmeat.net/projects/dbguestbook	2	3
lesson8	http://netbeans.org/kb/61/php/wish-list-lesson8.html	3	4
winestore	http://www.webdatabasebook.com	5	5
Total			
19	–	51	58

5.2 Procedure

The experiment has been performed by iterating the following steps for each considered application:

1. We ran the application. When possible we locally installed the application (e.g., OScommerce, Zencart), alternatively, we ran the online demo (e.g., OrangeHRM).
2. We manually built the conceptual model of the Web application, in which we identified the set of conceptual pages generated by starting from each form.
3. We measured our client/server-side metrics by applying our tool to the application code exercised from the considered forms. Note that, depending on the availability of the application source code and the capability of our prototype tool to measure the metrics on the given code, we collected different sets of metrics for different applications. For instance, for the applications for which we analyzed the online demo (e.g., OrangeHRM), we accessed only the client side code (HTML enriched by Javascript) thus we measured only the client side metrics. Overall, we evaluated 39 forms (out of 58) for the client side metrics and 18 (out of 58) for the server-side ones. There is no overlap between these two sets of forms.
4. We plot in a graph the values of our metrics with respect to the crawlability and we evaluated the correlation by computing the Spearman's coefficient.
5. For each Web page dynamically generated from the considered forms, we manually identified the set of capabilities (starting from those described in the taxonomy) required to reach/generate such a page.
6. We evaluated the relevance of the set of crawler capabilities by plotting them with respect to the number of pages they contribute to crawling.

5.3 Results

In this section, we summarize and discuss the results of the experiment and the most relevant threats to validity and we indicate some interesting outcome of the experiment that will be further investigated.

5.3.1 RQ1: Metrics as crawlability indicators

Metric measures obtained in the experiment are fully listed in Table 5 in Appendix A, while Table 2 reports only some

Table 2 Examples of the obtained results: metrics and crawlability

App	Page	Form	Client side metrics		Number of generated pages	1-PCOV
			NOF	JVICC		
oscommerce2	create_account	createAccount	20	18	4	0.75
oscommerce2	advanced_search	advancedSearch	11	46	3	0.67
campcaster	StationSettings	changeStationPrefs	11	8	2	0.50
…						
App	Page	Form	Server side metrics		Number of generated pages	1-PCOV
			ICC-L0	SL-ICC-L0		
addressbook	add_entry	theform	262	1	1	0.00
airalliance	process_itinerary	provide_itinerary_details	16	3	2	0.50
dbguestbook	admin	add_bad_word	151	6	4	0.75
…						

examples. The table lists: (1) the considered forms; (2) the computed metrics for each form; (3) the number of conceptual pages generated by each form, when clicking the submit button; and (4) the fraction of unexplored conceptual pages reachable from a form (i.e. 1-PCOV). PCOV has been computed taking into consideration, random crawler, that exercises HTML forms by submitting random sequences of characters and such a crawler can be easily implemented by tools such as *wget*[1] and *HTTPUnit*[2] that are able to submit HTTP requests to a Web server without any ability of managing client-side scripting languages, such as Javascript.

Figure 9a and b plots the collected measures of NOF, JVICC, ICC-L0 and SL-ICCL0 (y axes) for each analyzed form (x axes). In both figures, the solid-dot-based line (i.e., GeneratedPages) represents the number of pages generated by each form while the other lines represent the metric values measured for the corresponding form. Since the points have been ordered along the x axes by increasing values of 1-PCOV, we can visually inspect the trend of the metrics with respect to the number of pages generated by the forms and to the form crawlability.

Figure 9a shows that the values of the client-side metrics have an overall trend slightly comparable with the trend of the number of generated pages and thus potentially with the form crawlability. However, several values of NOF and JVICC seem to be far from their corresponding points on the GeneratedPage line and, moreover, several peaks exist in which NOF and JVICC really diverge from the number of generated pages.

Conversely, Fig. 9b shows that the values for the server-side metrics (ICC-L0 and SL-ICC-L0) have a trend strongly comparable with the number of generated pages, and several points overlap. In particular, the points of the SL-ICC-L0 line result really close to the points of the Generated-Pages line, thus indicating that a good correlation can be expected to exist between the metric and the form crawlability. A quite similar trend can be also observed for ICC-L0. However, a limited number of peaks, in which ICC-L0 diverges from the GeneratedPages line, negatively impacts on the overall relationship between ICC-L0 and the form crawlability.

Let us consider now two representative examples out of the set of outliers in our results: the *advancedSearch* form of *oscommerce2* and the *add_bad_word* form of *dbguestbook* (their metric values have been highlighted in Fig. 9a, b).

The *advancedSearch* form allows the user to search products in the online store. The form contains a not small number of fields (NOF=11) and it uses a not trivial Javascript code ($\text{JVICC} = 46$). Both the form fields and the Javascript code have no impact on the number of pages generated by the form as they are used to define options enriching the search, thus making the search functionality attractive for the user.

In the second example, the *add_bad_word* has a quite complex PHP source code with a not trivial control-flow ($ICC - L0 = 151$) that generates just four pages. However, a large part of the code is devoted to manage user session, user role, internal errors and activity selection. As noted before, such part of the code does not actually contribute to pages generation, it increases the value of ICC-L0, thus making this metric a not so precise indicator of the form crawlability (this is evident in Fig. 9b, where the points related to ICC-L0 and the number of generated pages for *add_bad_word* diverge).

[1] http://www.gnu.org/software/wget.

[2] http://httpunit.sourceforge.net.

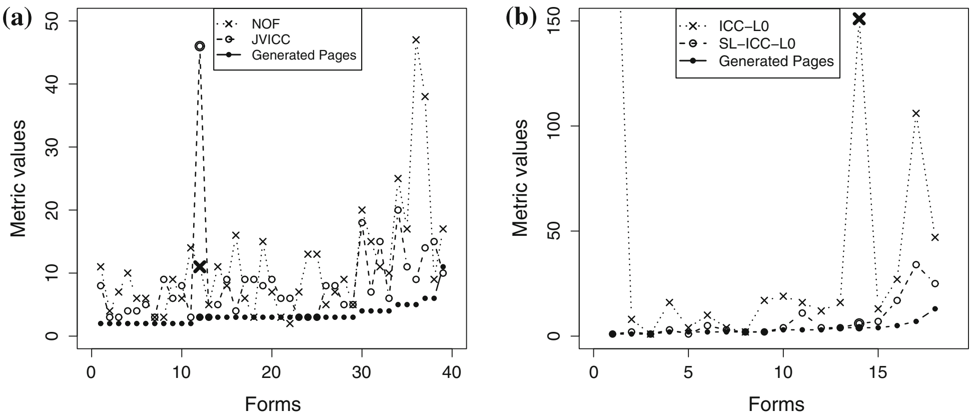


Fig. 9 Client and server metric values per form. **a** Client metric values per form. Notice that forms are plotted according to their increasing 1-PCOV measure, **b** Server metric values per form. Notice that forms are plotted according to their increasing 1-PCOV measure. *Y* axis range is limited to about 150 for readability and an outlier value of 262 for ICC-L0 at generated pages = 0 is not shown in the plot

Table 3 Metrics and crawlability: Spearman's ρ and p values

	ρ	p value
NOF versus JVICC	0.32	0.041
NOF versus 1-PCOV	0.55	0.00026
JVICC versus 1-PCOV	0.59	6.27E−05
ICC-L0 versus 1-PCOV	0.53	0.024
SL-ICC-L0 versus 1-PCOV	0.89	7.35E−07

Only avoiding such an amount of code, e.g., by means of code slicing, we can identify the code that actually contributes to the pages construction for the form $(\mathrm{SL} - \mathrm{ICC} - \mathrm{L0} = 6)$. In this case, SL-ICC-L0 represents a good indicator of the form crawlability (the points related to SL-ICC-L0 and the number of generated pages for *add_bad_word* overlap in Fig. 9b).

A numerical evaluation of the qualitative observations made above can be given by means of the Spearman's correlation coefficient. Table 3 shows the obtained correlation measures. We see that a moderate correlation (59%) exists between JVICC and crawlability and between NOF and crawlability (55%). Moreover, a lower correlation (32%) exists between NOF and JVICC, thus indicating that they (potentially) measure different information. The moderate correlation existing between NOF, JVICC versus crawlability suggests that both these metrics are not a so good indicator (indirect measure) of the number of pages dynamically generated by the forms. Furthermore, the existence of a correlation (even moderate) between JVICC and crawlability seems support our hypothesis for which if a client-side validation check is attached to a form by means of Javascript the complexity of such a code is only partially correlated with the number of pages generated by the form itself. Hence, this type of code has a limited impact on the ability of a crawler of crawling (e.g., reaching), the pages dynamically generated from the application by starting from the data of a form.

On the server side, the table shows that the set of collected data provides evidence of a moderate correlation (53%) between the cyclomatic complexity ICC-L0 and 1-PCOV (p value < 0.025) and a very strong correlation (89%) between the sliced version SL-ICC-L0 of interprocedural cyclomatic complexity and 1-PCOV (p value $< 10^{-6}$).

Considering the strong correlation obtained for the server-side SL-ICC-L0 metric, we additionally perform a regression analysis of PCOV versus SL-ICC-L0 assessing the goodness of fit for a linear model and two non-linear models (exponential and hyperbolic). In defining such models, we assume PCOV equal to 1 if SL-ICC-L0 is 1, as a (possibly sliced) server-side page having a cyclomatic complexity of 1 generates necessarily just one page, and thus this single unique page is covered by the crawler.

If we indicate with y the dependent variable PCOV, with x the independent variable SL-ICC-L0, and with β the model parameter, the three considered models have the following forms:

1. L1 (linear): $y = 1 + \beta(x - 1)$
2. NL1 (exponential): $y = e^{\beta(x-1)}$
3. NL2 (hyperbolic): $y = 1/(1 + \beta(x - 1))$

We re-scale x by subtracting 1 to map it into the $[0, +\infty)$ interval.

We use the "leave-one-out" cross-validation method to rank the models based on accuracy: taking the ith sample as the test sample, we train each model on the remaining samples and then we compare the predicted value with the test sample. We rank the models from measures based on the magnitude of the absolute error (MAE):

$$\mathrm{MAE} = |\text{actual PCOV} - \text{predicted PCOV}|$$

Table 4 summarizes the results: MMAE is the mean magnitude of absolute error, MdMAE is the median of MAE, PRED(0.25) and PRED(0.50) are the percentages of cases for which MAE ≤ 0.25 and MAE ≤ 0.50, respectively, SDMAE is the standard deviation of MAE. The table shows that the hyperbolic model provides the best predictor with a MMAE of 0.13 which means that on average, the predicted value differs from the sample about 13% of the value. The curve

Table 4 Prediction accuracy of linear, exponential and hyperbolic models

Model	MMAE	MdMAE	PRED(0.25)	PRED(0.50)	SDMAE
L1	0.33	0.42	33	72	0.21
NL1	0.25	0.33	44	94	0.17
NL2	0.13	0.13	83	100	0.09

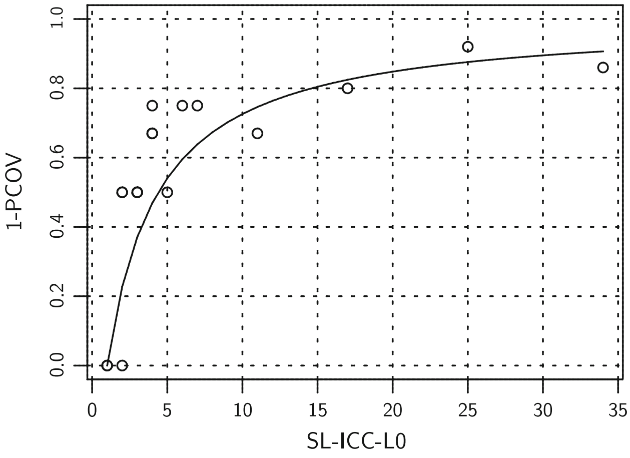


Fig. 10 Plot of 1-PCOV versus SL-ICC-L0. The *solid line* represents the regression curve of the hyperbolic model

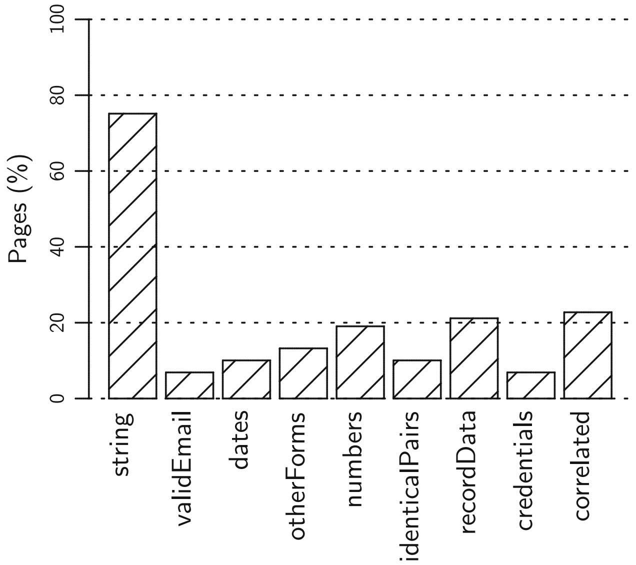


Fig. 11 Crawler's capability required to cover the pages under analysis (one page may require multiple capabilities)

of the hyperbolic prediction model is shown in Fig. 10. It is interesting to note that the form of the hyperbolic model has a natural interpretation. In fact, it can be read as the ratio of the number of pages a random crawler can easily traverse (1 page), to the typical fan-out of a server-side page, which in turn is $1 + \beta(x - 1)$, with $\beta = 0.294$ and $x - 1$ the number of (possibly sliced) conditional statements in the code (i.e., SL-ICC-L0-1).

These results highlight the trade-off between source code analysis effort and precision of ranking/prediction. It is relatively easy to implement a tool that computes ICC-L0 while it would be a difficult work to build a tool that performs the same slicing techniques we applied manually for this study. On the other hand, ICC-L0 is sufficient to build a ranking of forms by crawlability, while SL-ICC-L0 gives us a quite accurate predictor.

5.3.2 RQ2: Crawler capabilities

Figure 11 shows the percentage of dynamically generated pages with respect to each crawler's capability required to generate such pages (a page may require multiple capabilities). For instance, we see that the capability of generating strings (empty and not) is relevant since it is required in 75% of the analyzed pages. Furthermore, 19% of such pages requires the capability of generating numbers. We can notice that the most relevant capability is string generation, while the least relevant ones are both the ability of generating valid emails and credentials (e.g., login and password). The remaining capabilities are slightly above these two.

Figure 12 shows the percentage of pages that are covered by means of the number of capabilities in the horizontal axis. For instance, one capability is sufficient for covering 52% of the considered pages. While sets of two capabilities (e.g., the ability of generating both strings and numbers, or alternatively the ability of generating both valid emails and dates) are required to cover 24% of the pages. On average, three capabilities are sufficient to cover about the 80% of the dynamically generated pages.

Furthermore, Fig. 13 shows how the generated pages are covered by considering incrementally composed sets of capabilities, generated by taking into account the outcome shown in Fig. 11, and adding more capabilities in a greedy way. In the figure, we can observe that the capabilities of generating strings and correlated fields can cover more than 52% of the pages while considering also record data and numbers we can cover about 80% of the generated pages.

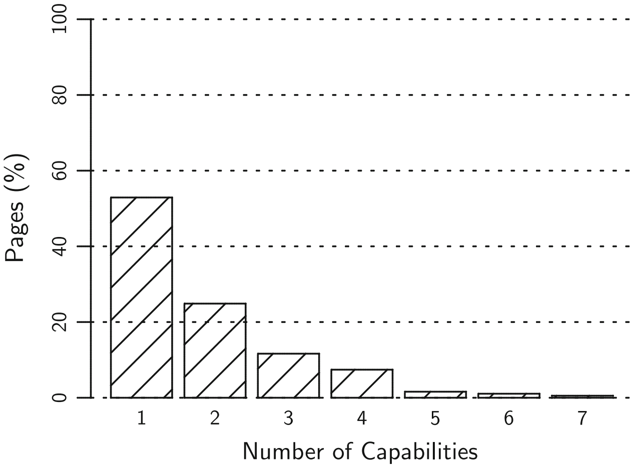


Fig. 12 Sets of crawler's capabilities with respect to the percentage of generated pages that require such sets of capabilities

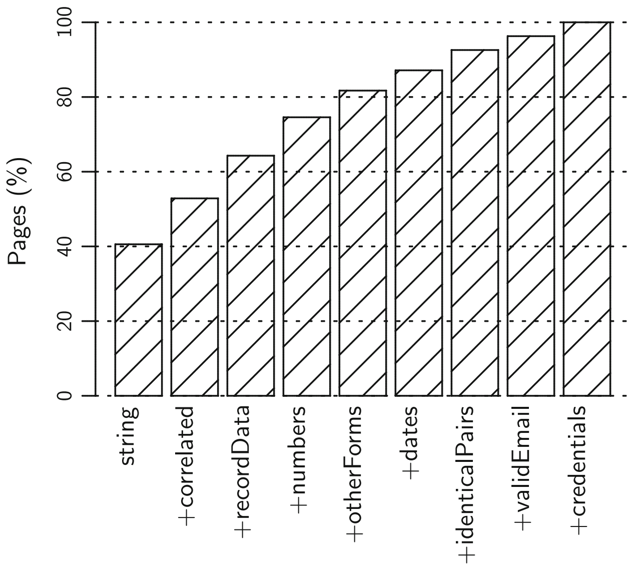


Fig. 13 Page coverage by incremental sets of crawler's capabilities

5.3.3 Threats to validity

Several threats to validity affect the overall results of the experiment. In this section, we discuss and analyze the most relevant ones concerning *External validity*, *Internal validity*, *Construct validity* and *Conclusion validity*.

External validity threats relate to the generalization of results. Two important threats that limit the generalization of the obtained results are related to: the limited number of applications, forms and pages considered to answer respectively *RQ1* and *RQ2*, and the representativeness of the used Web applications with respect to the application domain and characteristics (e.g., we mainly considered PHP applications). Further iterations of the experiment can better support the obtained results. However, we believe that 19 real applications belonging to different domains and 58 forms related to 51 pages make the context of our experiment realistic and effective.

Internal validity threats concern external factors that may affect a dependent variable. The most relevant threat concerning the internal validity is related to the subjectivity of some tasks performed in the experiment. In particular, in the subjective task involved in the construction of the application conceptual models (i.e., required for answering *RQ1*) knowledge, skills and human errors done by testers that performed such a task could influence the obtained results and different testers may lead to obtain different results. To limit this threat, we tried to formally define what is a conceptual model, its relevant characteristics and we asked the testers to apply the Conallen's design methodology [7].

Construct validity threats concern the relationship between theory and observation. A possible threat comes from the choice of considering only a random crawler for answering *RQ1* approximating the PCOV as $PCOV = 1 - \frac{Number of covered pages}{Number of generated pages}$. Another threat to the validity comes from the simple taxonomy of crawler capabilities considered for answering *RQ2*. For instance, in our taxonomy we do not distinguished among, e.g., positive or negative numbers, integer or real numbers. Further iterations of the experiment will consider these aspects as well as additional capabilities to obtain improved results.

Conclusion validity threats concern the relationship between the treatment and the outcome. One of such threats concerns the use of few points during the statistical analysis (i.e., limited number of applications, pages and forms considered in the experiment). Further investigation and experiment repetitions will lead to a larger set of data in which more sophisticated statistical analysis can be performed.

6 Related work

Several Web testing techniques and tools have been proposed as a result of the increased pervasiveness of Web applications. This increased pervasiveness demands for 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: the tester navigates through the Web application recording various testing scenarios which are then repeated during regression testing. However, the quality of the produced test suite relays on the tester's thoroughness and skill. Also, changes in the structure of the Web application could make the recorded test suite fail to run. This makes it necessary to re-record all or part of the tests. An alternative is based on the tools, such as HttpUnit. HttpUnit is a Java

API that provides the building blocks required to emulate the browser's behavior. When combined with a framework, such as JUnit, HttpUnit allows testers to create test cases to verify Web Application behavior [22]. Both approaches require significant manual effort from the tester and require time and resources that might not be available or cost-effective.

Web crawlers can be viewed as testing tools as they traverse a Web application reporting failures and broken links. Many open source and commercial crawlers exist with varying degrees of functionalities. WebSPHINX [30] is a customizable site-specific spider with a GUI interface. It also provides a class library that can be used to implement spiders in Java. Mercator [21] has similar features to WebSPHINX, but is designed to deal with scalability issues. JSpider[3] is another open source crawler that has the advantage of recording a Web application's structure to a database. However, all three crawlers do not provide support for automated form filling and submission. Teleport Pro[4] is a commercial crawler that provides a few additional features. The crawler gives the user the ability to provide authentication information to access password protected parts of the application. It also parses JavaScript to extract links. Girardi et al. [15] conducted a comparison of these and other crawlers based on completeness, robustness, features offered and download limiting options. The study concluded that the considered crawlers have different strengths and weaknesses but indicated that some commercial crawlers offer more completeness. Although we have the advantage of being automated, their ability to cover an application is closely related to their capabilities.

Model-based testing of Web applications was initially proposed by Ricca and Tonella [32] 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 [27,29]. These approaches extract test cases by studying the client-side behavior of the application.

Elbaum et al. [12] were the first to propose 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. Sampath et al. [33–35,37] focused on techniques for prioritizing session based test cases and reducing the test suite. Using session data to create test suites proved to be effective and reveals faults that were not discovered by white box techniques. However, if the application is new or was not configured to record session data this approach cannot be used.

Existing industrial practice relies almost exclusively on manual testing, augmented by the use of crawlers and technologies for simple testing, built on top of crawling technologies. Our metrics for the assessment of crawlability are directly related to this work, since they assess the degree to which the automated aspects of the testing process can be considered to be effective and the degree of effort that will be required in the manual phase of the process, not covered by crawling. However, in the literature, there are also recent techniques to improve automation beyond simple crawling. These approaches use more advanced algorithms to probe the branches not covered by a crawler. Our approach will also be useful for these more advanced techniques because it will help to assess the degree of improvement that can be achieved over-and-above crawling, using such techniques. This may help with decision maker in choosing when and where to deploy more advanced techniques.

Concolic testing of Web applications is used to cover server-side code or uncover vulnerabilities that could lead to security threats. Concolic testing is a combination of symbolic and concrete execution. The program is first run with no input values. Constraints are gathered through the application. Part of the constraint is negated and a constraint solver is used to produce inputs that satisfy the new constraint. The inputs are then used in the next run to execute a different path. Wassermann et al. [40] used this approach to dynamically generate test data for PHP Web applications. In their paper, only constraints related to the parts of the program that call database queries are examined. The technique was able to find vulnerabilities that could lead to SQL injections not found by other static techniques. Artzi et al. [4] used a similar approach to generate test data and find bugs in PHP applications. A tool called Apollo was developed to implement the approach and generate inputs. The tool also monitors the output for crashes and malformed HTML and minimizes the conditions on inputs that cause failures and outputs a bug report. To perform symbolic execution, the code needs to be transformed to simulate user actions such as button pressing. The transformation is done manually which limits the automation of the approach. The tool was evaluated on four PHP applications and recorded an average of over 50% line coverage. The results are promising, but the coverage achieved indicates that manual testing is still needed for testing the applications. Also, only server-side code is tested and client-side scripts, such as JavaScript still need to be considered.

Interface discovery can be useful in providing guidance for test data generation. Halfond and Orso [17,18] used static analysis to identify interfaces of Java Web applications. The discovered interfaces were then evaluated by measuring their effectiveness in generating test cases. The work was then

[3] http://j-spider.sourceforge.net/.

[4] http://www.tenmax.com/teleport/.

extended by Halfond et al. [16] to use symbolic execution to identify interfaces. Fisher et al. [13] also worked on interface discovery. Their approach was dynamic in that they submit all possible combinations and subsets of input parameters analyzing the output.

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 behavior 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 van Deursen [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 [23] suggests that testability metrics can be used to identify parts of the application causing a lower testability level by analysing test critical dependencies. Harman et al. [19,20] introduced testability transformation as a means of improving testability, by transforming systems from less testable to more testable versions with equivalent test adequate test suites. Our metrics, introduced in this paper, may be used to guide possible approaches to Web Application Testability Transformation, although this remains a topic for future work.

In Web applications, metrics have been used especially for usability, maintainability and evolution. Emad Ghosheh et al. [14] 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. [39] created a tool to collect a number of metrics to measure Web application evolution over an interval of time. Douglis et al. [10] conducted a study to evaluate caching methods in the World Wide Web. They defined a number of metrics and studied the relationship between a Web application’s characteristics and the effectiveness of caching methods. Palmer [31] defined and validated a number of usability metrics for Web applications. Navigability was among the metrics that were proved by the study to correlate to a Web application’s success. Although navigability and crawlability have similar context, navigability in Palmer’s paper is defined in terms of sequence and layout. Dhyani et al. [9] conducted a survey of Web application metrics that can be used in improving content. In the context of using metrics to aid testing, Bellettini et al. [5] 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 they were used for stopping the testing process based on user criteria.

The framework we propose combines the existing Web testing approaches with crawlability metrics. Crawlability could be considered closely related to traditional applications testability, but is focused on Web navigability. This can help in prioritizing parts of the application that are hard to crawl during the testing process. Combining testing with crawlability metrics supports the tester in identifying areas of the application that need more attention thereby decreasing the testing effort. The primary novelty of our IN-TESTING approach is that measures are used during the testing to guide the tester where her/his efforts could be most effective.

7 Conclusions

In this paper, we introduced the notion of crawlability and presented an experiment with 19 real applications analyzed to evaluate the correlation existing between the direct measure of the application’s crawlability and structural metrics. The obtained results show that our metrics can be reasonable indicators of crawlability. Although preliminary, our results highlight some interesting features of the systems under test and the way in which our metrics can be used to draw the tester’s attention and simplify the testing activity. Slice restricted quantification of the cyclomatic complexity is more effective than the same metrics computed on the original code. Around 80% of pages can be covered by crawlers which possess the four most effective crawling capabilities (i.e., string generation, correlated data generation, record reuse and number generation).

Future work will focus on the development of additional measurements at the server side to complement those presented in this paper. Moreover, we will study crawlability predictor metrics for crawlers with enhanced capabilities (i.e., those more advanced than the random crawler). The experimental results documented in this paper encourage us to proceed our investigation about the use of crawlability metrics for improving Web testing. In particular, we plan to further improve the implementation of the WATT tool [2] to provide the community with a tool that supports our IN-TESTING framework. We will add the investigated capabilities and the ability to select the most appropriate ones to WATT.

Appendix A

In this section, we provide additional data collected during the experiment about the measured metrics and crawlability but not included in the main sections of the paper for space reason.

Table 5 Metrics and crawlability

App	Page	Form	Client side metrics		Number of generated pages	1-PCOV
			NOF	JVICC		
oscommerce2	create_account	createAccount	20	18	4	0.75
oscommerce2	advanced_search	advancedSearch	11	46	3	0.67
campcaster	StationSettings	changeStationPrefs	11	8	2	0.50
campcaster	importPlayLists	PL_importForm	5	3	3	0.67
campcaster	AddGroup	addsubject	4	3	2	0.50
campcaster	AddAudio	uploadFile	7	3	2	0.50
campcaster	newPlayLists	editMetaData	10	4	2	0.50
campcaster	ChangePsw	chgPsw	6	4	2	0.50
campcaster	AddUser	addsubject	6	5	2	0.50
campcaster	newWebcam	addWebstream	11	5	3	0.67
ajchat	ajchat	f1	8	9	3	0.67
shop	indexphp	subscription_form	3	3	2	0.50
hotelBooking	avialability.php	checka	16	4	3	0.67
hotelBooking	indexphp	checka	15	7	4	0.75
hotelBooking	checkavail.php	checka	38	14	6	0.83
zencart	index	create_account	25	20	5	0.80
zencart	index_advancedSearch		11	15	4	0.75
zencart	product_reviewes	product_reviews_write	6	9	3	0.67
zencart	checkout	checkout_payment	3	9	3	0.67
orangeHRM	companyInfo_general	frmGenInfo	15	8	3	0.67
orangeHRM	companyInfo_structure	frmAddNode	7	9	3	0.67
orangeHRM	companyInfo_structure	frmDeleteNode	3	9	2	0.50
orangeHRM	jobSpecification	frmJobspec	3	6	3	0.67
spellCheck	speller	form1	2	6	3	0.67
spellCheck	controls	spellcheck	9	6	2	0.50
addressBook	AddToGroup	mainForm	6	8	2	0.50
addressBook	Edit	theForm	17	10	11	0.91
webChess	MainMenu	PersonalInfo	7	3	3	0.67
webChess	newUser	UserData	13	3	3	0.67
SchoolMate	index-addclasses	Classes	17	11	5	0.80
SchoolMate	index-reg	Registration	13	3	3	0.67
phpmyadmin	server_provilegies	userForm	47	9	5	0.80
phpmyadmin	db_operations	tbl_create	5	8	3	0.67
phpmyadmin	db_operations	dbop1	7	8	3	0.67
phpmyadmin	db_operations	dbop2	14	3	2	0.50
phpmyadmin	server_sql	sqlForm	9	15	6	0.83
phpmyadmin	user_password	chgPassword	10	6	4	0.75
phpmyadmin	structure	tbl_addField	9	5	3	0.67
phpmyadmin	structure	tbl_indexes	5	5	3	0.67

Table 5 continued

App	Page	Form	Server-side metrics		Number of generated pages	1-PCOV
			ICC-L0	SL-ICC-L0		
addressbook	add_entry	theform	262	1	1	0.00
airalliance	process_itinerary	provide_itinerary_details	16	3	2	0.50
dbguestbook	admin	add_bad_word	151	6	4	0.75
dbguestbook	guestbook	sign_guestbook	106	34	7	0.86
ecombill	enter_lab_tests	lab	1	1	1	0.00
lesson8	create_new_wisher	create_new_wisher	13	7	4	0.75
lesson8	edit_wish	edit_wish	10	5	2	0.50
lesson8	index	login	4	3	2	0.50
lesson8	index	wishlist	2	2	2	0.50
php-agenda	admin	deleteuser	19	4	3	0.67
php-agenda	register	register	16	4	4	0.75
php-agenda	main_page	add_todo	8	2	1	0.00
winestore	changepasswd	details	27	17	5	0.80
winestore	login	login	16	11	3	0.67
winestore	your_cart	update	12	4	3	0.67
winestore	details	details	47	25	13	0.92
winestore	search	seach	17	2	2	0.50

References

1. Alshahwan, N., Harman, M.: Automated session data repair for web application regression testing. In: ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, pp. 298–307. IEEE Computer Society, Washington (2008)
2. Alshahwan, N., Harman, M., Marchetto, A., Tonella, P.: Improving web application testing using testability measure. In: IEEE International Symposium on Web Systems Evolution (WSE), IEEE Computer Society, Edmonton (September 2009)
3. Anneliese, J.O., Andrews, A., Alexander, R.T.: Testing web applications by modeling with fsms. Softw. Syst. Model. **4**, 326–345 (2005)
4. Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., Ernst, M.D.: Finding bugs in dynamic web applications. In: ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis, pp. 261–272. ACM, New York (2008)
5. Bellettini, C., Marchetto, A., Trentini, A.: TestUml: user-metrics driven web applications testing. In: SAC '05: Proceedings of the 2005 ACM Symposium on Applied Computing, pp. 1694–1698. ACM, New York (2005)
6. Bruntink, M., van Deursen, A.: An empirical study into class testability. J. Syst. Softw. **79**(9), 1219–1232 (2006)
7. Conallen, J.: Modeling web applications with uml. In: White Paper (1999)
8. Conallen, J.: Building Web Applications with UML. Addison-Wesley, Reading (2000)
9. Dhyani, D., Ng, W.K., Bhowmick, S.S.: A survey of web metrics. ACM Comput. Surv. **34**(4), 469–503 (2002)
10. Douglis, F., Feldmann, A., Krishnamurthy, B., Mogul, J.: Rate of change and other metrics: a live study of the world wide web. In: USITS'97: Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems. USENIX Association, Berkeley (1997)
11. Elbaum, S., Karre, S., Rothermel, G.: Improving Web application testing with user session data. In: Proceedings of the 25th International Conference on Software Engineering (ICSE), pp. 49–59. IEEE Computer Society, Portland (May 2003)
12. Elbaum, S., Rothermel, G., Karre, S., Fisher, M. II.: Leveraging user-session data to support web application testing. IEEE Trans. Softw. Eng. **31**(3), 187–202 (2005)
13. Fisher, M. II, Elbaum, S., Rothermel, G.: Dynamic characterization of web application interfaces. In: FASE'07: Proceedings of the Fundamental Approaches to Software Engineering, pp. 260–275. Springer, Berlin (2007)
14. Ghosheh, E., Qaddour, J., Kuofie, M., Black, S.: A comparative analysis of maintainability approaches for web applications. In: AICCSA '06: Proceedings of the IEEE International Conference on Computer Systems and Applications, pp. 1155–1158, IEEE Computer Society, Washington (2006)
15. Girardi, C., Ricca, F., Tonella, P.: Web crawlers compared. Int. J. Web Inform. Syst. **2**, 85–94 (2006)
16. Halfond, W.G., Anand, S., Orso, A.: Precise interface identification to improve testing and analysis of web applications. In: ISSTA '09: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, pp. 285–296. ACM, New York (2009)
17. Halfond, W.G.J., Orso, A.: Improving test case generation for web applications using automated interface discovery. In: ESEC-FSE '07: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, pp. 145–154. ACM, New York (2007)
18. Halfond, W.G.J., Orso, A.: Automated identification of parameter mismatches in web applications. In: SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Sympo-

sium on Foundations of Software Engineering, pp. 181–191. ACM, New York (2008)

19. Harman, M.: Open problems in testability transformation (keynote). In: 1st International Workshop on Search Based Testing (SBT 2008), Lillehammer (2008)
20. Harman, M., Hu, L., Hierons, R.M., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Trans. Softw. Eng. **30**(1), 3–16 (2004)
21. Heydon, A., Najork, M.: Mercator: a scalable, extensible web crawler. World Wide Web **2**(4), 219–229 (1999)
22. Hieatt, E., Mee, R., Faster, G.: Testing the web application engineering internet. IEEE Softw. **19**(2), 60–65 (2002)
23. Jungmayr, S.: Testability measurement and software dependencies. In: Proceedings of the 12th International Workshop on Software Measurement, pp. 179–202. Magdeburg,Shaker Publication, Aachen (2002)
24. Kan, S.H.: Metrics and models in software quality engineering. Addison-Wesley, Reading (2003)
25. Kim, K.-H., Kim, Y.-G.: Process reverse engineering for bpr: a form-based approach. J. Inform. Manage. **23**(4), 187–200 (1998)
26. Lakhotia, K., McMinn, P., Harman, M.: Automated test data generation for coverage: Haven't we solved this problem yet? In: 4th Testing Academia and Industry Conference—Practice and Research Techniques (TAIC PART'09), pp. 95–104. IEEE Computer Society, Washington (2009)
27. Marchetto, A., Tonella, P., Ricca, F.: State-based testing of ajax web applications. In: International Conference on Software Testing Verification and Validation (ICST), IEEE Computer Society Lillehammer (April 2008)
28. McCabe, T.: A complexity measure. IEEE Trans. Softw. Eng. **2**(4), 308–320 (1976)
29. Mesbah, A., van Deursen, A.: Invariant-based automatic testing of ajax user interfaces. In: 31st International Conference on Software Engineering (ICSE). IEEE Computer Society (May 2009)
30. Miller, R.C., Bharat, K.: Sphinx: a framework for creating personal, site-specific web crawlers. Comput. Netw. ISDN Syst. **30**(1–7), 119–130 (1998)
31. Palmer, J.W.: Web site usability, design, and performance metrics. Info. Syst. Res. **13**(2), 151–167 (2002)
32. Ricca, F., Tonella, P.: Analysis and testing of web applications. In: ICSE '01: Proceedings of the 23rd International Conference on Software Engineering, pp. 25–34. IEEE Computer Society, Washington (2001)
33. Sampath, S., Bryce, R.C., Viswanath, G., Kandimalla, V., Koru, A.G.: Prioritizing user-session-based test cases for web applications testing. In: ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, pp. 141–150. IEEE Computer Society, Washington (2008)
34. Sampath, S., Sprenkle, S., Gibson, E., Pollock, L., Souter, A.: Analyzing clusters of web application user sessions. In: WODA '05: Proceedings of the Third International Workshop on Dynamic Analysis, pp. 1–7. ACM, New York (2005)
35. Sampath, S., Sprenkle, S., Gibson, E., Pollock, L., Greenwald, A. Souter: Applying concept analysis to user-session-based testing of web applications. IEEE Trans. Softw. Eng. **33**(10), 643–658 (2007)
36. Shepperd, M.J.: Foundations of software measurement. Prentice Hall, New Jersey (1995)
37. Sprenkle, S., Sampath, S., Gibson, E., Pollock, L., Souter, A.: An empirical comparison of test suite reduction techniques for user-session-based testing of web applications. In: ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance, pp. 587–596, IEEE Computer Society Washington (2005)
38. Tip, F.: A survey of program slicing techniques. J. Program. Lang. **3**(3), 121–189 (1995)
39. Warren, P., Boldyreff, C., Munro, M.: The evolution of websites. In: IWPC '99: Proceedings of the 7th International Workshop on Program Comprehension, p. 178. IEEE Computer Society, Washington (1999)
40. Wassermann, G., Yu, D., Chander, A., Dhurjati, D., Inamura, H., Su, Z.: Dynamic test input generation for web applications. In: ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis, pp. 249–260. ACM, New York (2008)
41. Weiser, M.: Program slicing. IEEE Trans. Softw. Eng. **10**(4), 352–357 (1984)
42. Wu, Y., Offutt, J.: Modeling and testing web-based applications. Technical report, George Mason University (2002)