Crawlability Metrics for Web Applications

Nadia Alshahwan, Mark Harman
University College London, UK
{mark.harman, nadia.alshahwan.10}@ucl.ac.uk

Alessandro Marchetto, Roberto Tiella, Paolo Tonella
Fondazione Bruno Kessler, Trento, Italy
{marchetto, tonella, tiella}@fbk.eu

***Abstract*—Automated web crawlers can be used to explore and exercise portions of a web application under test. However, the possibility to achieve full exploration of a web application through automated crawling is severely limited by the choice of the input values submitted with forms. Depending on the crawler's capabilities, a larger or smaller portion of web application will be automatically explored.**

In this paper, we introduce web crawlability metrics to quantify properties of application pages and forms that affect crawlability. Moreover, we show that our metrics can be used to identify the boundaries between those parts of the application that can be successfully crawled automatically and those parts that will require manual intervention or other crawlability support. We have validated our crawlability metrics on real web applications, for which low crawlability was indeed associated with the existence of pages never exercised during automated crawling.

I. Introduction

Web crawlers are a foundational technology for automating web engineering and web testing activities. Web crawling technology is best known as the first stage in web searching, forming the initial list of pages visited for subsequent indexing [11]. Crawlers are also useful in testing and validity checking, to find broken links and other bugs in websites [1]. Automated exploration of a web application under test can expose failures, such as crashes, errors and exceptions [3]. Crawlers are also important for automating the construction of models of web structures [6]. Such models can be used as the basis for test case generation [4], [18].

However, some websites are more crawlable than others [16]. We use the term crawlability in this paper to capture that property of a web site (or substructure such as an individual web page) that makes it amendable to automated web crawling. A site is completely crawlable if an automated web crawler can cover all pages in the site without any human intervention.

Complete crawlability is almost always impossible and impractical, and in some cases also undesirable, depending on the context. However, for developers, wishing to ensure that all aspects of their web application are properly exercised during automated testing, high degrees of crawlability are essential.

Automated crawling is the enabling technology used by many automated web testing techniques [4], [18]. Crawlability metrics can support and improve the effectiveness of such techniques. In fact, based on crawlability metrics, testers can prioritize those parts of the system that have low crawability for either more sophisticated techniques or for human intervention and manual testing. This can make their available test effort more effective since it can be targeted at those less crawlable parts of the system that are most likely to require such effort. Alternatively the metrics might point to parts of the website that could be re-engineered to better facilitate ongoing automated testing and maintenance. Hence software engineers need to understand and measure crawlability. For this to be possible, the first step is the provision of crawlabilty metrics.

To the best of our knowledge, our work is the first to investigate the notion of crawlability and its quantification through metrics. Preliminary results of our research have already been published [17]. However, the metrics considered in our previous work [17] did not take the crawler's capabilities into account, but instead investigated crawler capability separately, merely as an influencing factor. In the present work, we propose some metrics that can be used with different crawlers, since they incorporate the crawler's capabilities as the degree of exploration actually achieved by the crawler. More specifically, the proposed metrics combine a *dynamic* measurement of the exploration performed by a crawler with given capabilities (e.g., the code coverage or the fan-out measured during crawling) with a *static* measurement that approximates the size all possible alternative explorations (e.g., the lines of code, the cyclomatic complexity or the cyclomatic complexity after conditioned slicing).

The primary contributions of this paper are as follows:

- **Three crawlability metrics**, combining crawler-dependent (dynamic) and crawler independent (static) information. We also describe how these metrics can be used to support web testing.
- **Experimental data**, showing that the proposed metrics are a good indicator and predictor of crawlability, i.e., they indicate the presence of unexplored pages.
- The **WATT crawler**, which can be configured with variable capabilities and can be used to experiment with the application of crawlability metrics in various contexts, including web testing.

II. Crawlability

Intuitively, the crawlability of a Web application measures how difficult it is for a crawler with given capabilities to explore all the pages reachable from the home page. When such exploration involves form submission, it depends on the ability of the crawler to generate inputs that explore all different pages possibly generated in response to form submission. So

it depends on the input generation capabilities of the crawler. It also depends on the notion of "different" pages generated in response to a form submission, in that different concrete pages may be indeed instances of the same "conceptual" page, differing just by some irrelevant details about the displayed information (e.g., date and time information shown at the top of the page).

Hence, instead of an absolute definition of crawlability, we give a definition which is relative to: (1) the conceptual model of the pages to be crawled; and, (2) the crawler's capabilities:

a) **[Conceptual Web application model]***:* we consider a conceptual Web application model containing 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 the associated server side actions.

The UML Conallen model [9] of a Web application is an example of a conceptual model that is pretty close to the one referenced 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 may vary from time to time, but conceptually there is just one client page, showing the *search result*.

b) **[Crawler's capabilities]***:* we characterize a crawler by its specific crawling capabilities. Such capabilities include the algorithm used to generate input data when forms are to be filled-in to continue the exploration of the Web application and the algorithm used to recognize equivalent pages that should not be crawled again.

Common crawler's capabilities include generation of: (1) random strings; (2) legal (test) credentials; (3) numbers; (4) emails; (5) dates. 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: (1) page name or title; (2) string comparison (e.g., page diff); (3) tree comparison, considering the HTML parse trees of the pages.

A. Definition of crawlability

c) **[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.

d) **[Crawlability of a form]***:* 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, as in the conceptual model of the Web application.

Forms and form submission represent the main source of low crawlability (in fact, links are easily explored by crawlers), hence in the following we focus on form crawlability, under the assumption that once the crawlability of all forms is determined (estimated), the crawlability of the whole Web application can be also determined by aggregating the values computed for forms.

B. Metrics

Since usually the conceptual model of the Web application being crawled is unavailable, it is not possible to measure crawlability directly, by running the crawler and measuring the proportion of conceptual client pages (in the model) that are explored by the crawler after a given amount of time.

Typically, we are interested in the crawlability of a Web application for which no conceptual model is available, but for which we have the possibility to run a given crawler and to compute static and dynamic metrics to estimate the application/form crawlability.

In this case, crawlability is measured indirectly. Dynamic and static 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 (see Section V of this paper). The basic idea is to estimate crawlability of a form from the ratio of two quantities: (1) a dynamically-determined metrics related to pages the crawler actually discovered behind a form, when trying different inputs at runtime; and, (2) a statically-determined metrics related to all pages that can be generated on the server-side as a response to a form submission,

1) Static metrics: Crawlability of forms depends crucially on how input values are processed by the server-side code. Typically, on the server side input values are first validated and only when valid they trigger the generation of new pages. Moreover, under different conditions, different (conceptual) pages are produced. In fact, the server side code generates the next client page depending on the submitted input (we assume the session state to be part of the 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. In order to quantify such a variety of behaviors of the server code, we consider three metrics, having an increasing degree of sophistication:

The first, simplest metric that characterize the number of pages potentially generated is just the number of statements in the server script activated by the submission of a form:

- **STM [Statements]**: Number of statements that can be potentially executed to process the requests coming from a client-side form.

To compute STM we consider also the server components (e.g., functions) transitively invoked by the script handling the form. We resort to a simple static dependency analysis to discover such indirectly needed components. These constribute to the STM count.

The second metric we consider is McCabe's cyclomatic complexity, which measures the number of independent paths

in a program. Specifically, we use an interprocedural variant of the cyclomatic complexity, which includes also the contribution from transitively called functions, consistently with how STM is computed.

- **ICC [Interprocedural Computational Complexity]**: Number of independent paths in the interprocedural control flow graph.

In practice, for programming languages with no unconditioned jumps, we can obtain the number of independent paths of each procedure as the number of conditional and loop statements, incremented by 1. However, for called functions the increment is not applied, since the base execution path is already accounted for in the main component.

Sometimes, a single server side component manages different form submissions at the same time, or handles the same form submission differently, depending on the session state and hidden parameter values. In such cases, different statements will be responsible for processing the input received from different forms or from the same form under different session/hidden variable states. Such input will lead to different client page generation statements. In order to determine which portion of the server component is responsible for managing each alternative form/state, we take advantage of program slicing [14]. Specifically, we use conditioned program slicing [8] to determine the portion of the server side code that is executed when session and hidden parameters assume given, constant values. We then compute the interprocedural cyclomatic complexity on such conditioned slice, under the assumption that this is a substantially improved estimate of the independent page generation paths activated by a given form.

- **SICC [Sliced Interprocedural Cyclomatic Complexity]**: ICC computed on the conditioned slice of the main server component activated by form submission.

Let us consider the PHP server component shown in Figure 1. The component is both in charge of (a) presenting a form to allow a new user to register to a web application; and, (b) validating form values submitted by the user and possibly creating a new user in the system.

Looking at the code, we can see that if the component is activated by an HTTP-GET request, the “registration” form is presented to the user (line 20). If an HTTP-POST request is submitted instead, form parameters are checked and if they are correct, the actual registration task is executed (code not shown) and the user is directed to a confirmation page.

When an HTTP-POST request is submitted, `$_SERVER["REQUEST_METHOD"]` is set to ”POST” and `$_POST["action"]` is set to ”register” (because of the hidden field ’action’ set to ’register’ at line 32). When computing the conditioned slice of the main component under these assumption, we obtain that the condition at line 20 evaluates to false, regardless of the user input, hence the ’then’ branch can be sliced away; similarly, conditions at lines 35 and 36 evaluate to true; the remaining conditions depend on user inputs. As a result of slicing away all statements

```
1  <?php
2  function valid_data($id,$pwd1,$pwd2) {
3    if (empty($id)) {
4      return "userid must not be empty";
5    }
6    if (preg_match("/[^a-zA-Z0-9]/",$id)) {
7      return "userid must contain alphanums only";
8    }
9    if (empty($pwd1) || $pwd1 != $pwd2) {
10     return "passwords must not be empty and must
           be equal";
11   }
12   return NULL;
13 }
14
15 function userid_exists($id) { // ICC=3
16 ... }
17
18 session_start();
19 $request = $_SERVER['REQUEST_METHOD'];
20 if ($request == "GET") {
21   if (!isset($_SESSION['message'])) {
22     $_SESSION['message'] = "";
23   }
24 ?>
25   <h1>New User</h1>
26     <form action="registration.php" method="post">
27       <p style="color:red;"><?php echo $_SESSION['
             message']; ?></p>
28       Name: <input name="userid" type="text"/><br
             />
29       Password: <input name="passwd1" type="
             password"/><br/>
30       Password Confirmation: <input name="passwd2"
              type="password"/><br/>
31       <input type="submit" value="Register"/>
32       <input name="action" type="hidden" value="
             register"/>
33     </form>
34 <?php
35 } else if ($request == "POST") {
36   switch($_POST["action"]) {
37     case "register":
38       $userid = $_POST['userid'];
39       $passwd1 = $_POST['passwd1'];
40       $passwd2 = $_POST['passwd2'];
41       $tmp = valid_data($userid,$passwd1,$passwd2)
             ;
42       if (!is_null($tmp)) {
43         $_SESSION['message']=$tmp;
44         header("Location: registration.php");
45       } else if (userid_exists($userid)) {
46         $_SESSION['message']="user id already
               exists";
47         header("Location: registration.php");
48       } else {
49         $_SESSION['message']="";
50         $_SESSION['userid'] = $userid;
51         // registration code ...
52         header("Location: confirm_registration.php
               ");
53       }
54       break;
55     default:
56       header("Location: internal_error.php");
57   }
58 } else {
59   header("Location: internal_error.php");
60 }
61 ?>
```

Fig. 1. Example of PHP component

between 20 and 35, four conditions are dropped or have a known value (20, 21, 35, 36) in the main component, such that SICC = ICC - 4 = 8. Similarly, when an HTTP-GET request is submitted, we obtain SICC = ICC - 10 = 2.

2) Dynamic metrics: Static metrics do not account for the ease of exploration of the pages generated by a crawler with given capabilities. In order to take this into account, we define two metrics about the level of exploration reached by a crawler with given capabilities, when it generates input data for a given form.

The first, simpler dynamic metric is:

- **CSTM [Covered Statements]**: Number of statements of server components executed to process the requests coming from a client-side form.

When a form is exercised by the user in the browser or by a crawler, some server-side components are triggered by the application and some code is executed in order to process and answer the requests coming from the form. It's worth noticing that according to the inputs used in the requests and the state of the application, different pieces of code can be executed. Given a form *f* of an application, to measure CSTM we need to: (i) execute the application by exercising the form *f*, and (ii) profile the execution to collect information about the code statements executed for processing the requests related to the form *f*.

The second dynamic metric we consider is:

- **FOUT [Fan OUT]**: Number of distinct client pages downloaded by the crawler when a given page/form is reached.

Crawlers with poor capabilities will result in low FOUT values when they reach forms that are hard to crawl.

3) Crawlability metrics: To measure crawlability we combine dynamic metrics related to the set of pages actually explored with static metrics, indicating the number of client pages possibly generated when a form is submitted to the server. Specifically, we consider the following three combinations as the most interesting ones:

$$CRAW_0 = \frac{CSTM}{STM} \quad (1)$$

$$CRAW_1 = \frac{FOUT}{ICC} \quad (2)$$

$$CRAW_2 = \frac{FOUT}{SICC} \quad (3)$$

$CRAW_0$ is the well-known coverage ratio. $CRAW_1$ and $CRAW_2$ give the number of distinct client pages explored by a given crawler per independent path in the associated server component.

The variant with SICC makes use of a more precise estimate of the independent paths that can be actually traversed when a form is submitted from a client in a specific state.

III. Application to Web Testing

Testing tools that aim at page coverage are typically based on web crawling [4], [18]. Given a base URL, they automatically navigate links starting from that URL and use automated input generation techniques 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. The testing tool is making decisions that could affect page coverage. In particular, the testing tool may generate inadequate inputs when exploring specific areas of the web application. Directing the tester to these areas could be useful to focus the human effort necessary for improving the quality of testing and achieving higher coverage, while saving time and resources.

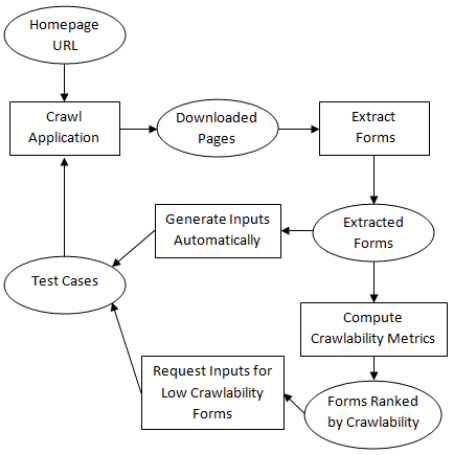

Fig. 2. Application of crawlability metrics to web testing

We propose an application of our crawlability metrics that provides the tester with this kind of information. Crawlability metrics point the tester to certain areas of the application that could possibly lead to new unexplored pages. Furthermore, they guide the tester by better focusing and limiting the manual effort spent during the testing activity.

Figure 2 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 tester and crawling starts from that point. The crawler downloads the web page and identifies any forms it contains. Input values for these forms are automatically generated by taking advantage of the crawler's capabilities. The crawler then submits those inputs and the process is repeated for each encountered page.

In addition to the crawling loop described above, the identified forms are analyzed and crawlability metrics are calculated for each form. Forms are then ranked by increasing crawlability and the ranking is displayed to the tester. Top ranked (least crawlable) forms are further inspected by the tester to determine if additional inputs can be used to extend the web application portion that was explored automatically. The input values provided by the tester 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 tester when more manual intervention is needed. The process can continue until the tester is satisfied or no new

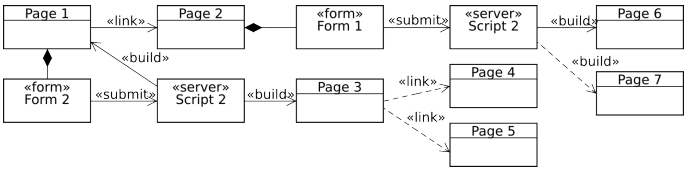

Fig. 3. A simple demonstration of how crawlability metrics can be used to achieve more coverage

pages are encountered.

Figure 3 is a simple demonstration of the result of the approach. Client pages of the web application downloaded automatically by the crawler are represented as UML classes, following the notation by Conallen [9]. A composition relation associates each client page with the forms it contains (e.g., *Page 1* and *Form 2*). Links to navigate from a client page to another are stereotyped as ≪*link*≫. The server side code executed in response to form submission is stereotyped as ≪*server*≫ and is accessed from forms via ≪*submit*≫ relationship (see server code *Script 1*). The client pages constructed in response to form submission by the server code are connected to the server pages via ≪*build*≫ relationships (see client page *Page 3*).

When the crawler cannot explore a given page, this is indicated in Figure 3 as a dashed relationship between classes. For instance, *Page 3* is never visited by the crawler and one reason for that could be that the automatically generated input values for *Form 2* do not pass the validation performed on the server (by *Script 2*), such that navigation to *Page 3* does not take place and the user is redirected to *Page 1*. As a consequence of the unreachability of *Page 3*, the client pages *Page 4* and *Page 5* are also never explored by the crawler. When input values are automatically generated for *Form 1*, the executed server code (*Script 1*) constructs the client page *Page 6*, but the crawler fails to make the server code generate another possible client page produced in response to the submission of *Form 2*, i.e., *Page 7*. One reason for that could be that very specific values are required in order for the server code to generate this page and the capabilities of the crawler might be insufficient to generate such inputs (within a reasonable amount of time).

The FOUT metrics for *Form 1* and *Form 2* are in this example $FOUT(Form1) = FOUT(Form2) = 1$ (the edges to count are the ≪*build*≫ relationships outgoing from the server code handling each form submission). Let us assume that after conditioned slicing the code in *Script 1* and in *Script 2* contains two alternative paths, such that $SICC(Form1) = SICC(Form2) = 2$. The corresponding values of the crawlability metrics $CRAW_2$ for *Form 1* and *Form 2* for this example will be $CRAW_2(Form1) = CRAW_2(Form2) = 0.5$. In fact, in the first case only half of the client pages generated in response to form submission have been explored, while no new client page has been explored when the second form was crawled (the user is redirected to the home page). By providing additional inputs, the tester may be able to explore also *Page 3*, and from here *Page 4* and *Page 5*, as well as *Page 7*. In general, low crawlability forms indicate the need for the tester's attention, since the crawler was able to explore only a small fraction of the code/independent paths on the server and the uncovered code/paths may indeed be associated with distinct client pages that are still to be explored and may require additional input to be reached and to be properly tested. Notice that in this example we consider $CRAW_2$ as crawlability metric but the same could happen for the other crawlability metrics.

IV. The WATT crawler

We have developed a crawler, called WATT (Web Application Testing Tool) that can be used to experiment with our approach. WATT can be configured with different combinations of capabilities. It also implements some similarity algorithms to determine if a page is new or should be regarded as a re-occurrence of a previously visited page.

The tool can be configured to use one of three input generation strategies: *default*, *empty* and *random*. When the 'default' strategy is chosen, the value in the `defval` attribute of a field in the `form` definition is used to fill out the field. The 'empty' strategy uses only empty strings. When the 'random' strategy is used, the crawler chooses a random value for enumerable field types (such as drop-down menus, checkboxes and radio buttons). Random strings are generated for unbounded field types such as text. WATT can additionally be configured to use more advanced input generation methods, when using random. Email addresses, numbers and valid dates can also be generated randomly, in addition to strings. The type of input to be generated is chosen randomly, with higher priority given to strings and numbers.

WATT can be configured to automatically login to applications that require authentication. A valid user name and password pair have to be provided by the user. The crawler will then automatically recognize login forms when crawling the application and will use the provided credentials to login. A login form is identified by the presence of one field of type password.

The crawler starts from a URL provided by the tester and explores the application in a depth first visit. The next action (form or link) is chosen randomly from the set of untried available actions on the last visited page. If all actions on the last visited page have been visited before, the crawler selects randomly from all actions. A trace is saved, with the sequence of actions performed together with information about encountered forms and links, input values used. A trace is terminated when a page with no links is reached, when no new page is encountered, or when a user defined number of actions is reached. The process is repeated (because it involves some non deterministic choices) until a user defined time limit is reached.

The structure of the output HTML is used to decide if a visited page is conceptually new or has been visited before. The sequence of HTML tags in the output page is extracted and compared to the structure of previously visited pages.

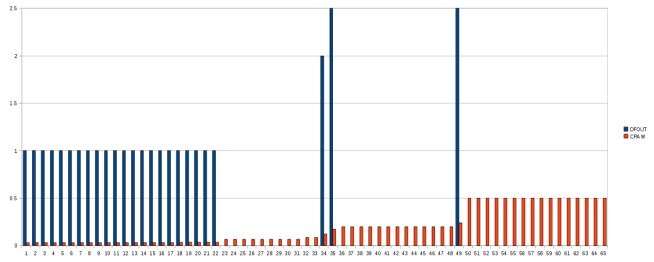

Fig. 4. Histogram showing DFOUT and $CRAW_2$ for forms of WebChess

In some cases two pages can be conceptually the same but have slightly different structure. A simple example of this is a page that displays items in table format. If two instances of this page are encountered but contain a different number of items in each table, a structural comparison would put them in different equivalence classes. To avoid (or minimize) this, we use a similarity measure. If the similarity measure is above a predefined threshold, the two pages are considered to be of the same class. For two pages P_i and P_j, their similarity is calculated using the following formula:

$$Sim = 2 \times |LCS| / (|Seq(P_i)| + |Seq(P_j)|)$$

Where $Seq(P_i)$ and $Seq(P_j)$ are the sequences of HTML tags for pages P_i and P_j, respectively; LCS is the longest common subsequence between $Seq(P_i)$ and $Seq(P_j)$. If no pages that satisfy the similarity measure are found, the new sequence is assigned to a new class.

V. Evaluation

The aim of the experiments we conducted was to validate empirically the proposed crawlability metrics. Specifically, the *goal* is to determine whether low crawlability values are associated with the existence of unexplored parts of a web application.

A. Research questions

The overall goal of the evaluation can be decomposed into the following research questions:

- **RQ1 [Crawlability as indicator of unexplored pages]**: Are the crawlability metrics good indicators of crawlability, thus of the presence of unexplored pages?
- **RQ2 [Role of conditioned slicing]**: Does conditioned slicing improve crawlability metrics based on the computation of the interprocedural cyclomatic complexity?

RQ1 is the main research question that motivated our work: crawlability metrics are supposed to be useful to highlight forms that lead to unexplored pages. We answer to RQ1 by means of two pieces of evidence: (1) we verify that the ranking by increasing crawlability correspond to (i.e., negatively correlates with) the ranking of forms by unexplored pages; and, (2) we train a binary classifier based on crawlability metrics to discriminate completely explored forms from partially explored ones. The first piece of evidence is purely observational, since it deals just with the existence of a negative correlation. The second piece of evidence is much stronger, since metrics are used as predictors of forms that have been explored only partially during crawling. Hence, the second part of the validation is clearly more important and relevant for testing.

RQ2 deals with our conjecture that without slicing the cyclomatic complexity is a poor indicator of the independent paths that lead to different client page generation. This is due to the common practice of implementing multiple behaviors in a single server component to accommodate different requests or requests made in different states. For this research question we use the same pieces of evidence as for RQ1, but we focus on the comparison between SICC and ICC. We expect an improvement (in correlation and/or predictive power) when SICC is used instead of ICC.

B. Metrics

To answer the research questions above, we compute the crawlability metrics described in Section II then, we manually determine the number of missing pages among those outgoing from a form.

These are the client pages built by the server component executed after form submission that are never visited by the crawler. In Figure 3 they are the client pages reachable from the $\ll$*server*$\gg$ components *Script 1* and *Script 2* along dashed lines. We count the number of such client pages, missing in the model downloaded by the crawler, and indicate them as the metrics DFOUT:

- **DFOUT [Delta Fan OUT]**: Number of distinct client pages that the crawler is unable to download when a given form is reached.

To answer RQ1, we measure the Spearman's rank correlation between crawlability metrics and DFOUT. We expect low crawlability to be associated with high DFOUT, hence a negative correlation between crawlability and DFOUT. We also train a threshold-based classifier on our metrics. More precisely, we apply the *leave-one-out* cross-validation procedure, by performing the following steps: (a) forms are manually classified as crawlable or not, depending on the conditions $\text{DFOUT} = 0, \text{DFOUT} \neq 0$; (b) a classifier is trained on all but one sample, by determining the threshold for the crawlability metric that maximizes the F-measure (harmonic mean of precision and recall); (c) the class predicted for the sample that was left out is marked as correctly or incorrectly classified; (d) steps (b), (c) are repeated for all the samples. The ratio of correctly classified samples over all samples gives the accuracy of the classifier.

We compare our results with the accuracy obtained by the 'a priori' classifier (a.k.a. biased classifier), i.e. the classifier which predicts whether a sample belongs to a class or the other based on the a-priori probabilities. To answer RQ2 we consider correlation and prediction accuracy with and without conditioned slicing (i.e., using SICC vs. ICC).

Web app	PHP files	Lines of PHP code	Downloads
FAQForge	19	834	5,264
WebChess	24	2,701	24,751
News Pro	30	5,473	n.a.
TimeClock	62	14,980	22,328

TABLE I
WEB APPLICATIONS SUBJECTED TO AUTOMATED CRAWLING IN THE EXPERIMENTS

	Discovered		Total		Requests on
Web app	Pages	Forms	Traces	Requests	Forms
FAQForge	22	23	6,676	99,340	27,116
WebChess	28	65	784	25,228	12,095
News Pro	58	17	12,033	89,892	16,979
TimeClock	65	21	112	111,463	38,751

TABLE II
CRAWLER ACTIVITY PER APPLICATION

Web app	Rho	P-value
$CRAW_0$ = *Coverage*		
FAQForge	-0.049	0.823
WebChess	0.172	0.172
News Pro	-0.661	0.003
TimeClock	-0.395	0.003
$CRAW_1$ = *FOUT / ICC*		
FAQForge	-0.23	0.290
WebChess	-0.05	0.681
News Pro	-0.76	0.002
TimeClock	-0.87	2.11e-7
$CRAW_2$ = *FOUT / SICC*		
FAQForge	-0.09	0.681
WebChess	-0.72	1.41e-11
News Pro	-0.54	0.04
TimeClock	-0.86	5.22e-7

TABLE III
SPEARMAN'S RANK CORRELATION RESULTS

C. Subjects

For the evaluation we applied our metrics to four open-source PHP applications taken from code repositories available on the Internet (e.g. Sourceforge[1]) and already used by other researchers in their works about Web testing [3], [2], [23]: FaqForge 1.3.2, Webchess 0.8.4, Utopia News Pro 1.4.0 and TimeClock 1.0.4. Table I summarizes the size of the applications, in terms of their PHP files and PHP-executable lines of code. It also reports the number of user downloads (if provided by the repository). FaqForge is a small application to create and manage documents. Webchess is a medium size application that allows a community of users to play chess together by means of a web interface. Utopia News Pro is a medium size template-based news management system. Timeclock is a medium size web application that provides features to track the employee working time for daily activities and vacation.

D. Experimental procedure

For each application, the following steps have been performed:

1) We downloaded and installed the application.
2) We ran the crawler for 10 hours, a summary of the activity performed is listed in Table II).
3) We ran our tool to analyze the PHP code of the application to measure the crawlability metrics for the set of discovered forms.
4) We manually determined DFOUT for each discovered form (this required both executing the applications and looking at the application code). This task has been performed by one of the authors not involved in crawling and in crawlability measurement.
5) We trained a threshold-based classifier on each of our metrics and evaluated its accuracy. We correlated our metrics with DFOUT.

[1]http://sourceforge.net

The WATT tool was configured with the following capabilities activated: *random string*, *credentials* and *numbers*. In fact, our previous studies [17] indicate that this combination is associated with the best crawler's performance.

E. Experimental results

Figure 4 shows the manually determined values of DFOUT (in blue/dark gray) and $CRAW_2$ (in red/light gray) for each form of WebChess. Values are ordered from left to right by increasing values of the metric $CRAW_2$. Similar plots have been produced for the other three applications and for the alternative crawlability metrics $CRAW_0$, $CRAW_1$. From these plots, we can argue that a threshold-based classifier is quite likely to be able to discriminate fully explored from partially explored forms, based on the crawlability metrics. A negative correlation is also quite apparent: $CRAW_2$ increases in the region where DFOUT goes to zero.

Spearman's ρ correlation coefficients between the ranking by increasing DFOUT and increasing $CRAW_i$ with $i = 0, 1, 2$, are listed in Table III, along with the corresponding p-values.

We can notice that: (a) a moderate to strong negative correlation exists (statistically confirmed) between DFOUT and $CRAW_2$ for WebChess (ρ =-0.72), News Pro (ρ=-0.54) and TimeClock (ρ=-0.86); (b) a strong negative correlation also exists (statistically confirmed) between DFOUT and $CRAW_1$ for News Pro (ρ=-0.76) and TimeClock (ρ=-0.87); (c) a moderate negative correlation exists (statistically confirmed) between DFOUT and $CRAW_0$ for News Pro (ρ=-0.66) and TimeClock (ρ=-0.39); (d) no correlation could be established between DFOUT and our metrics for the application FAQForge.

The accuracy (ratio of successfully classified forms) for classifiers trained on each of our metrics is shown in Table IV along with the baseline accuracy of the 'a priori' classifier. All classifiers perform quite poorly on FAQForge. This is an expected result, given the lack of any significant correlation between crawlability metrics and DFOUT for this application. If we restrict our analysis to the three applications (bottom

Web app	$CRAW_0$	$CRAW_1$	$CRAW_2$	a Priori
FAQForge	0.35	0.39	0.61	0.87
WebChess	0.51	0.51	0.95	0.62
News Pro	0.82	0.94	0.88	0.59
TimeClock	0.92	0.95	0.95	0.75

TABLE IV
ACCURACY FOR CLASSIFIERS TRAINED RESPECTIVELY ON $CRAW_0$, $CRAW_1$ AND $CRAW_2$ VS. THE 'A PRIORI' CLASSIFIER

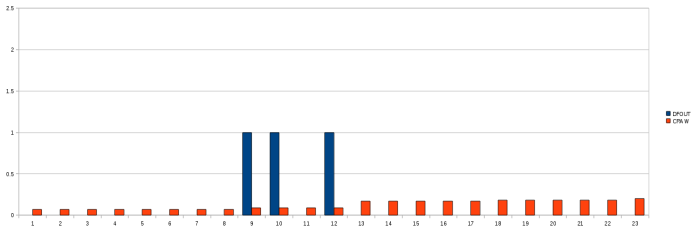

Fig. 5. Histogram showing DFOUT and $CRAW_2$ for forms of FaqForge

of Table IV) for which a correlation does exist between crawlability metrics and DFOUT, we can notice that: (a) in two out of three cases all crawlability metrics are superior to the 'a priori' classifier; (b) in all three cases $CRAW_2$ is superior to the 'a priori' classifier; (c) in two out of three cases $CRAW_2$ is superior to $CRAW_0$ and $CRAW_1$;

RQ1: These results provide support to our main research hypothesis, namely that crawlability is an indicator of unexplored pages. This is confirmed by the correlation between crawlability metrics and DFOUT and by the accuracy of the classifier trained on our crawlability metrics. The support is partial, since it was obtained for three subjects (WebChess, News Pro, TimeClock) out of four using $CRAW_2$ (see Table III), and two (News Pro, TimeClock) out of four for $CRAW_1$ and $CRAW_0$. For the application FAQForge, crawlability metrics are not good indicators of unexplored web site portions.

RQ2: Computing the interprocedural cyclomatic complexity after conditioned slicing improved the ranking by increasing crawlability of WebChess, for which the correlation is statistically significant. For TimeClock the difference of *Rho* between SICC and ICC is negligible. For News Pro the use of conditioned slicing leads to a decrease of the correlation. Hence, our research hypothesis about the usefulness of conditioned slicing is only partially supported by the experimental results. Such usefulness seems to be quite application-dependent. We argue that while dispatcher-based web applications, where a single script handles all requests, are expected to benefit from conditioned slicing, modern web applications, that are already properly modularize, are unaffected or negatively affected (e.g., when the sliced code remains highly complex, due to the input processing it performs) by conditioned slicing.

F. Discussion

We split the discussion by web application analyzed:

FaqForge. Only a small portion of this application is devoted to checking and processing the inputs coming from the user through forms. In fact, such a task is mainly delegated by the application to the database access layer (e.g., an INSERT may fail if inappropriate data is passed, which later results in empty or error pages displayed to the user). This feature makes the crawler able to explore most discovered forms exhaustively. In fact, the observed DFOUT is greater than 0 only for 3 out of 23 forms discovered by the crawler.

The lack of a good performance of our metrics as indicators of crawlability and hence of unexplored pages can be mainly due to a set of forms which have low crawlability, but are not associated with any unexplored page. They correspond to the leftmost histograms in the plot for FaqForge shown in Figure 5. The low crawlability of these forms is due to the existence of lots of statements / many independent paths in the server code that are not eliminated by conditioned slicing. Such paths are associated with presentation variants of the generated pages, which do not depend on the input. In fact, the branches taken at the decision points for such paths are known, once the state of the form originating the server request is set, but the conditioned slicing technique we are using is not smart enough to determine that these conditions are not open for the forms being analyzed. If we remove these forms from the ranking, we obtain *rho* = -0.66 and p-value = 0.006607.

Visual inspection of the histogram in Figure 5 (left) reveals that the ranking is indeed useful and meaningful, with the exception of the leftmost forms. In fact, the three forms with DFOUT > 0 are at position 9, 10 and 12, out of 23. We can notice that between position 1 and 12 the crawlability metrics assume values which are very close to each other, while after position 12 there is a substantial increase. This corresponds roughly to a partition of the forms into low crawlability and high crawlability ones. We can notice that all three forms having DFOUT > 0 belong to the low crawlability group.

WebChess. We observed that only $CRAW_2$ correlates with DFOUT and is a good predictor of the presence of unexplored pages. By analyzing the application and its code, we noticed that WebChess can be considered as a quite monolithic application. Most of the features (user preferences update, games management, etc.) are controlled by a huge PHP page ("mainmenu.php"), accessed by the user from the main client page by means of several forms (e.g., update user preference form, accept or reject invitation to play, etc.). As "mainmenu.php" contains the code that implements both logic and interface of several features, the complexity of its code (measured by ICC) is definitely over-estimating the different client pages that may be generated by each specific form. The slicing-based metric SICC, instead, provides a better estimate, since it isolates the code related to each form and it considers only the complexity of this code.

News Pro and **TimeClock**. We observed that all the crawlability metrics correlate with DFOUT in a statistically significant way and are good predictors of the presence of unexplored pages. By looking at the application code, we noticed that this is well-organized and structured for both applications. In fact, each application feature (e.g., user account management and office creation for TimeClock) executed from a form is implemented by a specific PHP code in a dedicated page (e.g.,

in TimeClock the "createUser.php" page controls the creation of a new user account) and a large amount of PHP code is devoted to check and process user inputs. Hence, in News Pro and TimeClock the control-flow complexity of each PHP code related to a form is a good indicator of the number of distinct client pages that can be generated, even when conditioned slicing is not used.

Overall Considerations Overall, results indicate that, a part from a subject (namely FAQForge), crawlability metrics are good indicators of the presence of unexplored pages. The ranking produced by the crawlability metrics or the output of the classifier are potentially very useful to understand which parts of the application need further (manual) crawling and testing effort. Metrics performance varies with the typology of the application under test. Monolithic applications which comprise a single or few huge components with a lot of responsibilities require a quite resource-consuming metric such as $CRAW_2$, which leverages the conditioned slicing approach to discern every single feature related to each form. On the other hand, applications which split clearly their features on different components require cheaper metrics such as $CRAW_0$ or $CRAW_1$. Furthermore $CRAW_1$ performs better than $CRAW_0$ again at the price of a slightly more demanding algorithm.

G. Threats to validity

External validity threats affect the generalization of results. The main limitations to the generalization of the obtained results are related to the limited number of applications considered and their representativeness with respect to the application domain and characteristics (e.g., we only considered four PHP applications). Further replications of the experiment can better support the obtained results. However, we chose four real applications belonging to different domains so as to make the context of our experiment as realistic and effective as possible.

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, the manual task involved in the computation of DFOUT. To limit this threat we tried to adopt a shared procedure with well defined guidelines to decide when a client page generated by the server is absent from those downloaded by the WATT crawler.

Construct validity threats concern the relationship between theory and observation. Our definition of the crawlability metrics CRAW was motivated by arguments about its dependence on the crawler's capabilities and on static metrics computed on the scripts handling the forms. However, different definitions of crawlability metrics may lead to different results.

Conclusion validity threats concern the relationship between the treatment and the outcome. We used a widely adopted, non-parametric statistical technique to determine the correlation between the ranking by crawlability and the desired ranking (by DFOUT), i.e., Spearman's rank correlation. We also used the standard cross-validation procedure to assess the performance of the classifier.

VI. Related work

In web applications, metrics have been used especially for usability, maintainability and evolution. Emad Ghosheh et al. [12] 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. [22] created a tool to collect a number of metrics to measure web application evolution over an interval of time. Palmer [20] 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. [10] conducted a survey of web application metrics that can be used in improving content. Bellettini et al. [4] presented TestUML, a tool that implements a testing technique for web applications in which metrics (e.g., number of pages or number of objects) are used to evaluate the coverage level and decide when to stop the testing process. To our knowledge, our proposal of metrics is the first one that addresses the crawlability of pages and forms directly.

Testability metrics have been defined and used for traditional software such as Object Oriented software for predicting testing effort. For instance, Bruntink and Deursen [7] evaluated and defined a set of testability metrics for Object Oriented programs and analyzed the relation between classes and their JUnit test cases. Jungmayr [15] suggests that testability metrics can be used to identify parts of the application causing a lower testability level by analysing test critical dependencies. However, their metric is related to fault prediction while in our case we define a crawlability metric to identify areas that could lead to additional coverage.

Available open source and commercial crawlers provide a way to navigate a web application to download its structure or report broken links. For example, webSPHINX [19] is an open source customizable spider that provides basic functionalities. JSpider[2] is another open source crawler that records a web application's structure to a database. However, both crawlers do not provide any support for automated form filling and submission. Teleport Pro[3] 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 an application. It also parses JavaScript to extract links. Benedikt et al. [5] developed the crawler VeriWeb that can be provided with input profiles that are used to fill forms. Girardi et al. [13] conducted a comparison of crawlers. They concluded that: (i) the crawlers have different strengths and weaknesses; (ii) some commercial crawlers offer more completeness; and (iii) the ability of crawlers to cover an application is closely related to their capabilities.

Raghavan and Garcia-Molina developed the crawler HiWE [21], that has quite advanced capabilities. This crawler uses

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

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

user provided values, data sources and built-in input categories (e.g. email, address) to fill forms. To identify error pages, HiWE uses a policy to identify the most significant part of the output HTML page (e.g. middle frame, main table) and string matching to identify common error messages. Our crawler uses HTML structure and a similarity measure to identify conceptually different pages. Marchetto et al. [18] reverse engineer a model for Web 2.0 applications by meqans of dynamic analysis. Coverage criteria are then applied to the model to extract effective test cases.

Our previous work on crawlability metrics [17] focused on the relationship between crawler's capabilities and level of web site exploration achieved. We identified the key capabilities necessary to maximize the chances of exploring a large portion of the application under test. This paper extends our previous work in four main directions: (1) a novel way to compute crawlability metrics, which takes the crawler's capabilities into account by combining dynamic and static exploration metrics; (2) a novel crawlability metrics, based on the computation of code coverage; (3) the WATT tool, which implements our approach to crawlability based testing; (4) novel experimental results, on additional case studies.

VII. Conclusions

Automated web crawling is a foundational technology for the automation of many tasks associated with web system design, maintenance and testing. Using web crawlability metrics, a software engineer can assess the degree of automated crawling that can be achieved, in order to improve the testing activity by complementing automated crawling with manual exploration of the low crawlability portion of the web application under test.

This paper introduced crawlabilty metrics that are aimed at providing support to the tester, who uses a crawler to automate the testing process, but who needs also information about where and why the crawler is likely to perform badly. We validated the proposed metrics using a set of experiments on real world web applications, using our tool WATT. Results indicate that our crawlability metrics are, indeed, well correlated to true web site crawlability for those sites studied and that they are good predictors of the presence of unexplored pages. Simple crawlability metrics, such as $CRAW_0$, are quite effective with well modularized web sites. On the other hand, dispatcher-based web sites, which involve large scripts, responsible for the processing of multiple forms and conditions, may require the use of more sophisticated metrics, such as $CRAW_2$, which resort to conditioned slicing.

References

[1] E. Al-Masri and Q. H. Mahmoud, "Investigating web services on the world wide web," in *Proceedings of the* 17^{th} *International Conference on World Wide Web (WWW' 08)*, J. Huai, R. Chen, H.-W. Hon, Y. Liu, W.-Y. Ma, A. Tomkins, and X. Zhang, Eds. Beijing, China: ACM, Apr. 2008, pp. 795–804. [Online]. Available: http://doi.acm.org/10.1145/1367497.1367605

[2] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Practical fault localization for dynamic web applications," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 265–274.

[3] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in web applications using dynamic test generation and explicit-state model checking," *IEEE Transactions on Software Engineering*, vol. 36, pp. 474–494, 2010.

[4] 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*. New York, NY, USA: ACM, 2005, pp. 1694–1698.

[5] 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.

[6] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph structure in the web," *Computer Networks*, vol. 33, no. 1-6, pp. 309 – 320, 2000. [Online]. Available: http://www.sciencedirect.com/science/article/B6VRG-40B2JGR-S/2/12aa9d9476c06da265c9686161c86908

[7] M. Bruntink and A. van Deursen, "An empirical study into class testability," *J. Syst. Softw.*, vol. 79, no. 9, pp. 1219–1232, 2006.

[8] G. Canfora, A. Cimitile, and A. D. Lucia, "Conditioned program slicing," *Information and Software Technology*, vol. 40, no. 11-12, pp. 595 – 607, 1998.

[9] J. Conallen, *Building Web Applications with UML*. Reading, MA: Addison-Wesley Publishing Company, 2000.

[10] D. Dhyani, W. K. Ng, and S. S. Bhowmick, "A survey of web metrics," *ACM Comput. Surv.*, vol. 34, no. 4, pp. 469–503, 2002.

[11] J. Edwards, K. S. McCurley, and J. A. Tomlin, "An adaptive model for optimizing performance of an incremental web crawler," in *Proceedings of the* 10^{th} *international conference on World Wide Web (WWW'01)*, 2001, pp. 106–113. [Online]. Available: http://doi.acm.org/10.1145/371920.371960

[12] 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*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 1155–1158.

[13] C. Girardi, F. Ricca, and P. Tonella, "Web crawlers compared," *International Journal of Web Information Systems*, vol. 2, pp. 85–94, 2006.

[14] M. Harman and R. M. Hierons, "An overview of program slicing," *Software Focus*, vol. 2, no. 3, pp. 85–92, 2001.

[15] S. Jungmayr, "Testability measurement and software dependencies," in *Proceedings of the 12th International Workshop on Software Measurement*. Aachen: Magdeburg,Shaker Publ., 2002, pp. 179–202.

[16] J. Madhavan, D. Ko, L. Kot, V. Ganapathy, A. Rasmussen, and A. Y. Halevy, "Google's deep web crawl," *The Proceedings of the Very Large Database Endowment (PVLDB)*, vol. 1, no. 2, pp. 1241–1252, 2008. [Online]. Available: http://www.vldb.org/pvldb/1/1454163.pdf

[17] A. Marchetto, R. Tiella, P. Tonella, N. Alshahwan, and M. Harman, "Crawlability metrics for automated web testing," *STTT*, vol. 13, no. 2, pp. 131–149, 2011.

[18] 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: IEEE Computer Society, April 2008.

[19] R. C. Miller and K. Bharat, "Sphinx: a framework for creating personal, site-specific web crawlers," *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 119–130, 1998.

[20] J. W. Palmer, "Web site usability, design, and performance metrics," *Info. Sys. Research*, vol. 13, no. 2, pp. 151–167, 2002.

[21] S. Raghavan and H. Garcia-Molina, "Crawling the hidden web," in *Proceedings of the 27th International Conference on Very Large Data Bases*, ser. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 129–138. [Online]. Available: http://portal.acm.org/citation.cfm?id=645927.672025

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

[23] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 32–41. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250739