Automatic generation of test drivers for model inference of web applications

Karim Hossen*, Roland Groz, Catherine Oriat and Jean-Luc Richier
Université de Grenoble,
F-38402 St Martin d'Hères Cedex, France
{lastname}@imag.fr

***Abstract*—In the "Internet of Services" (IoS) vision of the Internet, applications are developed as services using the web standards. Model-based testing combined with active model inference is one of the methods to test the applications pretty automatically, in particular to look for vulnerabilities. But one part still needs to be written manually, the test driver. It contains an abstraction of the real application and the methods to interact with the system at abstract and concrete level. We propose a generic abstraction of the web applications and an approach to generate the corresponding test driver automatically using a crawler to identify the needed information.**

Keywords: model inference, test driver generation, crawling method, web application

I. MODEL INFERENCE

In the IoS (Internet of services) vision, information is provided to the user as a service. Applications which were developed as a single block are now built by assembling components provided by various service providers. The complexity of this kind of application, its architecture and the fact that most of the time a formal model is not available, make model-based testing and verification difficult and time-consuming.

However, in a testing context, partial models can actually be retrieved, even for black box components, from observation gathered in the testing process. Model inference – that is, methods to automatically derive models from observation of the behaviors – can seriously help to test services by providing models of the components automatically.

Model inference is commonly divided into two categories: active and passive inference. Passive learning learns only from a given set of observations whereas active learning is free to interact with the tested system. In the case of web applications, we are free to interact with it through its web interface. As active learning converges faster, has better results than passive learning and is usable for website, it is the best choice.
Angluin's well-known algorithm (L*) [1] is in the active learning category and can infer a deterministic finite automaton (DFA). This algorithm has often been used as a base for developing enhanced model inference algorithms.

A. *Extended Models*

Web-based applications use a lot of structured messages to communicate. This makes application behavior mainly dependent on the inputs and outputs (I/O) and not only on the states. Basing the alphabet on the I/O value pairs would produce a large number of transitions and states and it would be highly inefficient. In order to use adapted models, we must have an efficient model which considers inputs and outputs, which may be structured.

Various methods have been proposed to enhance Angluin's algorithm to infer a Finite State Machine (FSM) with inputs and outputs [2]. In [3], the model was enhanced again to consider input/output parameters with finite (or infinite) domains; this is called Parameterized Finite State Machine (PFSM).

We first define a suitable and generic enough model for web application. In this paper, we consider a finite state machine with parameterized inputs and outputs.

An extended finite state machine (EFSM) *M* is a tuple (*X*, *Y*, *S*, and *T*) where *S* is a finite set of states which contains the initial state *s0*, *X* and *Y* are the input and output symbol sets, respectively. Each symbol has a finite set of parameters. *T* is a finite set of tuples (*s1*, *s2*, *x*, *y*, *pr*), e.g. transitions between states in S, where:

- *s1* and *s2* are the initial and final states of the transition, respectively;
- *x* is the input symbol of the transition;
- *y* is the output symbol of the transition;
- *pr* is the function defined over the input parameters of *x* and responsible to process their values and then output the values associated to the symbol *y*.

Note that this is a restricted form of classical EFSM since we do not need to consider stored variables and updates.
In the next section, we demonstrate how the inputs and outputs of a web application can be mapped as input and output symbols, respectively.

* Work funded by project SPaCIoS (n°257876, FP7-ICT-2009-5, ICT-2009.1.4: Trustworthy ICT)

B. *Issues*

Independently of the models presented before, the Angluin-based approaches need to communicate actively and several times with the application during the inference process. The main problem is that inference works at an abstract level and applications at a concrete level, e.g. Hypertext Transfer Protocol (HTTP). While input and output symbols are only manipulated at the abstract level, specially formatted requests and responses are needed by the application.

Therefore, the first step for the tester is to define these symbols. Depending on the inference model, symbols may be associated with parameters. An abstraction of an application is a set of parameterized input and output symbols. To build this abstraction, the tester has to manually browse the application and keep the appropriate input and output sets. Depending of the size of the application, the languages and the techniques involved in the application this task can be time-consuming or unrealizable.

When the abstraction is clearly defined, we still need to communicate with the application. This is where a test driver is needed. This driver, located between the inference component and the real system, will act as a proxy to convert abstract requests into concrete ones, and concrete responses into abstract ones as it is depicted in Figure 1. These two transformations called abstraction and concretization also have to be written manually. Again, depending on the number of symbols and their parameters, this task can be time consuming. Having a way to generate this test driver automatically will help the testers to use model inference then model-based testing and checking methods.

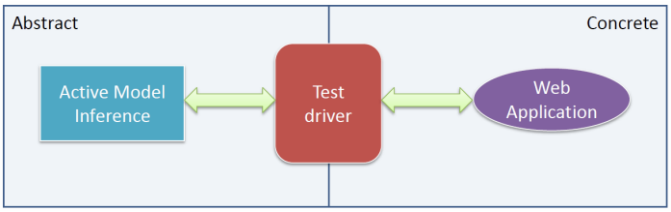

Figure 1 - The test driver between abstract and concrete levels

In section II, we define a generic web application abstraction. Next we propose a method to automatically generate the abtraction corresponding to a web application using crawling techniques. Then, in section IV, we experiment our test driver generation method on the WebGoat [4] application and finally in section V, future works are presented.

II. WEB APPLICATION ABSTRACTION

Modern web applications are developed in two parts. The back-end can be developed in multiple languages dedicated for web applications such as PHP and ASP or not dedicated such as Python, CGI and Java. The front-end is available from a web-browser through a common language: HTML/CSS. Regardless the back-end, analyzing the source code of the pages is sufficient to identify and extract the inputs and outputs needed.

An input is an action provided by the application and executed through a link or a form element in HTML. We define an input *I* as a tuple *(M, A, P)* where *M* is the HTTP method used, *A* is the URL address of a web page to handle the action and *P* is a finite set of couple (name, values) where name is the name of the parameter and values is a set of strings corresponding to the possible values found. In the case of a link, *P* can be the empty set and *M* is always the *GET* method. Parameters can have multiple values with select or radio type input tags for form, e.g. in Figure 2, and even for a link if multiple parameter definition exists, e.g. in Figure 3.

```
<select>
  <option value="1">A</option>
  <option value="2">B</option>
  <option value="3">C</option>
  <option value="4">D</option>
</select>
```

Figure 2 - Multiple parameter values in a form element

```
http://test/index.php?id=1&id=2&id=3&id=4
```

Figure 3 - Multiple parameter values in a link element

From an abstract input *I*, we are able to build the corresponding concrete HTTP request.

An output is a unique page with an abstraction on the content. To extract the different outputs of the application, we have to consider only the relevant information on the page. Page clustering is based on the structure of the page. We assume that the application has a finite set of distinct pages and each of these pages can be customized with parameters. From each page, we build a page tree. We define a page tree as a tree where a node represents a tag. The tags which are not related to the structure, e.g. *SPAN*, *CENTER*, are not considered. The same page can display information as a list. In this case, tags which are related to the structure, e.g. *LI*, *TR*, *TD*, are not considered.

At the end, we have the set of all distinct pages or outputs of the application. An output symbol is associated with each item of this set.
Using this set, for each page received during the inference process, we are able to convert it into the corresponding output symbols by comparing the page to the page in the set.

Like inputs, outputs may contain parameters. To detect the parameters, each output page has to be requested several times with different input parameter values. These values are randomly selected among the available values provided by the page. Following the content of the pages obtained, outputs parameters are detected by analyzing the differences. In Figure 4 and Figure 5, we can see the content of the same page obtained with different input parameters. We detect two output parameters in the *span* tags. We define an output parameter as the content of a *HTML* tag in the page.

```
<div id="#profile">
   <span id="#username">Alice</span>
   <span id="#money">200</span>
</select>
```

Figure 4 - A page obtained for the first time

```
<div id="#profile">
   <span id="#username">Bob</span>
   <span id="#money">100</span>
</select>
```

Figure 5 - The same page obtained a second time

These output parameters are directly dependent on the input parameters. For example, in a gallery of pictures, the content of the *img* tag is directly dependent on the id of the picture that the user wanted to display.
But then content can change even if the input parameters are the same. Typically these non-deterministic parameters are ads, date time and dynamic content like RSS flux, Chat. They are not considered as output parameters.

We have defined a method to extract all inputs in a page, differentiate two pages to find the outputs and extract the corresponding parameters. Combining these methods with a crawler allows us to extract all inputs and outputs of an application and then generate a test driver automatically.

III. CRAWLING

The goal of the crawling process is to browse as many different pages as possible in order to find the maximum of different outputs and inputs and generate a test driver automatically. Some critical data still need to be provided by the tester, e.g., credentials, but this is not comparable to the amount of data needed to provide without our method.

A classical crawling technique only enumerates all reachable links and forms found in the pages and processes them. It starts from a given page and stops when all pages are explored using depth-first-search or breath-first-search strategies.

These methods do not work anymore with web applications (as opposed to web sites) because the order of the pages visited is important. Consider a simple web application with *index.php*, *view.php*, *search.php*, *login.php* and *logout.php*. The classical crawler will start from *index.php* page which is the page given by the tester, then the only page accessible without being authenticated will be *login.php*. After being successfully authenticated, the crawler can access *view.php* which contains a link to *logout.php*. Since the crawler visits *logout.php*, no other pages like *search.php* will be accessible because *logout.php* changes the internal state of the application to another state in which we are not allowed to access to the website anymore. This is why we need a state-aware crawler.

To avoid this problem, we will not use simple requests to retrieve a page but a sequence of requests combined with depth-first-search and page clustering, as in the abstraction, to avoid loops and visit the same page again.

A given page will be requested several times but it will help the output parameter detection. On the previous website example, the sequences sent with the new approach and the logout page visited are in Figure 6.
By sending sequences of inputs, we are able to explore the whole application even if the internal state changes during the

```
1. index
2. index ; login
3. index ; login ; view
4. index ; login ; view ; logout
5. index ; login ; search
```

Figure 6 - Sequence of inputs

exploration.
The crawler stops when it obtains the same output and all inputs.

With the crawling strategy defined, we still need some heuristics to crawl efficiently recent applications and CMS. In the latter, a parameter can be used to select the content of the page. Typically, the index page has one parameter *id* to select the content or in gallery style application, each element of the gallery has one *id* associated. Each of these links differs only by one parameter value and leads to the same output. To avoid visiting this entire set of links, we use this heuristic: if at least two pages are the same, i.e. same output, and their parameters differs only by one value, other similar inputs will not be visited.

A form element can contain multiple actions, i.e. multiple submit type input tags. For example, the main page of a WebGoat lesson has a list of employees and four buttons, each associated with a different action. During the crawling, each action is extracted and all parameters are associated with it. Therefore, parameters may be unnecessary for some actions. The only way to remove them is to find the same action without the parameters. In this case, we consider that this action does not need these parameters and we remove them.

As we said previously, some critical parameters may have to be provided by the tester. To reduce their number, the crawler can use a random string to complete the input where no parameter values are specified by the tester and the website. This random value takes into account the type of the input and its attributes, e.g. size, maxlen, min, max. On the other case, when multiple parameters are available and if nothing is specified by the tester, the crawler selects one parameter value randomly.

IV. TEST DRIVER GENERATION

After the crawling, the abstraction, i.e. inputs and output symbols, their parameters and the set of different outputs are exported in an XML-based format to be used by third party inference tools. We have also developed a generic test driver written in Java which takes this XML file as input and provides through its interface the inputs, outputs, their parameters and functions to convert correctly the abstract inputs into concrete HTTP requests and the concretes HTTP responses into abstract output symbols. The XML file can also be edited manually by the tester to add or remove parameters.

V. EXPERIMENTS

The crawler is implemented in Java and use HtmlUnit, a GUI-less browser, to retrieve the pages and send the requests and JSOUP to parse the page. We have applied our method on WebGoat. This web application, written in Java/JSP, is a deliberately unsecure platform demonstrating all kind of web vulnerabilities and attacks. We choose this application because testing its security can be done using model-based testing and as no formal model is available, we can use model inference to generate one.

WebGoat is organized in lessons; the goal of a lesson is to show a type of vulnerability and its corresponding attack. In this experimentation, we choose the stored cross-site scripting (XSS) lesson, which demonstrate the vulnerability which is in the Top10 of OWASP security risk 2010 [5]. In this lesson, we have a human resource management application to manage the employee profiles with basic functionalities like view, search, update, and login/logout.

Basic information needed by the crawler and given through the configuration file consists of:

- IP:PORT and credentials
- We limit the crawler to the part of the pages corresponding to the lesson by setting the *limitSelector* parameter to *#lessonwrapper* which is the ID of the tag containing the lesson.
- External links are filtered using regular expressions.
- Values for the parameters; we only need to provide a valid employee ID and its password.

Our crawler has detected 11 inputs and 6 outputs, which are all the inputs and outputs for this lesson.
For two parameters, one in the *Search* action and the other in the *Update* action, no specified value was found. Thus, a random string has been generated. This is not the case here but the crawler warns the user that if he provides a value, this value may be more efficient and can lead to another page. By doing this, the crawler helps the user to provide relevant values and to improve the crawling.

In Figure 5, we have the actions followed by the crawler from each output. This represents the possible actions from each output page. It can be useful to detect unintended actions. The inferred model represents all possible actions for the user whereas this model represents all available actions on the website. By comparing these two models together, we are able to detect unintended action, as in [6]. *CreateProfile* leads to a dead-end node because this action is not implemented in this lesson and the request returns a blank page.

We also ran the crawler on WackoPicko [7], a realistic and intentionally vulnerable application. We obtained 21 inputs and 19 outputs.

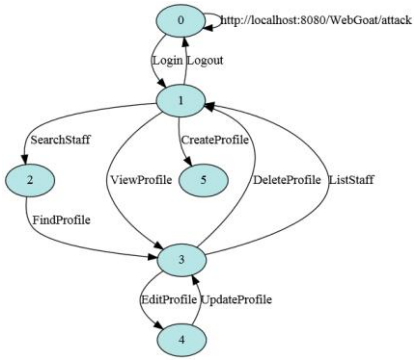

Figure 5 – WebGoat XSS Model representing the actions from the outputs

VI. CONCLUSION & FUTURE WORKS

We have defined a generic abstraction of web applications. Then, using crawling we are able to extract it and export it. We have developed a generic test driver to use this abstraction. Our method has been applied on one of the WebGoat lessons and all inputs and outputs have been found.

The abstraction extraction method presented considers all inputs and parameters in the pages. This can be improved using abstraction refinement [8] or parameters selection to keep only the most relevant parameters and reduce the time needed to infer the model. Crawling Rich Internet Applications (RIA) [9] can lead to more inputs, more outputs and a more precise model. The main improvement would be to merge crawling and inference. Inference approaches have to send a large number of requests to the application and store the observations in a table. Since crawling comes first, crawling request could be stored to be reused in the inference process. Alternatively, inference can be done while crawling [10].

VII. REFERENCE

1. D. Angluin, “Learning Regular Sets from Queries and Counterexamples”, Information and Computation, 75, 1987.
2. M. Shahbaz, R. Groz, “Inferring Mealy Machines”, Formal Methods, pp. 207-222, Eindhoven, the Netherlands, 2009.
3. Berg, Jonsson, Raffelt, “Regular Inference for State Machines Using Domains with Equality Tests”, in FASE 2008, LNCS 4961, 2008.
4. WebGoat - https://code.google.com/p/webgoat/
5. OWASP Top10 Application Security Risk 2010 - https://www.owasp.org/index.php/Top_10_2010-Main
6. Li X. and Xue Y. Block: A Black-box Approach for Detection of State Violation Attacks towards Web Applications. In Proceedings of the Annual Computer Security Applications Conference (ACSAC 2011)
7. WackoPicko - https://github.com/adamdoupe/WackoPicko
8. Aarts, Heidarian, Kuppens, Olsen and Vaandrager, Automata Learning Through Counterexample Guided Abstraction Refinement, Proceedings FM 2012, 18th International Symposium on Formal Methods.
9. Choudhary, S., Dincturk, M.E., Mirtaheri, S.M., Moosavi, A., Bochmann, G.v., Jourdan, G.-V. and Onut, I.V., Crawling Rich Internet Applications: The State of the Art, in Proceedings of the CASCON 2012, November 2012.
10. A. Doupé, L. Cavedon, C. Kruegel, G. Vigna, Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner, Proceedings of the USENIX Security Symposium 2012.