Some Modeling Challenges when Testing Rich Internet Applications for Security

Kamara Benjamin, Gregor v. Bochmann, Guy-Vincent Jourdan
School of Information Technology and Engineering
University of Ottawa
Ottawa, Canada
{kbenj067, bochmann, gvj}@site.uottawa.ca

Iosif-Viorel Onut
Center for Advanced Studies,
IBM
Ottawa, Canada
vioonut@ca.ibm.com

***Abstract*—Web-based applications are becoming more ubiquitous day by day, and among these applications, a new trend is emerging: rich Internet applications (RIAs), using technologies such as Ajax, Flex, or Silverlight, break away from the traditional approach of Web applications having server-side computation and synchronous communications between the web client and servers. RIAs introduce new challenges, new security vulnerabilities, and their behavior makes it difficult or impossible to test with current web-application security scanners. A new model is required to enable automated scanning of RIAs for security. In this paper, we evaluate the shortcomings of current approaches, we elaborate a framework that would permit automated scanning of RIAs, and we provide some directions to address the open problems.**

Keywords: rich Internet applications, formal models, software security

I. INTRODUCTION

Over the last two decades, with the spread and development of the Internet, the way software applications are engineered and delivered has evolved significantly. The threats posed by attacks carried through software have evolved alongside. The motives for such attacks have morphed from prankish activities or technical curiosity to multinational criminal schemes and politically motivated cyber-warfare. The consequences of these attacks for end users and business have also shifted from mostly annoyance or embarrassment to possible loss of data, business, money, and identity theft. In parallel, awareness of and research into the problem have increased dramatically, and solutions have been developed along several axes: better software engineering techniques, better education for programmers, development of an industry dedicated to software security, and automated tools to help secure software.

Among the technical evolutions over the past 15 years, one is particularly noteworthy: the use of "the web" to present application interfaces to the user. By turning to such "web applications", software can be engineered using a widely available standard to render the user interface (HTML, rendered in a web browser), and using a widely available standard protocol (HTTP) to exchange messages between the client and the server. Doing this has several obvious benefits: 1) it makes it much easier to engineer applications with a client side that will run on a large variety of systems, since almost any system provides one or several software tools that support the necessary standards; 2) it makes it very easy to distribute and update applications; 3) the configuration required per user is minimal; and 4) the systematic usage of standard TCP ports (80 and 443) is an easy way around firewalls.

It has several disadvantages as well, many of which have security implications: 1) above all, the protocol itself, HTTP, was not designed to serve applications, and various poor replacements for required features had to be added (such as cookies to maintain sessions); 2) the already mentioned approach to deliver everything over the same TCP port, thus avoiding port-based firewalls, is obviously a setback from the security viewpoint, since it defeats the purpose of having firewalls in the first place; 3) massive deployment of web applications create a network of related applications sharing machines, data, and databases, in such a way that the vulnerability of one of the applications often implies the demise of all co-hosted applications; and last but not least, 4) the relative ease with which web applications can be developed means that scores of poorly engineered web applications have been deployed all over the Internet.

Since the security of web applications is of paramount importance, web application scanners have been developed as effective tools for evaluating the security of web-based applications. These tools have been available for some time from large corporate players as well as many other players and even several open-source solution providers. These tools, which include the IBM® Rational® AppScan® family of products, and WebInspect® from HP®, all follow the same basic scheme: given a URL pointing at a web application, and some minimal amount of information about the application (such as a login ID and password), the tools "crawl" the application, that is, try to uncover all the possible pages (or client states) that can be reached, much as a search engine will index any web site. Once the pages have been uncovered, a series of automated security tests are performed on each page and a report assessing the security of the scanned application is created. The Rational AppScan family of products will typically generate several thousand different security tests on each page of an application. If the tool is effective, it is a valuable companion for security teams, since the large number of tests done automatically would require a very long time to do manually. These tests can be re-run on a regular basis to ensure that new versions or upgrades to the system are not introducing new security vulnerabilities.

The effectiveness of these tools is linked to their ability to effectively crawl the application and to the range of the

security test being applied; it is also affected by how up-to-date the tests are.

In the more recent past, a new type of web application has gained popularity: the so-called rich Internet applications (RIAs[1]) break away with the idea that a "screen", or a client state, is essentially a web page that is defined by its URL. Prior to RIAs, very little actual computation was done on the client side. RIAs have several benefits: 1) significant client-side computation can be performed, 2) the state of the application at the client side can be modified without going back to the server, and 3) technologies such as Ajax make it possible to send asynchronous requests to the server and handle the response in the background, without reloading the client page. Instead, the data which defines the client-side state, as represented by the current Document Object Model (DOM), is modified directly by client-side code and produces the next client state.

This seemingly purely technical difference has in fact a very deep impact on the way we can automatically test web applications: it is not true anymore that the current URL reflects the current state of the client; in order to crawl an application, testing tools must do more than simply recursively follow all the embedded links in each page; and the current DOM is not anymore built from scratch from the last received HTTP response. Many of the most popular websites, such as Google and YouTube, are RIAs, and some very advanced RIAs are entirely based on asynchronous server communications after the first page, meaning that the entire application has a single URL!

By moving to RIAs, we have lost the ability to crawl our applications. The consequences are severe: 1) search engines cannot index sites that use such applications, 2) automated testing tools fail to construct the initial model of the application, 3) automated tools to assess usability do not work either, and of course, 4) web application security scanners face the same issues. A new approach is required for crawling RIAs in order to build automatically a model that can then be used for various purposes such as security evaluation, usability tests, and indexing.

Our team is working in collaboration with IBM on creating a new generation of web scanners that are able to handle RIAs. As explained above, the first challenge is thus to crawl RIAs. Although our own end-goal is to assess security, the crawling activity (and thus the impact of this research) goes well beyond security, and touches on anything that involves automatic processing of web applications.

In this paper, we attempt to state the problem clearly, review the current state of the art, and propose early directions for solutions. We begin in Section II with an analysis of the needs for the model and the model construction algorithms. We then review the state of the art in Section III, drawing from research in different fields (including testing and indexing) but trying to address the same problem of crawling RIAs. In Section IV, we discuss the shortcomings of the existing solutions for our purpose. In Section V, we propose a new theoretical solution to the problem, which is more in line with our stated goals, as compared with other existing approaches. Still, the question is far from resolved, and we review some of the remaining difficulties in Section VI.

[1] Note that for simplicity we include in our definition of "RIAs" applications that have client-side code modifying the DOM even if they do not use technologies such as Ajax.

II. REQUIREMENTS

A. Assumptions about the application being crawled

Some assumptions about the application being modeled are necessary. The basic one is that we are looking at Rich Internet Application, that may use client-side JavaScript to update the local DOM and/or to connect back, synchronously or not, to the server. A series to limiting assumptions are also made in order to ease the modeling, or indeed to make it possible at all. Among others, we typically assume that:

- Actions are repeatable, that is, it is possible to "reset" the application in such a way that sending the same input in the same order and at the same time will produce the same output.
- The only source of non-determinism is concurrency. What we want to avoid is an application that will react differently, starting from the same global state, when the same input is given at two different times.
- Applications are "user-input"-free: every interaction between the application and the user can be modeled as a choice among a known finite set of possibilities. This fits well with inputs such as check boxes and drop-down menus. For "free text" input, it means that the crawler is made aware of all the possible "categories" of inputs.

B. Requirements of the Model

Given that an RIA does not consist of static HTML pages, but nevertheless contains a collection of different states, with content being updated without the need to reload the entire page (and thus without the need to change the URL), our model needs to be able to capture, under limiting assumptions such as the ones listed in Section II A., all the states of the application so that all the content will be available for analysis or indexing. We also need to be able to capture the events that lead to the movement from one state to another, as well as any additional data (such as variable values and user input) that may contribute to determining the next state.

Furthermore, the model that is produced should be produced in a deterministic way – we should expect to obtain the same model when crawling the unchanged website several times. We also need to be able to ensure that the model that is produced of a given website is complete, that is, all states are successfully explored and recorded.

C. State Equivalency

We would like to separate the strategy of construction of the model from the process of determining equivalence between different states. The latter task is very important and will have a great impact on the correctness of the model. It

seems very difficult to come up with a single solution for state equivalence. Clearly, deciding whether a given client state is "similar" to another client state is application-dependent. In addition, even within a given application, the notion of equivalence may vary depending on the purpose of the model being built. As a simple example, consider two states that have the very same content but in a different order. From a security viewpoint, and perhaps also from an indexing viewpoint, the two states should likely be marked as equivalent. However, from a usability viewpoint, they are not. So if the model being built is meant to assess the usability of the application, then crawling should continue, but if it is meant to assess security, it should stop (for the state in question). This situation might be very common in e-commerce applications, with large catalogs being published. In some cases, the entire catalog should be crawled, while in other cases, a single entry suffices.

Being able to adapt the equivalence relation to the application and the purpose of the model is thus crucial. The choice of an appropriate equivalence relation should be considered very carefully. If an equivalence evaluation method is too stringent, then it may result in too many states being produced in the model, leading to state explosions, long runs, and in some cases infinite runs. (For example, an application printing out the current time somewhere in the page may be crawled forever if the equivalence relation does not filter out the timestamp.) On the contrary, if the equivalence relation is too lax, we may end up with client states that are merged together while, in reality, they are different, leading to an incomplete, simplified model.

D. Efficiency

It is important that our model construction strategy has the ability to produce as much of the model of the website as quickly as possible. Indeed, crawling an application might be a very long (or even theoretically infinite) process; therefore, waiting for the end of the crawl might be impractical. For the case in which the crawl is stopped before completion, we should have a strategy that will maximize the amount of information found initially. In other words, we need an algorithm that executes events in such an order that we have a high probability of finding new states. We also need to avoid infinite runs; that is, we need to guarantee that the probability of uncovering any given reachable state is one, given enough time. It is also necessary that the crawling algorithm, when applied to the same application several times in the same context, will produce the same model each time. In order to do this, the algorithm must crawl websites in a deterministic fashion.

Clearly, the approach taken for state equivalence will also affect the performance of the algorithm. It would be beneficial to have an efficient algorithm for determining the equivalence of states.

III. STATE OF THE ART

Several papers have been published on the general question of crawling RIAs, most (if not all of them) focusing on Ajax-based applications.

In [1], Matter uses the approach of representing an Ajax application as a finite state machine. As in other similar papers, nodes/states are represented by the current DOM tree, and transitions are caused by events that lead to changes in the DOM. In this paper, the model of the application is grouped by page. That is, every state that shares a given URL is considered to be part of the same grouping. In this algorithm, states are compared based on the DOM's hashed value. Those with the same hashed value are considered to be the same (i.e., equivalent). Another idea of this paper is to increase the performance of Ajax crawling through the use of caching. In order to do this, the JavaScript™ code leading to an Ajax call, as well as any parameters used, are cached. The DOM produced as a result of this Ajax call is also cached. Then, whenever a JavaScript event is about to be executed, a check happens first to see whether the same event (with the same parameters) has already been cached. If it has, the cached DOM is used instead of making a new Ajax call. The problem with this method is that it relies heavily on the assumption of a stateless server.

In [2], Mesbah, Bozdag, and van Deursen again rely on the construction of a transition graph with the same definitions of state and transition common to the other papers. In order to determine whether or not one state is equal to another, they calculate the edit distance, or number of operations that would be needed to transform one DOM to the other (the so-called Levenshtein distance), to determine the distance/difference between DOMs. If this distance falls within a similarity threshold defined by the developer, then the two states are considered equivalent. However, note that this notion is not transitive, so relying directly on it would be problematic in our context since states that are pairwise below the set threshold for the distance cannot necessarily be grouped together.

In [3], Manku, Jain, and Sarma focus on the topic of state equivalence. They use another technique, simhash, which uses a hashed value to determine whether or not two documents/states are equivalent. To do this, they first divide a document into a set of "features", each having a specific weight. This data is used with simhash to produce a fingerprint for the document, which is then compared with those of other documents to determine equivalence with documents producing similar hash values. Again, we do not believe that this method would work effectively for our purpose since it is not an equivalence relation.

In [4], Marchetto, Tonella, and Ricca construct a transition graph of an Ajax application using both dynamic analysis and static analysis of code. The model constructed, while also using the transition graph approach, does not use DOM states. Instead, state abstraction is used. This means that, for example, in a site such as an e-commerce website, a state could be determined by values such as the current number of items in the shopping basket and the current total cost of those items. Transitions would then occur when items are added or removed from the shopping basket causing a change to those values. Again, this requires some manual activity but it is also an illustration of the transition graph

being the dominant representation of an Ajax application across the various papers.

In [5], Shah discusses a method of crawling Ajax-applications but does not focus on the modeling of those applications. Therefore, while the paper does not provide new ideas about modeling Ajax-applications, it discusses some techniques for crawling, including the use of tools such as Watir to automate the browser process and Narcissus to analyze JavaScript in order to identify which events would lead to Ajax (XML HTTP Request) calls. The identification of events leading to Ajax calls, and the ability to control the execution of their associated callbacks, will be critical for building our model.

IV. Current Problems and Limitations

The ideas reviewed in the state of the art section fail to meet our requirements in a number of ways, as explained in the following subsections.

A. Incomplete Models

The approaches that we have seen do not construct a complete model and skip some of the client states when crawling the application.

One of the obvious shortcomings of some of these models (for our needs) is neglecting to explore different combination of events that are enabled concurrently. When several events can be executed from a given state, it is possible that executing them in different orders will lead to different results (i.e., different states). Trying all the combinations is obviously very time-consuming, but running a single one of the possible sequences is not an acceptable trade-off.

A more subtle problem, which is neither addressed nor discussed in any existing model to our knowledge, comes from the observation that asynchronous requests are processed in two steps: in the first step, the request is sent, and in the second step, the response (the "callback") is processed. First, the order in which the several callback methods are executed could affect the resulting state. In addition, the state of the DOM *before* a callback method is executed should also be accounted for. In other words, executing an event that generates an asynchronous request back to the server identifies not two but three states: 1) the state before the event is executed, 2) the state reached once the event is executed and before the callback is executed, and 3) the state reached once the callback is executed. An event generating more than one asynchronous request will generate even more possible states.

Missing states means that some content would not be indexed, for example, or security vulnerabilities could be missed. It is therefore critical that the crawling method, given enough time, will not miss any state.

B. Strategy

When building the model, we crawl the application, trying to uncover all the states. This may take too long, so we cannot assume that we will carry the operation to the end. (If we do carry the operation to the end, then the strategy followed does not matter.) If the crawl is stopped before all states are explored, we still want to have uncovered the largest possible set of states. Therefore, the crawling strategy should first concentrate on finding new states in the model and keep the "cleaning up" for a later phase of the exploration (that is, try to uncover new states before trying to confirm that various combinations of actions indeed lead to the same state). The research that we have seen thus far does not attempt to use any strategy for uncovering as many new states as possible in the shortest amount of time.

C. State Equivalence

The papers that we have reviewed appear to focus on efficiency when it comes to deciding whether the current state is similar to an already crawled one. For our purpose, we need correctness first; that is, we need an equivalence relation (from the mathematical viewpoint). Additionally, many websites have pages that contain some content that should be ignored (such as advertisements or timestamps) when determining equivalence. Finally, as already mentioned, the notion of "similar page" is dependent on the application being crawled and the end purpose of the model being built. Again, none of the models reviewed accommodates this flexibility,

V. A Strategy For model Construction

As explained in previous sections, we need to be able to crawl an application in such a way that, if the process is not carried through to the end, the partial information uncovered is as rich as possible. This implies that the crawling algorithm should follow some strategy to ensure this property. It also implies that two assumptions must be made about what model information is most important.

The first such assumption we can make concerns the trade-off between fully exploring a state versus starting the exploration of another state: once we have uncovered a client's state *s,* which has a set of enabled events, in order to see all the possible other states that can be reached by executing these events, we have to execute all the events in any possible order. Doing so immediately may take a long time while showing many different paths leading to the same states. Instead, we want to uncover and explore new states first, and come back to this state *s* and explore it in more detail once we have no more new states to explore. Therefore, initially, we execute the possible events using only one of the possible orders, based on the assumption that executing all the events in a different order will not likely uncover new states. When we come back to that state later (because we ran out of unexplored states), we will explore the states reached when the events are executed in the different orders. Our second assumption is that new states are more likely to be found when a different subset of events is executed than when the same subset is executed but in a different order. So as we enumerate all the possible event orderings (*n!* different orderings for *n* events), we need to do it in such a way that we will first try different subsets of events (2^n possibilities).

To summarize, the assumptions are:

1. If we have two unexplored states *s1* and *s2*, after having executed all the events in *s1* (and possibly uncovered new states with new events in the process), executing events in *s2* is more likely to uncover new states than is executing the events of *s1* again but in a different order.
2. If we have a state *s1* in which we have already executed all the events in a few different orders, executing a subset of the events that has not been executed yet is more likely to yield a new state than is executing a subset of the events that was already executed but in a different order. This is so because the latter is more likely to lead to a state equivalent to one already visited.

Achieving a crawl that is compatible with the first assumption is rather simple. We will explore new states first: as long as we have states that have not been explored at all, these states should be explored by executing all events once, before further exploring any other state. Note that when exploring new states first, an in-depth strategy should be followed as much as possible, because it is always more efficient to follow the logical flow of the application. However, following a pure in-depth strategy might lead to an infinite path in which only a subset of the useful states are explored. As previously stated, the goal is to have a model such that any state will be discovered with probability 1 (in a finite amount of time, if possible). To achieve this goal, the in-depth strategy must be mitigated by a maximum depth, after which the algorithm should branch back to an earlier state, so as to avoid going down infinite branches forever.

Achieving a crawl that is compatible with the second assumption is more challenging. For this, we need to provide a strategy that will eventually enumerate all the possible sequences for executing the events that are enabled from a given state, in such a way that we will, as much as possible, execute a subset of events that was never executed before. To simplify the discussion, we are going to assume that we are exploring a state that has n enabled events, called e_1, e_2,…, e_n, and that these n events are independent, that is, executing a given subset of events in any order leads to the same state. In this case, there are 2^n possible subsets of events, which, when ordered by inclusion, define a hypercube of size n, with $n!$ different paths from the bottom to the top. An example is shown Figure 1 with a hypercube of dimension four. There are 4!=24 different paths in this hypercube, with 2^4=16 different states. We thus need a way to enumerate the 24 paths such that the 16 states will be visited as early as possible in the process.

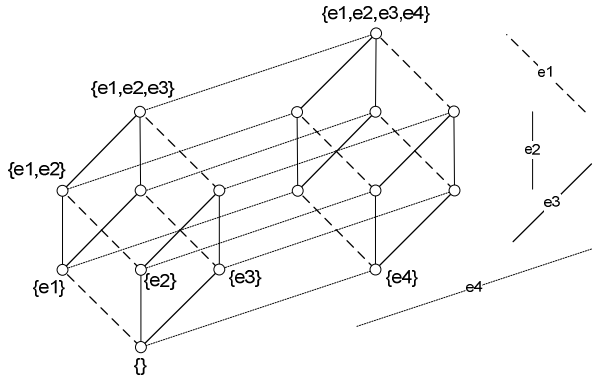

Figure 1: A hypercube of dimension 4, corresponding to the execution of events e_1, e_2, e_3, and e_4. Each edge corresponds to the execution of a given event (as per the legend on the right) and each vertex corresponds to the state defined by the set of executed events. (Only some of the states are labelled with the corresponding set of events for readability.)

Such a hypercube is a partially ordered set (a lattice in this case), and each path of the hypercube is actually a chain of the order, that is, a set of pairwise comparable elements. Finding a set of chains that cover all the elements of the order is known as a chain decomposition of the order. Since we want to cover the hypercube using as few chains as possible, we are trying to do what is known as a *minimal chain decomposition* of the order. (See [6], for example, for an overview of these concepts.) In 1950, Dilworth proved that the minimal number of chains necessary to decompose an order is equal to the width of this order, that is, the maximum number of pairwise non-comparable elements [7]. In the case of the hypercube of size n, it is well known that the width is equal to $\binom{n}{\lfloor n/2 \rfloor}$. For example, for the hypercube of size 4, the width is $\binom{4}{2}=6$. It is thus possible to achieve our aim with 6 paths, and this is the best that can be done. An example of a sequence of all 24 possible paths that reaches all 16 states within the first 6 paths is as follows:

1. e1;e2;e3;e4	2. e2;e3;e4;e1	3. e3;e4;e1;e2
4. e4;e1;e2;e3	5. e1;e3;e4;e2	6. e2;e4;e1;e3
=> all 16 states have been visited at this point		
7. e3;e1;e2;e4	8. e4;e2;e3;e1	9. e1;e4;e2;e3
10. e2;e1;e3;e4	11. e3;e2;e4;e1	12. e4;e3;e1;e2
13. e1;e2;e4;e3	14. e2;e3;e1;e4	15. e3;e4;e2;e1
16. e4;e1;e3;e2	17. e1;e3;e2;e4	18. e2;e4;e3;e1
19. e3;e1;e4;e2	20. e4;e2;e1;e3	21. e1;e4;e3;e2
22. e2;e1;e4;e3	23. e3;e2;e1;e4	24. e4;e3;e2;e1

In 1952, de Bruijn, Tengbergen, and Kruyswijk provided an algorithm for decomposing certain orders, including a hypercube [8]. In [9], Hsu, Logan, Shahriari, and Towse expose the methods as follows (adapted to our hypercube definition):

Definition (adapted from [9]): The *canonical symmetric chain decomposition*, or CSCD, of a hypercube of dimension n is given by the following recursive definition:

1. The CSCD of a hypercube of size 0 contains the single chain (Ø).
2. For $n \geq 1$, the CSCD of a hypercube of dimension n contains precisely the following chains:
    1) For every chain $A_0 < \ldots < A_k$ in the CSCD of a hypercube of dimension $n - 1$ with $k > 0$, the CSCD of a hypercube of dimension n contains the chains:

$$A_0 < \ldots < A_k < A_k \cup \{n\}$$
and
$$A_0 \cup \{n\} < \ldots < A_{k-1} \cup \{n\}.$$

    2) For every chain A_0 of size 1 in the CSCD of a hypercube of dimension $n - 1$, the CSCD of a hypercube of dimension n contains the chain:

$$A_0 < A_0 \cup \{n\}$$

Applying this method to the hypercube of dimension 4 shown in Figure 1 leads to the following 6 minimal chains decomposition:

1. $\{\} < \{e_1\} < \{e_1, e_2\} < \{e_1, e_2, e_3\} < \{e_1, e_2, e_3, e_4\}$
2. $\{e_4\} < \{e_1, e_4\} < \{e_1, e_2, e_4\}$
3. $\{e_3\} < \{e_1, e_3\} < \{e_1, e_3, e_4\}$
4. $\{e_3, e_4\}$
5. $\{e_2\} < \{e_2, e_3\} < \{e_2, e_3, e_4\}$
6. $\{e_2, e_4\}$

Note that this method is polynomial in the size of the end result (which is itself clearly exponential in the size of the input order, but this is intrinsic to the problem).

Following such a CSDC does not provide us directly the result we are aiming for, but is the crucial step towards it. The first problem is that we do not have complete paths but only simple chains inside the hypercube. These chains need to be extended into paths going from the bottom to the top. The second problem is that we are not getting all the paths but only a small subset of the possible paths.

Addressing both problems does not pose great difficulties. Here we simply provide a sketch of a possible solution: in order to generate all the possible paths, it is enough to follow any kind of depth-first strategy on the graph induced by the hypercube. So the proposed strategy becomes as follows: the first time a state is reached, with n enabled events, build a CSDC for the hypercube of size n. Extend each of the produced chains to n elements by removing one after the other all the events from the smallest set of the chain, and by adding one after the other all the missing events from the largest set of the chain. This provides the first $\binom{n}{\lfloor n/2 \rfloor}$ paths, covering all the possible sets of events. (For an added bonus, one might want to order this chain by decreasing number of elements in the CSDC.) This first set of paths will be followed when the state is explored the first $\binom{n}{\lfloor n/2 \rfloor}$ times. After this, the other paths should be explored by doing a depth-first traversal of the graph induced by the hypercube, starting from the paths that have already been visited. These choices (and any choice in the previous steps) must be made in a deterministic way, so that executing the crawling algorithm a second time (assuming that the context is the same) will lead to the same model.

Note that the algorithm that was sketched above will, in reality, be mixed with the more general strategy exposed earlier: that is, each path of the decomposition, starting from those deduced from the CSDC until the last one of the traversal, will be executed at different times during the crawl, not one after the other.

The strategy proposed here meets the requirements outlined in Section IV: we explore new states first, and we explore new subsets of events first. Consequently, if the crawl has to be stopped prematurely (before it runs out of states to explore), the portion of the model that will have been created will have uncovered a large number of states (according to our assumptions). Moreover, this is built efficiently in that it is done polynominally in the size of the resulting model.

VI. ANTICIPATED PROBLEMS

We anticipate facing several difficulties, both practical and theoretical, as we progress towards a complete solution. We list here some of them.

A. Data Input Values

Our current view of the model is mostly limited to states and events. However, input values are also very important, and different input values might lead to different states when executing the same event from the same state. Since, in many cases, the number of possible input values is theoretically infinite, we will have to sample and choose a few representatives. Several questions would have to be answered, including how to generate the values and how many samples to draw.

B. Statelessness of Server

The problem of server-side states is perhaps the most challenging of all the problems anticipated so far. Applications using server states may behave differently when the very same set of actions is done twice at the client-side. This may prevent any kind of efficient crawling (by enforcing a "normal flow" and thus forcing the crawling to restart from the initial state after each branch of the exploration). And it may also lead to models that represent the last crawl only. One idea in this respect is the possibility to make a distinction between events that do generate requests to the server (and thus may change the server state) and events that do not. Events that do not go back to the server can be crawled and reset entirely at the client-side.

C. State Equivalence

Finding efficient, workable strategies to decide whether two different states are equivalents with respect to the model will be particularly difficult. As previously explained, failing to recognize some states as equivalents might create a model that is needlessly large, and potentially (theoretically) infinite. On the other hand, wrongly concluding that two different states are equivalent will collapse the model and produce an inaccurate result.

Beyond the mere definition of equivalence, we also anticipate encoding challenges, to be able to find out the equivalence class of a given DOM rapidly and to be able to decide quickly whether the class has been seen before or not.

We anticipate that the equivalence relation will be context-dependent, based both on the application and the goal of the model.

D. State Explosion

In some cases, it is expected that the model will contain many thousands of states, which will have to be stored, compared, and recorded by the prototype. Note that this issue is not about long/infinite runs, but about the problem faced by a tool in building a model when handling a number of states that grows exponentially with the number of JavaScript events.

E. Model for Security

In the context of our research, the end goal is to build a model of an RIA that we will use to scan for security vulnerabilities using an automatic scanner. This is different from other possible uses of such models: what matters is not so much what states the client *should* be in as what states the client *could* be in. In other words, if an attack can be carried out by bypassing some of the logical flow of the application, then this attack should be detected by the scanner. The model that we are building at this point is a reflection of the actual states of the client, following the application flow. This model will need to be enhanced to also capture "illegal" operations on the client side.

VII. Conclusion

RIAs challenge the current techniques for automated processing of web applications. Until a new framework is created, RIAs will not be automatically indexed, automatically tested, or automatically assessed for security or usability.

In this paper, we have provided some directions in order to resolve these problems. We have shown that the current crawling methods fall short of delivering the necessary features, and we have provided a first step for a better solution. Yet much remains to be done, and this paper only provides an initial discussion on this very important problem.

Acknowledgment

This work is supported in part by IBM, and by the Natural Science and Engineering Research Council of Canada.

Disclaimer

The views expressed in this article are the sole responsibility of the authors and do not necessarily reflect those of the Center for Advanced Studies, IBM.

References

[1] R. Matter, "AJAX Crawl: making AJAX applications searchable. ETH, Eidenossische Technische Hochschule Zurich, Department of Computer Science, Distribution Computing Group (2008). Doi: 10.3929/etz-a-005665330.

[2] A. Mesbah, E. Bozdag, and A. van Deursen, "Crawling AJAX by Inferring User Inferface State Changes," Proc. 8th Int. Conf. Web Engineering, ICW08 (2008) p. 122-134.

[3] G. S. Manku, A. Jain, and A. D. Sarma, "Detecting near-duplicates for Web crawling," In Proc. 16th WWW, pages 141-150, Banff, Alberta, Canada, May 2007.

[4] A. Marchetto, P.Tonella, and F. Ricca. State-based testing of AJAX web applications," Proc. 1st IEEE Intl. Conf. on Software Testing Verification and Validation (ICST '08). IEEE Computer Society, 2008.

[5] S. Shah, "Crawling AJAX-driven Web 2.0 Applications," http://www.infosecwriters.com/text_resources/pdf/Crawling_AJAX_SShah.pdf

[6] I. Anderson, "Combinatorics of Finite Sets," Oxford Univ. Press, London, 1987.

[7] R. P. Dilworth, (1950), "A Decomposition Theorem for Partially Ordered Sets", Annals of Mathematics, vol. 51, 1951, pp. 161–166.

[8] N. G. de Bruijn, C. Tengbergen, and D. Kruyswijk, "On the set of divisors of a number," Niew Arch. Wisk 23(1951), 191-194.

[9] T. Hsu, M. Logan, S. Shahriari and C. Towse, "Partitioning the Boolean Lattice into Chains of Large Minimum Size", Journal of Combinatorial Theory, Vol. 97(1), January 2002, pp. 62-84.