A Model-Based Approach for Crawling Rich Internet Applications

MUSTAFA EMRE DINCTURK, GUY-VINCENT JOURDAN,
and GREGOR V. BOCHMANN, University of Ottawa
IOSIF VIOREL ONUT, IBM

New Web technologies, like AJAX, result in more responsive and interactive Web applications, sometimes called Rich Internet Applications (RIAs). Crawling techniques developed for traditional Web applications are not sufficient for crawling RIAs. The inability to crawl RIAs is a problem that needs to be addressed for at least making RIAs searchable and testable. We present a new methodology, called "model-based crawling", that can be used as a basis to design efficient crawling strategies for RIAs. We illustrate model-based crawling with a sample strategy, called the "hypercube strategy". The performances of our model-based crawling strategies are compared against existing standard crawling strategies, including breadth-first, depth-first, and a greedy strategy. Experimental results show that our model-based crawling approach is significantly more efficient than these standard strategies.

Categories and Subject Descriptors: H.5.3 [**Information Interfaces and Presentation**]: Hypertext/Hypermedia—Navigation; H.3.3 [**Information Storage and Retrieval**]: Information and Retrieval—Search process; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms: Design, Algorithms, Experimentation

Additional Key Words and Phrases: Crawling, rich Internet applications, AJAX, modeling, dynamic analysis, DOM

ACM Reference Format:
Mustafa Emre Dincturk, Guy-Vincent Jourdan, Gregor V. Bochmann, and Iosif Viorel Onut. 2014. A model-based approach for crawling rich Internet applications. ACM Trans. Web 8, 3, Article 19 (June 2014), 39 pages.
DOI: http://dx.doi.org/10.1145/2626371

1. INTRODUCTION

Crawling is the activity of capturing all the reachable pages of a Web application together with the information on how to reach them. A crawler is a tool that performs

G.-V. Jourdan and G. V. Bochmann are Fellows of IBM Canada CAS Research, Markham, Ontario, Canada. I. V. Onut is affiliated with IBM Canada Software Lab, Canada.
This article is a significantly improved version of Benjamin et al. [2011] which appears in *Proceedings of the 11th International Conference on Web Engineering (ICWE2011).*
This work is supported in part by IBM and the Natural Science and Engineering Research Council of Canada. The views expressed in this article are the sole responsibility of the authors and do not necessarily reflect those of IBM.
IBM and AppScan are trademarks or registered trademarks of International Business Management Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.
Authors' addresses: M. E. Dincturk (corresponding author), G.-V. Jourdan, and G. V. Bochmann, EECS, University of Ottawa, 800 King Edward Avenue, Ottawa, ON, K1N 6N5, Canada; email: mdinc075@uottawa.ca; I. V. Onut, Research and Development, IBM Security AppScan Enterprise, IBM, 770 Palladium Drive, Ottawa, ON, K2V 1C8, Canada.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2014 ACM 1559-1131/2014/06-ART19 $15.00
DOI: http://dx.doi.org/10.1145/2626371

crawling. A crawler navigates through the pages of the application starting from an initial page, just like a human navigates using a browser, but in an automated, systematic, and efficient way. The most common reason for crawling is indexing, that is, gathering all the content of a site to make it searchable through search engines. But, crawling is also needed for automated model inference that is used for activities such as automated testing, automated security assessment, or automated usability verification.

Over the years, Rich Internet Applications (RIAs) have become the new norm for Web applications. RIAs break away from the traditional concept of a Web application running exclusively on the server side. A Web application consists of a client software (a Web browser) that allows the user to send a request to a Web server that provides the content as a response. In traditional Web applications, the user navigates to another page of the application by following one of the URLs embedded in the current page. Following a URL generates a new request to the server and the server responds with a new HTML page that completely replaces the current page. Initially, applications relied completely on the server to implement the application logic and the communication between the client and the server was synchronous. These applications were not very responsive and lacked the "desktop feel" of non-Web applications.

RIAs changed this situation with two enhancements: First, RIA computations can be carried out at the client side via scripts such as JavaScript. Thus, the current page can be modified partially or completely by client-side scripts, without going back to the server. The events defined on HTML elements trigger the execution of such scripts. The second enhancement is asynchronous communication between the server and the client. This allows the user to use the application without having to wait for the reception of the responses for the previous requests. These changes were introduced gradually, but today the most common such technology is AJAX (Asynchronous JavaScript And XML [Garrett 2005]).

These improvements came at a great cost; we have lost the crawling ability. Traditional crawling techniques are not sufficient for crawling RIAs. In traditional Web applications, a page is identified by its URL and each page contains the URLs of those pages that can be reached directly from it. Hence, crawling a traditional Web application is not a difficult task: start by visiting an initial URL, collect all the embedded URLs in the corresponding page, and keep collecting new URLs by visiting the already collected URLs until all URLs are visited. But in RIAs, the current page can be dynamically changed with the execution of events, hence a URL does not correspond to a single page; many different pages can be reached under the same URL. Moreover, client-side modifications can add and remove URLs in the current page, so scanning the initial page for embedded URLs is not sufficient; the correct sequence of events must be executed first.

This unintended consequence of RIA technologies is not usually understood by users, who still expect RIAs to be indexed. This is simply not the case; to our knowledge, none of the current search engines and Web application testers has the ability to crawl RIAs[1]. Crawling RIAs is a problem that needs to be addressed in order to keep Web applications testable and indexed by search engines.

[1]For example, Google [2009] acknowledges its inability to crawl AJAX applications and suggests a method where the Web developer has to present a static HTML snapshot of each state reachable by AJAX, when asked by Google. The Web developer also has to produce URLs for AJAX states by appending a hashbang sign (#!) followed by a unique name for the state at the end of the application's URL and put them somewhere visible to the crawler, such as Sitemap. When the crawler sees such a URL it will understand that this is an AJAX state and will ask the server for the static HTML snapshot corresponding to that state. Obviously this method just makes crawling the responsibility of the Web developer and is not a real solution to RIA crawling. Also see Bau et al. [2010] for a survey of Web application security testers.

The contribution of this article is a new general methodology to design efficient strategies for inferring models of RIAs: “model-based crawling”. Model-based crawling is a generic approach for crawling RIAs; it can be used to define several actual crawling strategies. It requires the definition of a model that is used as a basis for generating the crawling strategy. Model-based crawling is the framework used to create an actual strategy from a chosen model. Based on this chosen model, the created strategy is able to make anticipations about the behavior of the application and thus discover the states in the application much faster. This is the major difference of our approach with the existing strategies that do not have any such mechanism to anticipate. In this article, we formally define the entire framework for model-based crawling. We also present in detail one such model and its specific strategy: the “hypercube” strategy[2]. We prove this strategy is optimal under the assumptions made about the behavior of the application. We chose to use the hypercube in this article because of its formalism and optimality, as a good illustration of the concepts behind model-based crawling. Note, however, that the model of the hypercube is not a very realistic model for RIAs, so the performance is not the best (but still much better than the usual breadth-first and depth-first approaches). Other models based on the principles presented in this article have been proposed, with much improved performances. We will illustrate these results as well, using real RIAs in the experimental study in addition to some test applications.

In this article, we focus on AJAX-based applications, however, the same concepts and ideas are still applicable to other similar RIA technologies (such as Flex [Apache 2004] and Silverlight [Microsoft 2007]).

The rest of the article is organized as follows: Section 2 contains a discussion of the general concepts and challenges for crawling RIAs. Section 3 presents a summary of the work related to crawling RIAs. Section 4 presents an overview of our methodology. In Section 5 we describe the hypercube strategy and mention two other crawling strategies designed using our approach. Section 6 presents the technical details, proof of optimality, and complexity analysis of the hypercube strategy. Section 7 contains the experimental results and Section 8 concludes the work with a summary of our contributions and some research directions.

2. CRAWLING RICH INTERNET APPLICATIONS

An RIA can be considered as two programs, one on client side and one on the server side, running concurrently. In an RIA, when the user requests a page from the server, the response may contain the contents of the HTML document and some code that can be executed on client side. This client-side code is usually executed as a reaction to the user’s actions that are defined as “events” on the page. The client-side code can modify the current page into a different view without contacting the server. When the page is modified on client side, the URL of the page does not change. These scripts can also send asynchronous requests to the server, that is, when such a request is made, the user interaction is not blocked by waiting for the response to come. The data retrieved from the server can be used to partially update the page rather than refreshing the whole page.

2.1. Document Object Model and Events

In a browser, an HTML page is represented as an instance of the Document Object Model (DOM) [W3C 2005]. A DOM is a tree data structure that is formed following

[2]An early version of the hypercube strategy was presented in Benjamin et al. [2011]. That version was not practical since it had a precomputation overhead that could be exponentially larger than the actual size of the model being built. In this article, we present an improved version of the strategy that is carried out on-the-fly without requiring any precomputation.

the hierarchical structure of the HTML document. DOM also defines a platform- and language-independent interface to access and modify the contents, style, and structure of the underlying document.

AJAX-based RIAs use the scripting language JavaScript to modify the page through the DOM interface in the browser. In AJAX-based applications, when the client requests a URL from the server, the response contains the contents of the HTML document and the JavaScript code that can be executed on the document. The JavaScript code execution can be triggered by events defined as part of the DOM.

Definition 2.1 (*Event*). An *event* is a user interaction (or a timeout) that triggers client-side code execution.

In Web pages, events are associated with the DOM elements. In a page, the DOM elements are allowed to react to certain predefined user interactions (a mouse click, mouse over, selecting an input field, submitting a form, etc.) and Web developers can define what JavaScript code should be run in case an event occurs on an element. The code that runs when triggered by the event is called an event handler. A crawler can analyze the DOM of a page to detect the elements having registered event handlers and execute these handlers as if a user interaction took place. In the following, we refer to running the event handlers registered for an event on a DOM element simply as an "event execution".

2.2. A Model of a Web Application

The result of crawling is called a "model" of the application. A model contains the discovered pages and the possible ways to move from one page to another. In modeling terms, a model consists of client states (or simply states) and transitions.

Definition 2.2 (*Client State*). A *client state* (or simply state) is the state of the application as it is seen on the client side. States represent DOM instances.

Definition 2.3 (*Transition*). A *transition* represents the execution of an event that leads the application from one state to another.

In Web applications, there are two ways a crawler (or a user) can trigger a transition: follow a URL or execute an event. The latter is only possible in RIAs.

2.3. Crawling Traditional Applications vs. Crawling RIAs

In traditional Web applications, there is a one-to-one relation between the set of DOMs reachable in the application and the URLs, thus a DOM can be identified by a URL. Hence, crawling traditional applications is relatively easy; starting from a given seed URL, it is possible to discover the client states by visiting those URLs that are found on the already visited client states.

In RIAs, however, one can reach many client states from a single URL. When an event execution modifies the DOM, the URL does not change. This means some client states in RIAs are not reachable directly by loading a URL, but rather are reachable by executing a sequence of events, starting from a given URL of the application. Some RIAs even have a single URL. There are also RIAs that use the traditional URL-based navigation together with RIA technologies. That is, an RIA may contain several URLs leading to different sections of the application and each section may contain pages reachable only via events. For this reason, a complete crawling approach for RIAs must provide a mix of exploration through execution of events as well as traditional URL-based exploration.

There is an important difference between following a URL and executing an event in the context of crawling: the result of an event execution depends on the state where the event is executed, whereas a URL always leads to the same state regardless of the state in which the URL is seen. Hence, it is sufficient to follow a URL once, but an event should be executed in each state where the event is seen. In this sense, following a URL cannot be treated as a "regular" event.

The traditional URL-based exploration is well understood [Olston and Najork 2010]. The focus of this article is on event-based exploration, that is, crawling those parts of the RIA that are reached through event executions under a given URL. This means the techniques we explain in the remainder must thus be applied to each distinct URL in the application for a complete coverage.

2.4. Model Representation

In event-based exploration, the aim is to start from a client state that can be directly reached by a URL and extract a model that contains all the client states reachable by event executions. This model can be conceptualized as a Finite State Machine (FSM). We formulate an FSM as a tuple $M = (S, s_1, \Sigma, \delta)$, where:

—S is the *finite set of client states*;
—$s_1 \in S$ is the *initial client state* of the URL;
—Σ is the set of all events in the application; and
—$\delta : S \times \Sigma \rightarrow S$ is the *transition function.*

The initial state s_1 is the client state that represents the page reached when the URL is loaded.

During exploration, the application can be in only one of its client states, referred to as the "current state".

For two client states s_i and s_j and an event e, if $\delta(s_i, e) = s_j$, then the application (modeled by the FSM M) performs a transition from the client state s_i to the client state s_j when the event e is executed in s_i. We denote such a transition by $(s_i, s_j; e)$. The client state from which the transition originates (s_i) is called the source state and the state to which the transition leads (s_j) is called the destination state of the transition.

The transition function δ is a partial function: from a given state s only a subset of the events in Σ can be executed. This subset contains those events associated with elements existing in the DOM represented by s. It is called the "enabled events" at s.

An FSM $M = (S, s_1, \Sigma, \delta)$ can be represented as a directed graph $G = (V, E)$, where:

—V is the set of vertices such that a vertex $v_i \in V$ represents the client state $s_i \in S$; and
—E is the set of labeled directed edges such that $(v_i, v_j; e) \in E$ iff $\delta(s_i, e) = s_j$. When it is not important, we omit the edge's event label and simply write (v_i, v_j).

In a graph, any sequence of adjacent edges is called a path. Given paths P, P', P_P, P_S, we say P' is a subpath of P if $P = P_P P' P_S$, where P_P and P_S are (possibly empty) prefix and suffix of P, respectively. The length of a path P is the number of edges in P.

2.5. Building a Complete Model

Under our working assumptions (Section 2.6), we aim at building a correct and complete model. Building a complete model means that every state (and every transition) will eventually be discovered. Achieving this requires to execute each enabled event at each discovered state. Since the result of an event execution may depend on the state in which the event is executed, it is not enough to execute the event from only one of the states where the event is enabled. We refer to the first execution of an event e from a state s as the "*exploration* of e from s" (sometimes, we say a transition is explored to mean that the corresponding event is explored from the source state).

To build a model for a given URL, initially we start with a single vertex representing the initial state of the URL. The initial state is reached by loading the URL. Then, the crawler identifies the enabled events on the initial state and explores one of the enabled events. After each event exploration, the model is augmented by adding a new edge for the newly discovered transition. If a new state is discovered, a new vertex is added to the model. When from each discovered state s every enabled event at s is explored, a complete model is obtained for the URL.

During crawling, the crawler often needs to move from the current state to another known (already discovered) state, in order to explore an event from the latter. This is either because all the enabled events of the current state might have already been explored or exploring an event from another state may seem more preferable to the crawler. In such cases, the crawler uses a *transfer sequence* to move to the desired state.

Definition 2.4 (*Transfer Sequence*). A transfer sequence is a sequence of already explored events executed by the crawler to move from one known state to another.

A transfer sequence corresponds to a path in the extracted model. One may ask why the crawler needs to execute a transfer sequence instead of storing each DOM it discovers and simply reloading the stored DOM of the desired state. However, this is not feasible: this requires the crawler to allocate a significant amount of storage to store the DOMs and, more importantly, in most RIAs, when a stored DOM is loaded to the memory, the functionality of the application will be broken since the JavaScript and the server-side context of the application will not be correct.

In some cases, the crawler executes a transfer sequence after a "*reset*".

Definition 2.5 (*Reset*). A reset is the action of going back to the initial state by loading the URL.

Sometimes, resetting may be the only option to continue crawling of a URL: we may reach a state where there is no enabled event, or the crawler needs to transfer to a state that is only reachable through the initial state of the URL and the crawler has not discovered a transfer sequence to the initial state from the current state.

2.6. Working Assumptions

When building a model, we make some limiting assumptions regarding the behavior of the application being crawled. The assumptions we make are in line with the ones made in most published work: determinism and user inputs.

2.6.1. Determinism. The behavior of the application is deterministic from the point of view of the crawler: from the same state, executing the same event leads to the same state[3]. Formally, the following is satisfied.

$$\forall s_x, s_y \in S\ \forall e \in \Sigma.\ s_x = s_y \wedge \delta(s_x, e) = s_k \wedge \delta(s_y, e) = s_l \Rightarrow s_k = s_l \tag{1}$$

Similarly, a reset is assumed to always lead to the same initial state.

A dependence of the application that is not directly observable on the client side, such as server-side states, can potentially violate this assumption. How to cope with such violations is not addressed in this article.

2.6.2. User Inputs. The second assumption is regarding the user-input values. Entering a user-input value is also considered as an event during crawling. However, the number of possible values that can be entered by a user are infinitely many (for example, consider the text that can be entered in a text field of a form), hence it is not usually

[3]Because of this assumption, we represent the model we are building as a deterministic FSM: δ is a function.

feasible to try every possible value during crawling. We assume that the crawler is provided with a set of user-input values to be used during crawling. We are not making any assumptions regarding the coverage of the provided set, but merely guarantee that the model being built is complete for the values provided.

How to a choose a subset of the possible user-input values that will provide a good coverage is out of scope of this article. There has been some research addressing this problem [Ntoulas et al. 2005; Wu et al. 2006; Lu et al. 2008]. Ideally, the subset provided to the crawler must be sufficient to discover all the states of the application.

2.7. Crawling Strategy

A crawling strategy is an algorithm that decides what event should be explored next. Under the working assumptions of Section 2.6, the basic goal of a crawling strategy is the construction of a correct and complete model of the application made by systematically exploring each enabled event at each discovered state.

2.7.1. Efficiency of a Crawling Strategy. Our aim is to efficiently produce a correct and complete model. Our definition of efficiency goes beyond producing the model as quickly as possible: in a real setting, the crawl may not run to the end due to lack of time. We note that states have more information value than the transitions, thus the primary goal of efficient crawl is to find as many states as possible, as early as possible in the crawl. This way, if the crawl is stopped before the end, a more efficient strategy will provide more information. In addition, we still aim for a complete model as quickly as possible, of course.

This efficiency of a strategy can be measured in terms of number of event executions and resets it requires to discover all the states, and then the entire model. Event executions and the resets are the actions that dominate the crawling time and solely depend on the decisions of the strategy. In general, a reset can be expected to take more time than an event execution because a reset loads an entire page from scratch whereas an event usually changes a page partially.

It seems clear that an efficient strategy for RIAs will have the following characteristics.

(1) An efficient strategy predicts exploring which events are more likely to discover a new state. Since the number of events to explore in a typical RIA is much larger than the number of states, a strategy cannot discover the states earlier on during the crawl if it cannot prioritize the events that will lead to new states.
(2) An efficient strategy minimizes the total length of *transfer sequences* used during the crawl since the purpose of executing a transfer sequence is not exploration.

2.7.2. Standard Crawling Strategies. Two existing and widely used crawling strategies are the *breadth first* and the *depth first*. Although these strategies work well with traditional applications, they are not efficient for crawling RIAs since they lack the mentioned characteristics of an efficient strategy: Neither strategy has a mechanism to predict which event is more likely to discover a new state. In addition, both strategies explore the states in a strict order that increases the number and the length of transfer sequences used by these strategies. That is, the breadth-first strategy explores the least recently discovered state first, whereas the depth-first crawling strategy explores the most recently discovered state first. Note that exploring a state means to explore every enabled event of the state. This implies, for example, when these strategies explore an event from a state s, and if as a result, another state s' is reached, the crawler needs to transfer from s' to s in order to finish remaining unexplored events in s (in the case of depth first, assume s' is a known state). A more efficient strategy would try to find an event to explore from the current state or from a state that is closer to the current state, rather than going back to the previous state after each event exploration.

2.8. DOM Equivalence

To be able to build a model, the crawler must decide, after each event exploration, whether the DOM it has reached corresponds to a new state or not. This is needed to avoid exploring the same states over and over again. Moreover, if the current DOM is not a new state, the crawler must know to which state it corresponds.

A simple mechanism is equality where two DOMs correspond to the same state if and only if they are identical. But equality is a very strict relation and not very useful for most applications. Web pages often contain parts that change when the page is visited at different times or that do not contain any useful information (for the purpose of crawling). For example, if the page contains timestamps, counters, or changing advertisements, using equality will fail to recognize a page when the page is visited at a later time, simply because these “unimportant” parts have changed (see Choudhary et al. [2012] for a technique that aims at identifying automatically the nonrelevant parts in a Web page).

More generally, the crawler could use a DOM equivalence relation[4]. A DOM equivalence relation partitions the DOMs into equivalence classes such that each equivalence class represents a state in the model. Using the DOM equivalence relation, the crawler decides whether the current DOM maps to an existing state in the model or not.

The choice of a DOM equivalence relation should be considered very carefully since it affects the correctness and the efficiency of crawling. If the equivalence relation is too strict (like equality), then it may result in too many states being produced, essentially resulting in state explosion, long runs, and in some cases infinite runs. On the contrary, if the equivalence relation is too lax, we may end up with client states that are merged while, in reality, they should be considered different, leading to an incomplete, simplified model.

Unfortunately, it is hard to propose a single DOM equivalence relation that can be useful in all situations. The choice of the DOM equivalence depends on the purpose of the crawl as well as the application being crawled. For instance, if the purpose of the crawl is content indexing, then the text content of pages should be taken into account. But, in the case of security analysis, the text content usually has no significance for deciding the equivalence of DOMs.

For the correctness of the model produced, it is important to have a DOM equivalence relation that is an equivalence relation in the mathematical sense (i.e., the relation must be reflexive, symmetric, and transitive). In addition, it is reasonable to constrain the equivalence relation such that the DOMs in the same equivalence class have the same set of enabled events, otherwise, two equivalent states would have different ways to leave them. This will result in a model that cannot be used reliably to move from one state to the other. When two DOMs with different sets of events are mapped to the same state, we can never be sure which set of events we are going to find in that state when we visit it again.

When implementing a DOM equivalence relation, it is important to use an efficient mechanism to decide the equivalence class of a given DOM. It is usually not feasible to store seen DOMs and compare a given DOM against all. For this reason,

[4]Mathematically, a binary relation, $\sim$ on a set A is an equivalence relation iff it has the following three properties: (1) reflexivity ($\forall a \in A.\, a \sim a$), (2) symmetry ($\forall a, b \in A.\, a \sim b \Rightarrow b \sim a$), (3) transitivity ($\forall a, b, c \in A.\, a \sim b \wedge b \sim c \Rightarrow a \sim c$). An equivalence relation partitions the underlying set, that is, divides the set into nonempty, disjoint subsets whose union covers the entire set. Each subset in the partition is called an equivalence class.

fingerprinting techniques are usually used to determine the equivalence class of a DOM. That is, when a DOM is reached, it is first transformed into a normalized form (for example, by removing unimportant components of the DOM) and the hash of this normalized DOM is produced. This hash value is stored and used to efficiently identify equivalent DOMs: if two DOMs have the same hash values then they are considered equivalent.

We note that DOM equivalence is a concept independent of the crawling strategy; in a nutshell, the DOM equivalence shapes the model of the RIA that must be inferred by any correct crawling strategy. Different (correct) crawling strategies will eventually yield the same model when crawling the same RIA with the same DOM equivalence, but some strategies will be more efficient than others in doing so.

2.9. Event Identification

Another challenge in crawling RIAs is identification of events. The crawler should be able to detect the enabled events at a state and produce identifiers to differentiate between these events. The event identification mechanism must be deterministic, that is, for an event at a state, the same event identifier must be produced every time the state is visited. This is required since the crawler must know whether an event has already been explored from the state or not. Also, to be able to trigger a known transition at a later time, the crawler needs to recognize the event that corresponds to the transition among the events enabled at the source state. The event identification is also important for DOM equivalence, since we require that two equivalent DOMs need to have the same set of enabled events.

In addition, an event identification mechanism should allow the crawler to recognize the instances of the same event at different states. Although it is still be possible to crawl an application without this capability, this is important for designing efficient crawling strategies. Recognizing instances of the same event at different states allows crawling strategies to make predictions about the event's behavior.

Since events are associated with DOM elements, the problem of event identification is related to unique identification of DOM elements. This is challenging since it is difficult to identify a single solution that would work for all applications. One may suggest using the path of an element from the root node in the DOM tree as an identifier, but this path changes if the place of an element changes in the tree. Similarly, one may be tempted to use the *id* attributes of the DOM elements, but this is not a complete solution on its own. This is because there can be elements with no id assigned. Moreover, although the ids of the elements in a DOM are supposed to be unique, there is no mechanism to enforce this, it is still possible to assign the same id to multiple elements in the same DOM. Also, there is no requirement for ids to be consistent across different DOMs. Generating the event identifier based on a combination of information about an element such as the values of some selected attributes, the number of attributes, and the element type can be possible, but in this case the question of which attributes to include/exclude becomes important.

Like DOM equivalence, event identification should be considered independent of the crawling strategy since a strategy works with any appropriate event identification mechanism.

3. RELATED WORK

For traditional Web applications, crawling is a well-studied problem [Arasu et al. 2001; Page et al. 1998] (see Olston and Najork [2010] for a recent survey of traditional Web crawling). For traditional applications, in addition to the fundamental problem of automatically discovering existing pages, research has been done on how to best use the model obtained after crawling. For example, the question of how often a page should be revisited to detect any possible changes on the page [Cho and Garcia-Molina 2003; Coffman et al. 1998], or ranking pages according to some "importance" metric in order to list more important pages first in the search results [Arasu et al. 2001; Page et al. 1998] are addressed. Note that, in the case of RIAs, the current research is still trying to address the fundamental problem of automatically discovering existing pages.

There has been some research focusing on crawling of RIAs, but to our knowledge there has not been much attention paid to the efficiency of crawling strategies, except for Peng et al. [2012]. Most of the existing approaches use either a breadth-first or depth-first crawling strategy, often with slight modifications. For the reasons explained in Section 2.7, these strategies are not efficient for RIAs. The major difference of our work from the existing ones is that our goal is not only being able to crawl RIAs, but doing so efficiently with better strategies. Another difference of our work is that we aim at obtaining a complete model that can be used for any purpose. Under our working assumptions, we guarantee that every state (and transition) will eventually be discovered. This is not always the goal in the related research.

Mesbah et al. [2008, 2012] introduced a tool, called Crawljax, for crawling AJAX applications. Crawljax converts AJAX applications to multipage static pages that can be used for indexing. The tool extracts an FSM model of the application using a variation of the depth-first strategy. One of the drawbacks of its default strategy is that it only explores a subset of the enabled events in each state. That is, only those events registered to DOM elements that are different from the previous state are explored. This default strategy may not find all the states, since executing a fixed event from different states may lead to different states[5]. In addition, Crawljax uses an edit distance (the number of operations needed to change one DOM to the other, the so-called Levenstein distance) to decide whether the current DOM corresponds to a different state than the previous one. If the distance is below some given threshold, then the current DOM is considered "equivalent" to a previous one. Since the notion of distance is not transitive, it is not an equivalence relation in the mathematical sense. Using a distance has the problem of incorrectly grouping together those client states whose distance is actually above the given threshold. The same group also published research regarding testing of AJAX applications: Mesbah and van Deursen [2009] focus on invariant-based testing, Bezemer et al. [2009] focus on the security testing, and Roest et al. [2010] consider the regression testing of AJAX applications.

Duda et al. [2009] do not propose a new crawling strategy but rather use the breadth-first strategy to crawl AJAX applications. They propose to reduce the communication costs of the crawler by caching the JavaScript function calls (together with actual parameters) that result in AJAX requests and the response received from the server. If a function call with the same actual parameters is made in the future, the cached response is used instead of making a new AJAX call. In Frey [2007], the author proposes a ranking mechanism for the states in RIAs. The proposed mechanism, called AjaxRank, is an adaptation of PageRank [Page et al. 1998]. Similar to PageRank, the AjaxRank is connectivity based. In the AjaxRank, the initial state of the URL is given more importance (since it is the only state directly reachable from anywhere), hence those states that are closer to the initial state also get higher ranks.

Amalfitano et al. [2008] focus on modeling and testing RIAs using execution traces. Their work is based on first manually obtaining some execution traces from user sessions. Once the traces are obtained, they are analyzed and an FSM model is formed by grouping together equivalent user interfaces according to an equivalence relation. In a later paper [Amalfitano et al. 2010], they introduced a tool, called CrawlRIA, that

[5]What is explained here is the Crawljax's default strategy. However, the tool can be configured to explore every event, in which case its crawling strategy becomes the standard depth-first strategy.

automatically generates execution traces using a depth-first strategy. That is, starting from the initial state, events are executed in a depth-first manner until a client state that is equivalent to a previously visited client state is reached. Then the sequence of states and events is stored as a trace and, after a reset, crawling continues from the initial state to record another trace. These automatically generated traces are later used to form an FSM model using the same technique used for user-generated traces. They do not propose a new crawling strategy.

In Peng et al. [2012], the authors suggested using a simple greedy strategy, that is, the strategy is to explore an event from the current state if there is an unexplored event. If the current state has no unexplored event, the crawler transfers to the closest state with an unexplored event. Except ours, this is the only work that proposed a different strategy than the standard strategies. The greedy strategy tries to minimize the transfer sequences, but still does not have a mechanism to decide whether an event is more likely to discover a new state.

4. MODEL-BASED CRAWLING

Our goal is to crawl RIAs “efficiently”, that is, to find all the client states as quickly as possible while being guaranteed that every given client state will eventually be found (under the working assumptions of Section 2.6). Without any knowledge of the RIA being crawled, it seems difficult to devise a general efficient strategy. For example, the breadth-first and depth-first strategies are guaranteed to discover a complete model when given enough time, but are usually not very efficient (for the reasons explained in Section 2.7).

In this section, we describe a generic framework, called “model-based crawling”, that can be used to design more efficient strategies: if we can identify some general patterns that we anticipate will be found in the actual models of the RIAs being crawled, we can use these patterns to forecast an anticipated model of the application. These anticipated models can be used as a guide for the crawling strategy that will be efficient if the application behaves as predicted by the anticipated model. To be correct, the strategy should also be able to gracefully handle “violations” occuring when the application does not behave as expected. We formally define the model-based crawling framework and identify the necessary steps that should be taken to devise a correct strategy in this framework.

4.1. Metamodel

We use the term *metamodel* to represent a class of applications that share certain behavioral patterns. A metamodel is defined by specifying the characteristics of the applications that constitute the instances of the metamodel. Different metamodels can be established using different sets of characteristics. These characteristics usually capture the relations of the events with the states and with the other events. For example, the characteristics may provide an answer to questions such as: is executing a particular event going to lead to a new state? or is executing a particular event going to lead to a state where there is a different set of events?, or which of the known states is reached when a particular event is executed? and so on. In model-based crawling, a metamodel is used as a means to anticipate the model of the application being crawled.

4.2. Methodology

We introduce a methodology called “model-based crawling” to design efficient strategies based on a chosen metamodel. Such a strategy uses the chosen metamodel as a guide for crawling. The strategy initially assumes that the application we are crawling is an instance of the chosen metamodel. Thus, the strategy will be very efficient (possibly, an optimal one) for crawling applications that are instances of the chosen metamodel.

However, this does not mean that applications that are not instances of the chosen metamodel cannot be efficiently crawled. In fact, the actual RIA will in practice almost never be a perfect match for the given metamodel. Model-based crawling must account for the discrepancy between the anticipated model and the actual model, allowing the strategy to adapt to the application being crawled.

To summarize, a model-based strategy is designed in three steps.

(1) A metamodel is chosen.
(2) A strategy optimized for those applications that follow the metamodel is designed. Ideally, the strategy must be optimal if the actual model is a perfect match for the metamodel.
(3) Steps to take are specified in case the application that is crawled deviates from the metamodel.

4.2.1. Actual Model vs. Anticipated Model. In model-based crawling we often talk about two models, namely the actual model of the application and the model we are anticipating to find according to the chosen metamodel.

Definition 4.1 (*Actual Model*). The *actual model* of a given application is the model discovered during crawling. As defined in Section 2, we represent the actual model as a graph $G = (V, E)$.

Definition 4.2 (*Anticipated Model*). The *anticipated model* of the application is the model we are anticipating to discover based on the metamodel characteristics. We represent the anticipated model also as a graph, written $G' = (V', E')$.

4.2.2. Choosing a Metamodel. A crucial step in model-based crawling is to find a metamodel that will allow us to anticipate the behavior of the application as accurately as possible. It is a challenge to find a good metamodel that is generic enough to cover most RIAs, but at the same time specific enough to allow making some valid anticipations. To find a good metamodel, common behaviors that apply to a majority of RIAs can be observed and experiments with different metamodels can be done. In the next section, we present the first metamodel we have experimented with as an example for the model-based crawling approach and provide references to some other metamodels that are devised after the hypercube. We discuss a possible solution to the challenge of finding good metamodels as part of the future works.

4.2.3. Designing an Optimized Strategy. When designing a strategy for a given metamodel, we often use a two-phase approach.

—*The state exploration phase* is the first phase that aims to discover all those states anticipated by the metamodel as efficiently as possible. Given the extracted model so far and assuming the unexplored parts of the application will follow the metamodel anticipations, it is possible to know whether there is any more state to be discovered. Once, based on these anticipations, it is decided that there is not any new state to discover, the strategy moves on to the second phase.
—*The transition exploration phase* is the second phase, exploring those events that have not been previously explored. In the state exploration phase, the crawling strategy does not necessarily explore every event; in this first phase, the strategy only explores those events which it anticipates will help discovering new states. However, we cannot be sure that we have discovered all the states unless each event is explored. If a new state is discovered in the transition exploration phase, or the strategy anticipates that more states can be discovered, then the strategy switches back to the state exploration phase.

4.2.4. Handling Violations. During the crawl, whenever a discrepancy between the actual model and the anticipated model is detected, the anticipated model and the strategy are revised according to the actual model uncovered so far. There may be different ways to achieve this, but one simple and consistent mechanism is to assume that the characteristics of the metamodel will still be valid for the unexplored parts of the application. So, using the same characteristics, a strategy can be obtained for the unexplored parts as was done initially.

After the violations are handled, the anticipated model must conform to the actual model. This means the violation handling mechanism makes sure that the actual model is a subgraph of the anticipated model. The difference between the two models lies in the anticipated states yet to be discovered V^A and unexplored anticipated transitions E^A (i.e., $V' = V \cup V^A$ and $E' = E \cup E^A$).

5. OVERVIEW OF THE HYPERCUBE METAMODEL AND ITS STRATEGY

In the following, we present the hypercube metamodel and an optimal strategy to crawl the instances of this metamodel as an example of the model-based crawling methodology. We have introduced the idea of using a hypercube as a metamodel for crawling RIAs in Benjamin et al. [2010]. In Benjamin [2010] and Benjamin et al. [2011], we have presented an initial version of the hypercube strategy that is much improved in this article. In this section, we give an outline of the hypercube metamodel and of its strategy, and we briefly discuss two other metamodels. In the next section, we provide the technical details and proofs for the hypercube strategy.

5.1. The Hypercube Metamodel

As its name suggests, the hypercube metamodel is the class of models that have a hypercube structure. The hypercube metamodel is formed based on two assumptions.

—*A1*. The events that are enabled in a state are pairwise independent. That is, in a state with a set of enabled events, executing a given subset of these events leads to the same state regardless of the order of their execution.
—*A2*. When an event e is executed in state s, the set of events that are enabled in the reached state is the same as the events enabled in s minus e.

These assumptions reflect the anticipation that executing an event does not affect (enable or disable) other events and executing a set of events from the same state in different orders is likely to lead to the same state. Based on these assumptions, the initial anticipated model for an application whose initial state has n events is a hypercube of dimension n. Figure 1 shows a hypercube of dimension 4. The vertex at the bottom of the hypercube represents the initial state with four events $\{e1, e2, e3, e4\}$ enabled. Initially, the remaining vertices represent the anticipated states. Each vertex is labeled by the events enabled in the state (for readability, not all the labels are shown). Each edge is directed from the lower incident vertex to the upper incident vertex and represents an initially anticipated transition of the application.

In a hypercube of dimension n, there are 2^n states and $n \times 2^{n-1}$ transitions. The height of a state in the hypercube is the number of transitions that must be traversed to reach it from the initial state. The set of states in a hypercube of dimension n can be partitioned as $\{L_0, L_1, L_2, \ldots, L_n\}$, where L_i is the set of states of height i. We call L_i the "level i" of the hypercube. L_0, L_n, and $L_{\lfloor n/2 \rfloor}$ are called the bottom, the top, and the middle of the hypercube, respectively. We refer to all levels higher than the middle as the "upper half" and levels lower than the middle as the "lower half" of the hypercube.

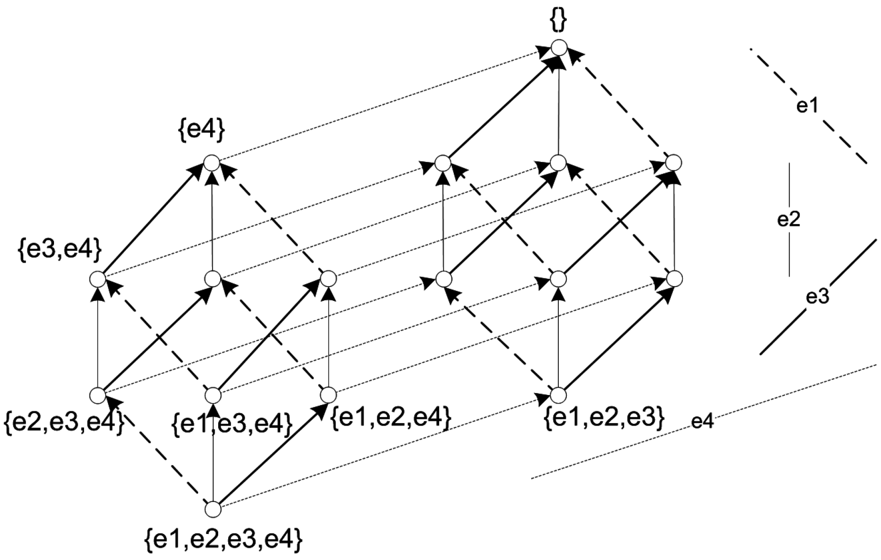

Fig. 1. A hypercube of dimension 4.

5.2. Violations of the Hypercube Assumptions

When the RIA does not fully follow the hypercube metamodel, we have "violations" of the hypercube assumptions. With this metamodel, there are four possible violations that are not mutually exclusive. The first metamodel assumption *A1* can be violated in two ways.

—*Unexpected Split.* In this case, after executing an event, we expect to reach a state that has already been visited but we actually reach a new state.
—*Unexpected Merge.* In this case, after executing an event, we unexpectedly reach a known state (i.e., not the expected known state).

A2 can also be violated in two ways.

—*Appearing Events.* There are some enabled events not expected to be enabled in the reached state.
—*Disappearing Events.* Some events expected to be enabled in the reached state are not enabled.

As explained before, we need to have an efficient strategy that handles all four violations.

5.3. The Strategy for the Hypercube Metamodel

For the hypercube metamodel, we presented an optimal crawling strategy in Benjamin [2010] and Benjamin et al. [2011]. We provide an overview of these algorithms here; all the technical details and proofs are given in Section 6. These algorithms use the characterization of the hypercube as a partially ordered set to produce an efficient strategy.

A hypercube is the partially ordered set of all subsets of n elements ordered by inclusion. In our case, each subset of the n events found in the initial state represents a state in the hypercube. This characterization is useful for generating an optimal state exploration strategy for a hypercube model. In a partially ordered set, a set of pairwise comparable elements is called a chain. Thus, each directed path in the hypercube is a chain. A set of chains covering every element of the order is known as a chain decomposition of the order. A *Minimum Chain Decomposition (MCD)* of the order is a chain decomposition of minimum cardinality (see Anderson [1987] for an overview of concepts). For the hypercube model, a minimum chain decomposition is a set A of paths that contain every state in the hypercube such that A is of minimal cardinality with this property (i.e., following an MCD of a hypercube allows us to visit every state in the hypercube using the minimum number of events and the minimum number of resets). Dilworth [1950] proved that the cardinality of any minimal chain decomposition is equal to the *width* of the order, that is, the maximum number of pairwise noncomparable elements. In a hypercube of dimension n, the width is the number of states in the middle level which is $\binom{n}{\lfloor n/2 \rfloor}$. A minimum chain decomposition algorithm that can also be used for hypercubes is given in Bruijn et al. [1951]. A chain given by this algorithm is of the form $C =< v_i, v_{i+1}, \ldots, v_{i+k} >$, where v_i is a state at level i. In this decomposition, each chain is a unique path in the hypercube, but it does not necessarily start from the bottom of the hypercube. In a chain C the state at the lowest level is called the *chain starter state* or the *bottom* of C. The set of MCD chains produced for a hypercube of dimension 4 (shown in Figure 1) is the following.

(1) $< \{e1, e2, e3, e4\}, \{e2, e3, e4\}, \{e3, e4\}, \{e4\}, \{\} >$
(2) $< \{e1, e2, e3\}, \{e2, e3\}, \{e3\} >$
(3) $< \{e1, e2, e4\}, \{e2, e4\}, \{e2\} >$
(4) $< \{e1, e2\} >$
(5) $< \{e1, e3, e4\}, \{e1, e4\}, \{e1\} >$
(6) $< \{e1, e3\} >$

By definition, an MCD provides a complete coverage of the states of a hypercube in an optimal way, that is, using the minimum number of events and the minimum number of resets. Thus, for state exploration it is enough to generate an MCD of the hypercube. However, an MCD does not cover all the transitions of the hypercube. In the initial hypercube strategy, we devised another algorithm that generates a larger set of chains, called Minimum Transition Coverage (MTC), to cover all the transitions in a hypercube in an optimal way. Since MCD chains already traverse some of the transitions, the MTC algorithm can be constrained with an already generated MCD so that the set of MTC chains contains every MCD chain. The number of paths in an MTC is $\binom{n}{\lfloor n/2 \rfloor} \times \lceil n/2 \rceil$, which is the number of transitions leaving the middle level.

The combination of MCD and MTC provides an optimal way of crawling an RIA that perfectly follows the hypercube metamodel. That is, for a hypercube, MTC uses the minimal number of resets and event executions to traverse each transition at least once. Moreover, among MTC chains the ones that contain the MCD chains are given exploration priority. Thus all the states of the hypercube are visited first, using the minimal number of events and resets.

The algorithm also provides a revision procedure to update the existing set of chains to handle violations of the hypercube assumptions. The revision procedure basically replaces the chains that become invalid and adds new chains if necessary.

5.4. An Example

To better explain the concepts of the anticipated model and the violations of the hypercube assumptions, we provide a simple example. The example details the partial exploration of an example application whose model is shown in Figure 2. The initial state of the application $s1$ has three enabled events $\{e1, e2, e3\}$ and there are 8 states in total.

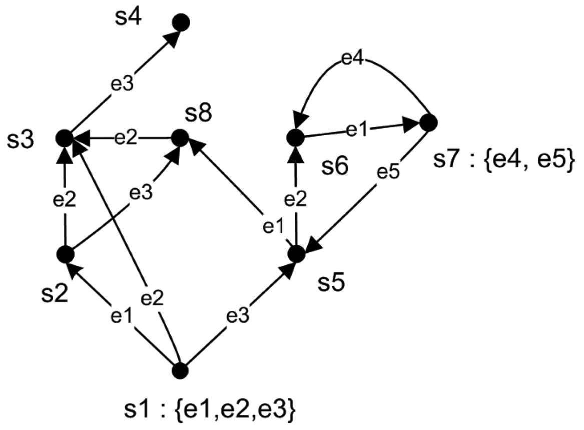

Fig. 2. Model of the example application.

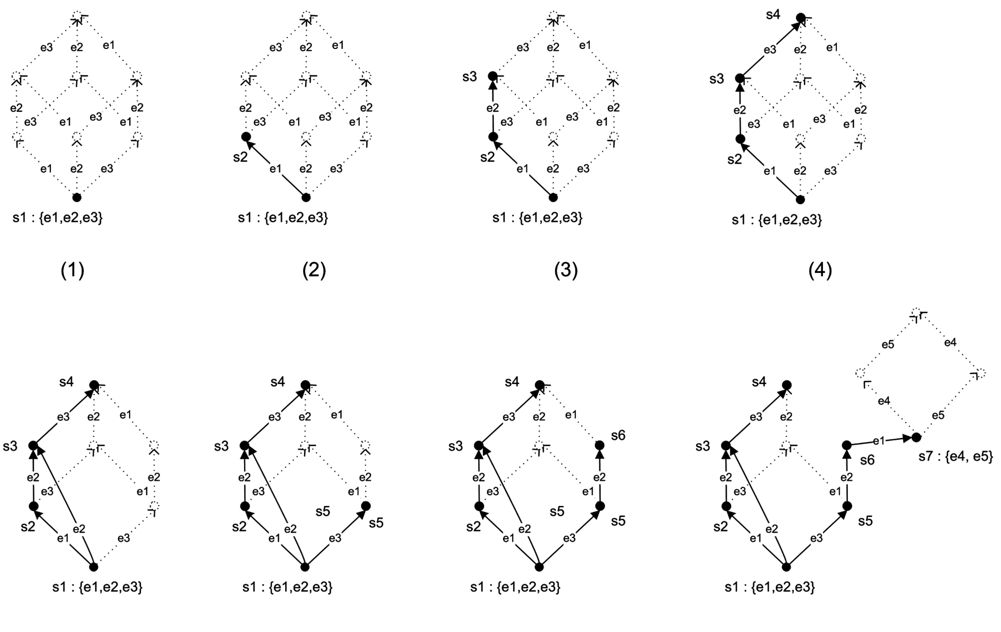

Fig. 3. (Partial) crawling of the example application.

Figure 3 shows (partially) the steps taken by the hypercube strategy to crawl the application in Figure 2. Each diagram in Figure 3 shows the current anticipated model: the solid nodes and edges show the actual discovered model whereas dashed nodes and edges belong to the anticipated model only.

The first diagram, 3.(1), shows the initial situation: the only discovered state is $s1$ and the anticipated model is a hypercube of dimension 3 based on $s1$. The MTC chains that are generated for crawling this hypercube are listed next. The highlighted sequences are the MCD chains.

(1) $< \{\mathbf{e1}, \mathbf{e2}, \mathbf{e3}\},\{\mathbf{e2}, \mathbf{e3}\},\{\mathbf{e3}\}, \{\} >$
(2) $< \{e1, e2, e3\}, \{\mathbf{e1}, \mathbf{e3}\},\{\mathbf{e1}\}, \{\} >$
(3) $< \{e1, e2, e3\}, \{\mathbf{e1}, \mathbf{e2}\},\{\mathbf{e2}\}, \{\} >$
(4) $< \{e1, e2, e3\}, \{e2, e3\}, \{e2\} >$
(5) $< \{e1, e2, e3\}, \{e1, e3\}, \{e3\} >$
(6) $< \{e1, e2, e3\}, \{e1, e2\}, \{e1\} >$

The hypercube strategy starts with the execution of the first chain. This is shown in diagrams 3.(1)–(4). In this example, the first chain is executed without any violations and the states $s1, s2, s3, s4$ are discovered as anticipated.

3.(5) shows the situation after executing the first event ($e2$ from $s1$) of the second chain. In this case, we were anticipating to reach a new state that has enabled events $\{e1, e3\}$ but we have reached an already discovered state ($s3$) that has $e3$ as the only enabled event. This is a violation of both *A1* and *A2* since we have an unexpected merge and a disappearing event. Note that, after this violation, the anticipated model is updated. The state we were expecting to reach is removed since this was the only way to reach it in the model (this also means that the 5th chain is not valid anymore, since it was supposed execute an event from the unreachable anticipated state, thus it has to be removed as well). After this violation we cannot continue with the current chain; the strategy moves on to the third chain.

3.(6) and 3.(7) show the execution of the first two events in the third chain. These executions do not cause any violations. However, as shown in 3.(8), the last event in the chain causes a violation. We were expecting to reach $s4$ by executing $e1$ from $s6$, but we reached to a new state $s7$ with two enabled events $\{e4, e5\}$. This is a violation of both *A1* and *A2*: it is an unexpected split and there are appearing/disappearing events in the state reached. After this violation, the anticipated model is updated by adding a new hypercube of dimension 2 based on $s7$. This means new chains have to be computed and added to the set of existing chains while the current chain becomes invalid.

As the concepts of an anticipated model and the violations are depicted in Figure 3, we do not show the further exploration steps that would be performed by the strategy, until all the events are explored.

5.5. Other Model-Based Crawling Strategies

The hypercube strategy presented here and detailed in the next section is an elegant one: the metamodel is a simple, formal, and well-studied mathematical object (the hypercube) and the proposed strategy optimal. Thus, it is a good choice to illustrate the concepts behind model-based crawling. Unfortunately, the assumptions that are made about the RIAs being crawled, described in Section 5.1, are not commonly found in actual RIAs. In practice, as will be shown in Section 7, when crawling “real-life” RIAs the strategy is faced with violations of the metamodel assumptions fairly often. Consequently, while still much better than depth-first and breadth-first methods, the results on these real RIAs are not as good as what is achieved with metamodels that are closer to the reality of RIAs’ behaviors.

Other metamodels have been proposed based on model-based crawling. These other models are better representative of real RIAs and, as a consequence, yield better results than the hypercube on our tests. These metamodels and their strategies have been described elsewhere. We do include here a brief description of two of them, to show that model-based crawling can indeed be used to create other strategies that are more efficient.

The first such model is the Menu model [Choudhary 2012; Choudhary et al. 2013]: in this metamodel, each event is classified in one of three categories. The “menu” events are events that always bring the application to the same state, regardless of the state from which these events were executed. The "self-loop" events are events that always bring the application back to the state from which these events were executed. Finally, the other events are unclassified and no assumption is made regarding what happens when they are executed. The strategy consists in assigning each event to its category and using the inferred model rapidly to find new states (see Choudhary et al. [2013] for details).

The second metamodel is the Probability model [Dincturk 2013; Dincturk et al. 2012]: in this metamodel, instead of assigning events to strict categories, a probability of finding a new state is calculated for each event. It can be seen as a refinement of the menu model with a dynamic range of categories in which events are assigned, and where the category in which an event is assigned changes over time. The strategy consists in computing the probability of each event using a Bayesian formula that includes the history of event execution up to that point, and using the current set of probability to decide which event to explore next (see Dincturk et al. [2012] for details).

Of course, many other metamodels and corresponding strategies can be created based on the model-based crawling. If an efficient strategy can be found for the metamodel and if the metamodel is a good representative of the RIAs being crawled, the result of the crawl will be good. As more metamodels and efficient strategies for these metamodels are added to the framework, it will eventually be possible to build a "metacrawler" that selects the best metamodel on-the-fly based on the actual behavior of the RIAs being crawled.

6. THE HYPERCUBE STRATEGY: TECHNICAL DETAILS AND PROOF OF OPTIMALITY

In order to execute the hypercube strategy in an efficient manner, the whole strategy cannot be generated beforehand, because the anticipated model has a size exponential in the number of events enabled in the bottom state. For example, if we attempted to crawl an RIA that has 20 events on its initial page (which is in fact a very small number), we would need to generate 1,847,560 chains. If the application being crawled happens not to fit the hypercube model, then this precomputation would be largely in vain.

Fortunately, the selection of the next event to execute can be made on-the-fly. For the state exploration phase, we need the ability to identify the successor of a state in its MCD chain. That is, for any state in the hypercube we need a way to figure out which event to execute to reach the next state in the MCD chain. If we can do this, then we can execute an MCD chain step by step without costly precomputation overhead and still have an optimal strategy. We then need to do the same thing with the transition exploration, which is going to be simpler to achieve with simple greedy approach. Since there are no stored chains to maintain, in case of violations the revision of the strategy will also be simplified. In the following, we detail this efficient execution of the hypercube strategy and provide algorithms for this purpose.

Algorithm 1 shows the global variables and the main body of the hypercube strategy that extracts a model for a given URL. The global variables are listed next.

—v_1 is a vertex representing the initial state. We assume the method `Load` loads the given URL and returns a vertex that represents the initial state.
—$G = (V, E)$ is the extracted model, initially $G = (\{v_1\}, \emptyset)$.
—$G' = (V', E')$ is the anticipated model, initially a hypercube based on v_1. Note that the anticipated model (which can be very large) is not actually constructed in memory. It is merely a hypothetical graph that we use for explanation purposes. The structure

of a hypercube allows us to know all the transitions from any state without actually creating the anticipated model.

—$v_{current}$ is a reference to the vertex representing the current state. It is updated after each event execution.

—*phase* shows the current phase of the crawling strategy. It can have one of the three possible values: *stateExploration*, *transitionExploration*, or *terminate*. It is initialized to *stateExploration*. Crawling continues until its value becomes *terminate*.

ALGORITHM 1: The Hypercube Strategy

```
Input: url: the URL of the application to crawl.
Output: $G = (V, E)$: the model of the application
global $v_1$ = Load(url) ;
global $V = \{v_1\}$, $E = \emptyset$ ; // the actual model
global $G' = (V', E')$ = A hypercube graph based on $v_1$ ; // the anticipated model
global $v_{current} = v_1$;
global $phase = stateExploration$;
while $phase$ != $terminate$ do
    if $phase == stateExploration$ then
        StateExploration();
    else
        TransitionExploration();
    end
end
```

6.1. State Exploration Phase

The optimal state exploration strategy for a hypercube is to follow an MCD of the hypercube. However, we must do it without generating the chains in the MCD ahead of time. The key to achieve this is the ability to determine the state that comes after the current state in an MCD chain, without generating the chain beforehand. For an MCD, we call the state that follows a given state v in the MCD chain as the MCD successor of v and write MCDSuccessor(v). In addition to the successors, we also need to identify from which state an MCD chain starts: the *chain starter*.

6.1.1. Identifying MCD Successors and Chain Starters. Aigner [1973] and Greene and Kleitman [1976] present two different approaches for defining the successor function MCDSuccessor that gives a minimal chain decomposition of a hypercube (according to Griggs et al. [2004], both approaches yield the same decomposition that is also produced by the algorithm given in Bruijn et al. [1951]).

The approach of Greene and Kleitman [1976] is based on parenthesis matching and works as follows: since each state in the hypercube is characterized by a subset of the set of events enabled at the initial state ($\{e_1, e_2, \ldots, e_n\}$), a possible representation of a state v in the hypercube is an n-bits string representation $x_1x_2\ldots x_n \in \{0, 1\}^n$ such that the bit x_i is 0 if and only if e_i is enabled at v. To find MCDSuccessor(v), we use this bit string representation of v. We regard each 0 as a left parenthesis and each 1 as a right parenthesis and match the parentheses in the traditional manner as shown in the Function `MCDSuccessor`.

The function keeps track of a set called *IndexesOfUnmatchedZeros*. The set is empty initially and will contain the indexes of the unmatched zeros at the end. The function starts from the leftmost bit x_1 and scans the string such that when a 0 bit is encountered, the index of the 0 bit is added temporarily to the set. When a 1 bit is encountered, it is matched with the rightmost unmatched 0 bit to the 1 bit's left. This is achieved by removing from the set the maximum value. At the end, if the set is empty (i.e., all 0's are matched) then v has no successor. That means v is the last state in the MCD chain. Otherwise, the minimum index stored in the set is the index of the leftmost unmatched 0 bit. We obtain the bit string of MCDSuccessor(v) by flipping the bit at that position. That means, if i is this minimum index then we have to execute event e_i to reach the MCDSuccessor(v) from v. Using this simple method, an MCD chain can be followed without precomputation. In addition, the starting states of these MCD chains are the states whose bit strings do not contain any unmatched 1 bits.

For instance, if the bit string representation of a state v is 1100110001 then we have the following parenthesis representation

```
))(())((()
1100110001
```

where the leftmost unmatched 0 (left parenthesis) is the seventh bit so we have to execute the corresponding event (i.e., the seventh event among the events enabled at the bottom of the hypercube) from v to reach the successor of v, MCDSuccessor(v) = 1100111001. In addition, since the bit string of v contains unmatched 1's (the first and the second bits), v is not a chain starter state.

Function MCDSuccessor(v)

```
Input: a vertex $v$
Output: the MCD successor of $v$
$IndexesOfUnmatchedZeros = \emptyset$;
Let $x_1x_2\ldots x_n \in \{0,1\}^n$ be the bit string representation of $v$;
$i = 1$;
while $i <= n$ do
    if $x_i == 0$ then
        $IndexesOfUnmatchedZeros = IndexesOfUnmatchedZeros \cup i$;
    else
        $IndexesOfUnmatchedZeros = IndexesOfUnmatchedZeros \setminus$
        MAX($IndexesOfUnmatchedZeros$);
    end
end
if $IndexesOfUnmatchedZeros == \emptyset$ then
    return $nil$;
else
    $bitStringSuccessor$ = FlipTheBitAt($x_1x_2\ldots x_n$, MIN($IndexesOfUnmatchedZeros$));
    return the vertex corresponding to bitStringSuccessor;
end
```

6.1.2. Execution of the State Exploration Phase. The procedure `StateExploration` describes the execution of the state exploration phase. In this phase, we follow the MCD chains one by one using the successor function described earlier. In order to execute an MCD chain, we first need to find a chain starter state whose MCD chain has not been executed. At the very beginning of the crawl, since the initial state is a chain starter state, we can immediately start by executing the corresponding MCD chain. But, if the current state is not a chain starter, we find a path from the current state to the closest chain starter that we have not tried to reach before (note that the chain starter state is an anticipated state; it is possible that the chain starter we are expecting to reach does not exist)[6]. We use the path to attempt to reach to the chain starter. (To execute a path, the function `ExecutePath` is used. This function, which will be given later, executes the events in the path one after the other, and if needed updates the actual and anticipated models. If a violation is detected during the execution of the path, `ExecutePath` returns with value false.) If there is no violation, we do reach the chain starter and we start the execution of the corresponding MCD chain. If a violation occurs, we stop the execution of the current chain (which is not valid anymore) and start looking for another chain starter. The state exploration phase finishes when we have tried to execute all MCD chains for the current anticipated model.

Procedure StateExploration

```
while there is a chain starter that we have not yet attempted to reach do
    Let $P$ be a path in $G'$ from $v_{current}$ to the closest such chain starter;
    // try to reach to the chain starter
    if ExecutePath($P$) == $TRUE$ then
        // execute the MCD chain
        $v_{successor}$ = MCDSuccessor ($v_{current}$);
        while $v_{successor}$! = $nil$ do
            if ExecutePath(($v_{current}, v_{successor}$)) == $FALSE$ then
                break;
            else
                $v_{successor}$ = MCDSuccessor(($v_{successor}$));
            end
        end
        // if the end of chain is reached, extend the chain
        if $v_{successor}$ == $nil$ then
            while there is an unexplored transition $(v_{current}, v'; e)$ do
                // explore the event and check for violation
                if ExecutePath(($v_{current}, v'; e$)) == $FALSE$ then
                    break;
                end
            end
        end
    end
end
$phase = transitionExploration$;
```

When executing this strategy, there are some additional steps that we must take in order to preserve the optimality of the hypercube strategy. First of all, during the state exploration phase, we explore more than the MCD chains. In addition to the MCD chains, whenever possible, we try to traverse unexplored transitions on the path used to reach a chain starter (rather than using already explored transitions). Also, when we come to the end of an MCD chain, we continue exploring transitions rather than immediately starting the next MCD chain. In other words, the MCD chains are extended towards the bottom and the top using unexplored transitions. Otherwise, those transitions that we have not explored during state exploration while we had the opportunity will cause at least one extra event execution and possibly one extra reset. Moreover, when constructing a path P, we want it to be in the following form: $P = P_P P_S$,

[6]If there are multiple such chain starters, we choose the one whose MCD chain is longer. This is because a longer MCD chain means more anticipated states to discover. In an n-dimensional hypercube, the length of an MCD chain whose chain starter is at level l is given by the formula: $n - 2 \times l$.

where P_P is a (possibly empty) prefix path that consists of already explored transitions and P_S is a path that contains only previously unexplored transitions. This means, in a path, all the transitions that follow the first unexplored transition should also be unexplored. In particular, those paths that are used to reach a chain starter during the state exploration phase should contain, as much as possible, unexplored transitions and there should not be an already explored transition following an unexplored transition. Based on the same principle, the paths that extend the MCD chain toward the top and the paths taken during the transition exploration (the phase explained next) should end when a state without unexplored transitions is reached. In that case, we should go back to the bottom of the hypercube and start exploring a new chain.

6.2. Transition Exploration Phase

The procedure `TransitionExploration` describes the execution of the transition exploration phase. In this phase, we use a simple greedy strategy to explore the remaining unexplored events. The strategy is to always explore an unexplored transition that is closest to the current state. That is, we search in the actual model the shortest path from the current state to a state which has an unexplored event. We reach that state using the path, execute the unexplored event, and check whether the state that is reached violates hypercube assumptions. Any violation is handled as we explain next.

Procedure TransitionExploration

```
while $phase == transitionExploration$ do
    if $there is an unexplored transition$ then
        Let $(v, v'; e) \in E'$ be the closest unexplored transition to $v_{current}$;
        Let $P$ be a path constructed by appending $(v, v'; e)$ to a shortest path from $v_{current}$ to $v$;
        ExecutePath($P$);
    else
        $phase = terminate$;
    end
end
```

The crawl terminates when all the enabled events in each discovered state are explored.

6.3. Executing Events, Updating the Models and Handling Violations

To execute transitions we call the function *ExecutePath*. Given a path, the function triggers the transitions in the path one after the other, possibly after a reset. A transition is triggered by executing its event. We assume a method called `Execute` executes the given event from the current state and returns a vertex representing the state reached. That is, if the event execution leads to a known state, `Execute` returns the vertex of the state. Otherwise, `Execute` creates and returns a new vertex for the newly discovered state (we use the method `Execute` as a notational convenience combining event execution and the DOM equivalence relation).

The path provided to `ExecutePath` may contain both anticipated transitions (i.e., not yet explored) and already explored transitions. The return value of `ExecutePath` shows whether a violation of the hypercube assumptions is detected during the execution. If a violation is detected, the function returns immediately with value false.

After each explored transition, we must update the actual model, check for the violations of the hypercube assumptions, and, if needed, update the anticipated model. The function `Update` describes the mechanism to update the models and to handle the violations. The returned value of the function is false when a violation is detected.

```
Function ExecutePath(P)
Input: a path to traverse
Output: FALSE if any hypercube assumption is violated, otherwise TRUE
if P requires reset then
    $v_{current}$ = Load(url);
end
foreach transition $(v, v'; e) \in P$ from the first to the last do
    $v_{current}$ = Execute($e$);
    if $(v, v'; e) \in E^A$ then // is this an event exploration?
        if Update($(v, v'; e)$) == FALSE then
            return FALSE;
        end
    end
end
return TRUE;
```

The function Update is given the transition that has just been explored. The function adds this transition to the actual model. If a new state is reached, it also adds a new state to the actual model. Then it checks for a violation.This is checked by the expression $v_{current}\ != v'$. A violation is detected if the inequality holds. Here, we are checking the inequality of two vertices: one representing the state reached, $v_{current}$ (an actual state) and the other, v', representing the state that we were anticipating to reach. The latter can be representing either an anticipated state or an actual state. The semantics of the comparison is different in these cases. If v' represents an actual state (i.e., we have just explored a transition that was anticipated to connect two known states), then we just check whether the vertices represent different states. Otherwise, if v' represents an anticipated state, then the inequality is satisfied only if the reached state is not new or the enabled events on the new state do not satisfy the hypercube assumption *A2*.

When there is a violation, we update the anticipated model, still assuming that the hypercube assumptions (*A1* and *A2*) remain valid for the unexplored parts of the application. For this reason, if we unexpectedly reach a new state, we add a new (anticipated) hypercube to the anticipated model. Note also that in such a case the

```
Function Update($(v, v'; e)$)
Input: the transition recently explored
Output: FALSE if any hypercube assumption is violated, otherwise TRUE
$E = E \cup \{(v, v_{current}; e)\}$ ; // add a new transition to the model
$isNewState = v_{current} \notin V$ ; // is this a new state?
if $isNewState$ then
    $V = V \cup v_{current}$;
end
$isViolated = v_{current}\ != v'$ ; // is this a violation?
if $isViolated$ then
    if $isNewState$ then
        add to $G'$ a hypercube based on $v_{current}$;
        $phase = stateExploration$;
    end
    $E' = E' \setminus \{(v, v'; e)\}$ ; // remove the violated, anticipated transition
    remove from $G'$ any vertex that has become unreachable;
end
return $!isViolated$;
```

phase is reset to *stateExploration*. In case of any violation, we have to remove the anticipated transition and any unreachable anticipated states from the anticipated model (again, since the anticipated model is a hypothetical graph, we do not actually remove anything in the real implementation of the strategy).

6.4. Complexity Analysis

The worst-case time complexity of the hypercube strategy can be analyzed in terms of the size of the actual model of the application, $|E|$, and the maximum number of enabled events at a state, denoted n (i.e., n is the maximum outdegree of the actual model G). In the state exploration phase, we look for the chain stater state closest to the current state. To find such a state, we start traversing the actual model built so far in a breadth-first manner (note that we are searching the graph in memory rather than executing any event). During the traversal, we consider whether any of the unexplored transitions leads to a chain starter state. Checking if a transition leads to a chain starter is done using the parenthesis matching method explained in Section 6.1.1. This method requires to scan the bit string representation of a state having at most n bits and this takes $O(n)$ time. The number of unexplored transitions is at most $|E|$, so a traversal to look for chain starter takes $O(n \times |E|)$. Since there can be at most $|V|$ chain starter states, this traversal is done at most $|V|$ times. During the crawl, the total time spent looking for a chain starter is $O(n \times |E| \times |V|)$. Once a chain starter is found, we start following the MCD chain. Finding the MCD successor of a state requires $O(n)$ time. This calculation is done at most once for each state discovered, hence, the total time for following MCD chains is $O(n \times |V|)$.

Therefore, the complexity of the state exploration algorithm is $O(n \times |E| \times |V|) + O(n \times |V|) = O(n \times |E| \times |V|) = O(n \times |E|^2)$

For the transition exploration phase, we search in the actual model for the closest unexplored transition. Again, to find such a transition, we traverse the actual model starting from the current state in a breadth-first manner, which requires $O(|E|)$ time. This traversal is done at most $|E|$ times.

Hence, the complexity of the transition exploration algorithm $O(|E| \times |E|) = O(|E|^2)$ thus, the overall complexity is $O(n \times |E|^2) + O(|E|^2) = O(n \times |E|^2)$.

The hypercube strategy does not have a significant overhead compared to the greedy strategy or the *optimized* implementations of the breadth-first and the depth-first crawling strategies (we say that an implementation of a standard crawling strategy is *optimized* if the implementation always uses the shortest known transfer sequence, as will be explained in Section 7.2). These strategies have complexity $O(|E|^2)$; the factor n is the overhead of the hypercube strategy.

6.5. Optimality of the Hypercube Strategy

In this section, we show that the hypercube strategy is optimal for applications that are instances of the hypercube metamodel. In particular, we show that the hypercube strategy is optimal for both the number of resets and the number of event executions for the complete crawling of a hypercube model.

In addition to the optimality of the complete crawling (exploring all transitions), we are also concerned with finding all the states in the hypercube first. The number of resets required to visit all the states of the hypercube is clearly optimal since we are using an MCD of the hypercube, which is by definition the smallest number of chains (thus of resets) to go over all the states. However, for the number of events executed to visit all the states, the hypercube strategy is deliberately not optimal. This is because when we come to the end of an MCD chain we continue exploration from the current state instead of resetting and executing another MCD chain immediately, in order to keep the number of resets required for the overall crawling at the optimal value. If we do not extend the MCD chains, then this number is optimal, but in that case the number of resets is not optimal anymore for the complete crawl. This is because we will need to reset again later during the transition exploration phase to visit these transitions.

6.5.1. Number of Resets. First, we consider the number of resets. Note that, in a hypercube, each transition leaving a state at level i enters a state at level $i+1$. Once we are at a state at level i, unless we reset, there is no way to reach another state at level $j \leq i$. This means that if there are k transitions leaving level i, then to traverse each one at least once, we have to reset the application k times (counting the first load as a reset). Since the level with the largest number of outgoing transitions is the middle level, the lower bound on the number of resets for crawling the hypercube is $r^* = \binom{n}{\lfloor n/2 \rfloor} \times \lceil n/2 \rceil$, where $\binom{n}{\lfloor n/2 \rfloor}$ is the number of states at the middle level and $\lceil n/2 \rceil$ is the number of outgoing transitions for each state at the middle level.

We first introduce the notation that will be used in the following. Let $C_H = \{C^1, C^2, \ldots, C^m\}$ denote the set of all chains executed by the hypercube strategy when crawling a hypercube of dimension n. In particular $C^i = \{v_0^i, v_1^i, \ldots, v_k^i\}$ represents the ith chain executed by the strategy where v_j^i is the state at level j. We show that the number of chains executed by the hypercube strategy for crawling a hypercube is r^* (i.e., $m = r^*$) so only r^* resets are used by the hypercube strategy. We begin by explaining the following properties of the hypercube strategy.

LEMMA 6.1. *Let $C^u \in C_H$ such that $u \leq r^*$, then there exists a transition $t = (v_i^u, v_{i+1}^u)$ with $i \leq \lfloor n/2 \rfloor$ such that C^u is the first chain to traverse t.*

PROOF. If $u \leq \binom{n}{\lfloor n/2 \rfloor}$ then C^u is executed during state exploration and contains an MCD chain $C^u_{MCD} \in C^u$. If $u = 1$, then the statement holds trivially. Let (v_i^u, v_{i+1}^u) be the transition traversed to reach the chain starter v_{i+1} of C^u_{MCD}. Then v_{i+1}^u cannot be in the upper half, since a chain starter is either at the middle level or in the lower half. Hence $i+1 \leq \lfloor n/2 \rfloor$ implies $i < \lfloor n/2 \rfloor$ and obviously (v_i^u, v_{i+1}^u) was untraversed before C^u. If $u > \binom{n}{\lfloor n/2 \rfloor}$ then C^u is executed during the transition exploration phase. According to the transition exploration strategy, the first untraversed transition of C^u will be the one closest to the bottom among the untraversed transitions in the hypercube. Hence, unless all transitions leaving the middle level are traversed, the source of the first untraversed transition of C^u cannot be at a higher level than the middle. Since to traverse all the transitions leaving the middle level we need at least r^* chains and $u \leq r^*$, there exists $t = (v_i^u, v_{i+1}^u)$ with $i \leq \lfloor n/2 \rfloor$ such that C^u is the first chain to traverse t. □

LEMMA 6.2. *If a chain $C^u \in C_H$ enters state v_i^u using an already traversed transition and leaves v_i^u using a previously untraversed transition, then at the time C^u is executed, all transitions entering v_i^u have already been traversed.*

PROOF. Let $t = (v_{i-1}^u, v_i^u)$, $t' = (v_i^u, v_{i+1}^u)$ be the transitions that C^u traversed to enter and leave v_i^u. If $u \leq \binom{n}{\lfloor n/2 \rfloor}$ then C^u is executed during state exploration and contains an MCD chain $C^u_{MCD} \in C^u$. If t and t' are the transitions on the subpath that leads to the chain starter of C^u_{MCD} then t' is the first untraversed transition in C^u. Since the hypercube strategy tries to use unexplored transitions as much as possible, all transitions entering v_i^u must have already been traversed in this case. Note that t cannot be a transition in C^u_{MCD} as all transitions in an MCD chain are traversed for the first time. Also t cannot be a transition that is used to extend C^u_{MCD} towards the top, since only previously untraversed transitions are used for this purpose. If $u > \binom{n}{\lfloor n/2 \rfloor}$ then C^u

is executed during the transition exploration phase. Assume there is an untraversed transition (v'_{i-1}, v^u_i). Let (v^u_{j-1}, v^u_j) be the first untraversed transition in C^u. Obviously $i \neq j$. If $i < j$, then (v^u_{j-1}, v^u_j) cannot be the first previously untraversed transition in C. The transition exploration strategy would traverse (v'_{i-1}, v^u_i) first since it is closer to the bottom of the hypercube. If $i > j$, then there is no untraversed transition leaving v^u_{i-1}, otherwise the transition exploration strategy would prefer the untraversed one instead of t. But then the transition exploration strategy would not execute t at all, since the transition exploration strategy does not continue further from the current state (v^u_{i-1}) when it has no untraversed transition. □

LEMMA 6.3. *If $C^u \in C_H$ contains a transition $t = (v^u_i, v^u_{i+1})$ traversed for the first time by C^u such that $i \leq \lfloor n/2 \rfloor$, then C^u is also the first chain traversing every transition following t in C^u and the size of C^u is at least $\lfloor n/2 \rfloor + 1$.*

PROOF. As explained in Section 6.1.2, in a chain executed by the hypercube strategy, all transitions following the first unexplored transition are also unexplored. Hence, C^u is the first chain traversing all transitions after t in C^u.

Now we show that the size of C^u is at least $\lfloor n/2 \rfloor + 1$. The case $i = \lfloor n/2 \rfloor$ is trivial. Assume $i < \lfloor n/2 \rfloor$. If C^u is executed during state exploration, then there is an MCD chain $C^u_{MCD} \in C^u$. Since every MCD chain contains a state in the middle level, C^u_{MCD} also contains a state in the middle level, namely $v^u_{\lfloor n/2 \rfloor}$. If $v^u_{\lfloor n/2 \rfloor}$ has a successor $v^u_{\lfloor n/2 \rfloor+1}$ in C^u_{MCD} we will definitely reach $v^u_{\lfloor n/2 \rfloor+1}$. Otherwise, since it is the first time $v^u_{\lfloor n/2 \rfloor}$ is visited, none of its outgoing transitions is traversed before and the strategy will follow one of them. Assume C^u is executed during transition exploration. Let (v^u_j, v^u_{j+1}) be any transition traversed by C^u such that $j \geq i$. We know that (v^u_j, v^u_{j+1}) is untraversed before C^u. By Lemma 6.2, if a transition t' entering v^u_{j+1} was already traversed at the time C^u is executed, then, among all chains that traversed t', only the one that traversed t' for the first time left v^u_{j+1} using a previously untraversed transition. (If at all, others used an already traversed transition to leave v^u_{j+1}). Combining this with the fact that in a hypercube for all levels $j \leq \lfloor n/2 \rfloor$ the number of transitions leaving a state v_j at level j is greater than or equal to the number of transitions entering v_j, we can conclude that there is an untraversed transition leaving v^u_{j+1} and the transition exploration strategy will follow it. □

LEMMA 6.4. *Each transition leaving the middle level is traversed by the first r^* chains.*

PROOF. This is a consequence of Lemma 6.1 and Lemma 6.3. □

LEMMA 6.5. *Each transition leaving a state in the lower half of the hypercube is traversed by the first r^* chains.*

PROOF. Assume there exists a transition leaving a state in the lower half of the hypercube that has not been traversed after r^* chains. Let v_i be a state at level i such that $i < \lfloor n/2 \rfloor$ and (v_i, v_{i+1}) is untraversed after r^* chains. Let $C = \{v_0, v_1, \ldots, v_i, v_{i+1}, \ldots, v_k\}$ be the chain that traverses (v_i, v_{i+1}) for the first time. By Lemma 6.3, $k > \lfloor n/2 \rfloor$ and all transitions (v_j, v_{j+1}) in C where $i \leq j \leq k$ are also traversed for the first time by C. In particular, $(v_{\lfloor n/2 \rfloor}, v_{\lfloor n/2 \rfloor+1})$ was untraversed before C. But this contradicts Lemma 6.4. □

LEMMA 6.6. *Each transition entering a state in the upper half of the hypercube is traversed by the first r^* chains.*

PROOF. In a hypercube, the number of transitions entering a state in the upper half is greater than the number of transitions leaving it. So each state in the upper half must be visited more than the number of outgoing transitions. The hypercube strategy continues to traverse untraversed transitions as long as the current state has one. That means, if all incoming transitions of an upper-level state are traversed then all outgoing transitions are also traversed by the strategy. By Lemma 6.4, every transition entering the level $\lfloor n/2 \rfloor + 1$ (first level of the upper half) is traversed by r^* chains. The result extends similarly for the higher levels. □

THEOREM 6.7. *Every transition in the hypercube is traversed by the first r^* chains of the hypercube strategy.*

PROOF. Follows from Lemma 6.5 and Lemma 6.6. □

6.5.2. Number of Event Executions. The hypercube strategy is also optimal in terms of the total number of events executed to crawl an application whose model is a hypercube. The optimal number of event executions required to crawl a hypercube of dimension $n \geq 2$ is $e^* = 2^{n-2}n + r^*n/2$. For simplicity assume n is even, then this formula can be derived as follows (the number is also valid for odd n). As we have shown, at least r^* chains are needed to crawl the hypercube and each of these chains has to cover a transition leaving a middle-level state. As a result, some transitions in the lower half must be traversed more than once. Since there are r^* chains to execute, for $1 \leq i \leq n/2$ the total number of transition traversals for transitions entering level i is r^*. So, for all transitions whose source is in the lower half, $r^*n/2$ event executions are needed. Each transition whose source is in the middle level or the upper half needs to be executed at least once. There are $2^{n-2}n$ such transitions. So in total, the optimal number of event executions to crawl the hypercube is $e^* = 2^{n-2}n + r^*n/2$. It is not difficult to see that the hypercube strategy uses the optimal number of event executions as stated by the following.

THEOREM 6.8. *The hypercube strategy executes exactly e^* events when crawling a hypercube application.*

PROOF. We have already shown that the hypercube strategy uses the optimal number of chains, which is r^*. So, in the lower half of the hypercube no more than $r^*n/2$ events will be executed. Then it is enough to show that each transition leaving a state at level $\lfloor n/2 \rfloor \leq i \leq n$ is traversed exactly once. By Lemma 6.1 and Lemma 6.3, each chain $C^u \in C_H$ is the first chain to traverse some $t = (v_i^u, v_{i+1}^u)$ with $i \leq \lfloor n/2 \rfloor$ and every transition after t is also traversed for the first time by C^u, including all transitions traversed by the hypercube strategy in the upper half. Hence, there is no chain that retraverses a transition in the upper half. Since we have also shown that the hypercube strategy covers all transitions, each transition in the upper half is traversed exactly once. □

7. PERFORMANCE EVALUATION

In this section, we present the results of our experimental study using five real and three test applications. The research questions for this study are as follows.

—Can a model-based strategy be more efficient than the standard crawling strategies for crawling RIAs?
—How does the hypercube strategy perform when the application being crawled does not follow the hypercube metamodel?

7.1. Measuring the Efficiency of a Strategy

We defined the efficiency of a strategy in terms of the time the strategy needs to discover all the states of the application. Although we provide the time measurements for this experimental study, our preferred way of assessing efficiency is to measure the number of events executed and the resets used by a strategy during the crawl. The reason is that the time measurements also depend on factors external to the crawling strategy, such as the hardware used to run the experiments and the network delays, which can be different in different runs. In addition, the event executions and resets are normally the operations that dominate the crawling time and they only depend on the decisions of the strategy. We combine the number of resets and the event executions used by a strategy to define a cost unit as follows.

—We measure for each application:
  —$t(e)_{avg}$: the average event execution time. This is obtained by measuring the time for executing each event in a randomly selected set of events in the application and taking the average.
  —$t(r)_{avg}$: the average time to perform a reset.
—For simplicity, we consider each event execution to take $t(e)_{avg}$ and take this as a cost unit.
—We calculate "the cost of reset": $c_r = t(r)avg/t(e)avg$.
—Finally, the cost that is spent by a strategy to find all the states of an application is calculated by

$$n_e + n_r \times c_r,$$

where n_e and n_r are the total number of events executed and resets used by a strategy to find all the states, respectively[7].

7.2. Strategies Used for Comparison and the Optimal Cost

We compare the hypercube strategy as well as the other two model-based crawling strategies, the menu and the probability, with the following.

—*The Optimized Breadth-First and the Optimized Depth-First Strategies.* These are the standard crawling strategies that are widely used. The breadth-first strategy explores the least recently discovered state first. The depth-first strategy explores the most recently discovered state first. Our implementations for the breadth-first and the depth-first methods are optimized so that (like the other strategies), to reach the state where an event will be explored, the shortest known transfer sequence from the current state is used during the crawl. The "default" versions of the breadth-first and the depth-first strategies (simply resetting to reach a state) fare much worse than the results presented here.
—*The Greedy Strategy.* This is a simple strategy that prefers to explore any event from the current state if there is an unexplored event. Otherwise, it explores any event from any state that is closest to the current state.
—*The Optimal Cost to Discover All States.* We also present, for each application, the optimal cost required to discover all the states of the application. The optimal cost can only be calculated after the model of the application is obtained using one of the crawling strategies. The optimal cost can be calculated by finding an optimal

[7]We measure the value of c_r before crawling an application and give this value as a parameter to each strategy. A strategy, knowing how costly a reset is compared to an average event execution, can decide whether to reset or not when transferring from the current state to another known state. Although our measurements on the test applications show that the cost of reset is greater than 1, this does not mean it cannot be less than or equal to 1.

path that visits all nodes (states) of the known directed graph (model), known as the Asymmetric Traveling Salesman Problem (ATSP). We use an exact ATSP solver [Carpaneto et al. 1995] to get an optimal path. Note that this optimal path cannot be known before crawling, hence it is not a crawling strategy. This is provided just to see how far the strategies are from an optimal solution.

7.3. Subject Applications

We compare the strategies using three test and five real RIAs. We acknowledge that the number of applications is not very high, however, we are limited by the availability of tools that can be used for RIA crawling. For crawling an RIA, a tool must have the complete control of the JavaScript execution. The existing tools that provide such level of control are mostly experimental and typically implement a small subset of JavaScript. Thus, often each new RIA requires a significant amount of work to improve existing tools by implementing the JavaScript functionality required by each new application. This work, unrelated to our research focus, prevents us from using a large set of applications for experiments.

In addition, each application crawled is first mirrored on our Web site (the mirrored version is the one crawled), to ensure that we are able to again crawl the same version and to avoid stressing the real site with a large number of automated requests. This also prevents too much variation due to network delay.

The applications we have used in this experimental study are the following[8].

(1) *Bebop*. This is an AJAX-based bibliography browsing application. It allows filtering a bibliography according to different categories (authors, year, etc.) and displaying details about each publication in the bibliography.
(2) *Elfinder*. This is an AJAX-based file management application. The version we have used allows browsing a file system and displaying a preview for each directory/file.
(3) *FileTree*. This is an AJAX-based file explorer, an application that allows navigating a directory structure on the Web server.
(4) *Periodic Table*. This is an AJAX-based periodic table. When a chemical element is clicked in the table, detailed information about the chemical element is displayed.
(5) *Clipmarks*. This is an AJAX-based social network to easily share multimedia (text, video, images) found on the other Web sites with the users of the network. We have used a partial local copy of this Web site for the experimental study.
(6) *Altoro Mutual*. This is an AJAX version of a demo Web site by IBM® Security AppScan® Team. It is a fictional banking site.
(7) *TestRIA*. This is an AJAX test application we developed that is in the form of a generic company site.
(8) *Hypercube10D*. This is a test application whose model is a 10-dimensional hypercube.

Except for the Hypercube10D, the models of the applications are completely unrelated to the hypercube metamodel (i.e., they do not follow the hypercube metamodel). Table I summarizes the number of states (#s), the number of transitions (#t) (both obtained after crawling), and the cost of reset (c_r) according to our measurements.

7.4. Experimental Setup

We have implemented all the mentioned crawling strategies in a prototype of IBM® Security AppScan® Enterprise, a security scanner for Web applications[9]. Each strategy is implemented as a separate class in the same codebase. That is, each strategy uses

[8]Links to the applications can be found at http://ssrg.eecs.uottawa.ca/testbeds.html.
[9]The implementation details are available at http://ssrg.eecs.uottawa.ca/docs/prototype.pdf.

Table I. Applications

	Name	$\#s$	$\#t$	c_r
Real Applications	Bebop	1,800	145,811	2
	Elfinder	1,360	43,816	10
	FileTree	214	8,428	2
	Periodic Table	240	29,034	8
	Clipmarks	129	10,580	18
Test Applications	TestRIA	39	305	2
	Altoro Mutual	45	1,210	2
	Hypercube10D	1,024	5,120	3

the same implementation of the DOM equivalence relation, an event identification mechanism, the same embedded browser (JavaScript execution and DOM manipulation engine), and so on. For this reason, at the end of the crawl each strategy discovers the same model for an application (this is also verified by comparing the models produced by each strategy against each other). The sole difference is the decisions the strategies make to discover this model.

Although the majority of the events in our subject applications are user interaction events, some applications, for example, Elfinder, have timeout events as well. Our crawler executes any timeout events in addition to the event triggered by user interactions. After each event execution (or page reload), the crawler checks whether there are timeout events registered as a result of the event execution (page load). The crawler executes all such timeout events, so that we reach a stable DOM after each event execution.

To verify that the model extracted by our tool is a correct model of the application behavior, we manually checked the states and transitions in the model against the observed behavior of the application in an actual browser. To ease this verification process, we have developed a model visualization tool that can represent the extracted model as a directed graph (shown in Figure 4). It also allows us to replay in an actual browser the transitions the crawler traversed to reach any state.

7.5. Results

For each application and for each strategy, we present two sets of measurements:

(1) how fast the strategy discovers the states of the application (i.e., strategy efficiency),
(2) how fast the strategy finishes crawling (i.e., discovers all the transitions of the application).

Among these two, we are primarily interested in the first. In our definition, the first set of measurements shows how efficient the strategy is for an application. However, we have to crawl each application completely to obtain the first set of measurements, since the crawler cannot know for sure that all the states are discovered until all transitions are taken at least once.

7.5.1. Strategy Efficiency. We use box plots to present the results for the strategy efficiency. The box plots in Figure 5 show the cost measurements (as defined in Section 7.1) and the box plots in Figure 6 show the time measurements (in seconds). Each figure contains a box plot for each (application, strategy) pair. A box plot consists of a line and a box on the line. The minimum point of the line shows the cost (or time) of discovering the first state (equal to c_r for the plots in Figure 5). The lower edge, the line in the middle, and the higher edge of the box show the cost (or time) of discovering 25%, 50%, and 75% of the states, respectively. The maximum point of the line shows the cost (or time) of discovering all the states. The plots are drawn in logarithmic scale for better visualization.

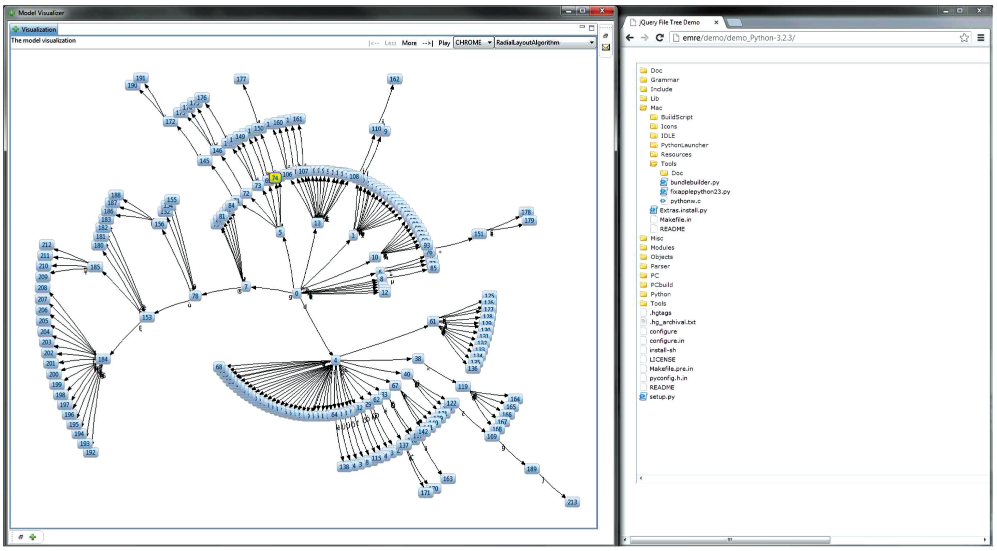

Fig. 4. On the left, the extracted model of the FileTree partially shown in the visualization tool (the tool is configured not to show all the transitions for a clearer picture) and a state that is selected to be shown in the browser. On the right, the page corresponding to the selected state, reached automatically by replaying the event executions taken by the crawler in a real browser.

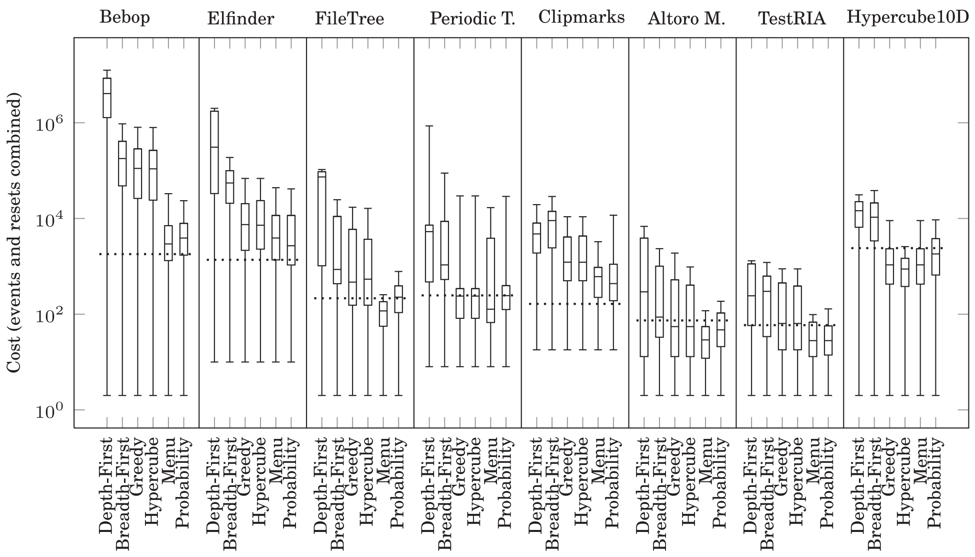

Fig. 5. Costs of discovering the states (strategy efficiency), in logarithmic scale. The cost measure combines the number of events executed and the number of resets used by the strategy during the crawl; see Section 7.1.

In Figure 5, the costs of discovering states are shown. The vertical dotted lines in this figure shows the optimal cost of discovering all the states for an application. It can be seen that for all applications the hypercube strategy is significantly more efficient than the breadth-first and the depth-first strategies. It is also evident that the hypercube strategy shows a similar performance to the greedy strategy when the application does

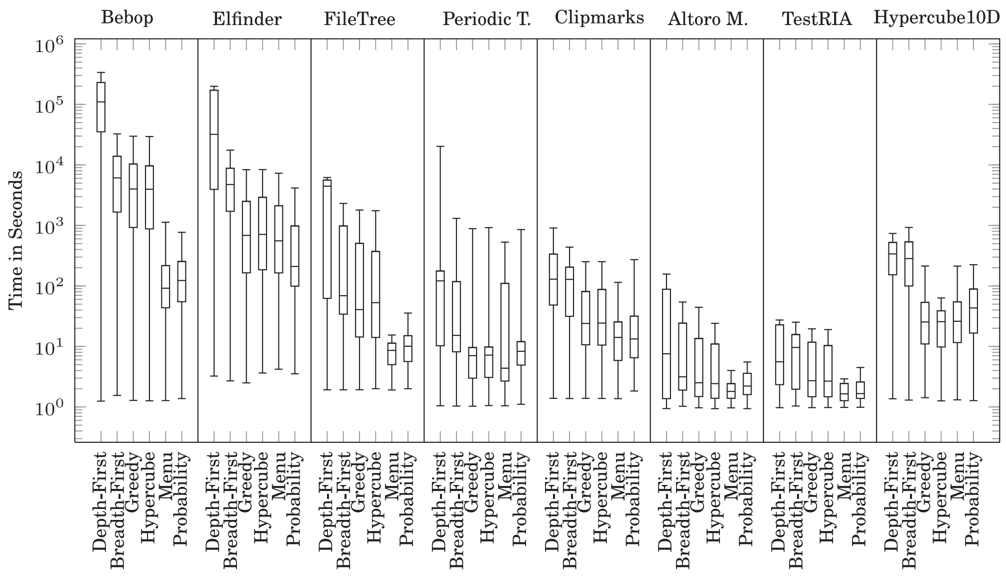

Fig. 6. Time for discovering the states (strategy efficiency), in logarithmic scale.

not follow the hypercube metamodel (all applications except the Hypercube10D). The other model-based crawling strategies, the menu and the probability, are even more efficient.

The results are not surprising, given the fact that the hypercube strategy always tries to explore an event from a state that is closer to the current state when the application does not follow the hypercube metamodel. This means the hypercube strategy falls back to a greedy strategy when the application does not follow the metamodel. However, there is a slight difference between the greedy strategy and the hypercube. Among two states that are at the same distance from the current state, the hypercube strategy prefers to explore an event from the state that has the larger number of events (in the same case, the greedy strategy has no preference). This is because, according to the hypercube assumptions, the state with more enabled events has a larger anticipated model, which means more chances of discovering a new state. This explains the slight performance differences that can be seen between the hypercube and the greedy. It is worth noting that, although the application Altoro Mutual does not follow the hypercube metamodel, this preference makes the hypercube strategy slightly better for the application. Of course, the Hypercube strategy guarantees an optimal crawling when the application follows the hypercube metamodel, as can be seen in the case of Hypercube10D.

There is, however, a more important difference between the greedy strategy and the hypercube strategy: the hypercube strategy is usually slightly better, and at worst as efficient as the greedy strategy. But the hypercube strategy is a good illustration of a more general approach to model building, namely model-based crawling. It opens up the possibility to improve upon these results using other model-based strategies, such as the menu and the probability, which are based on the same principles. The greedy strategy, on the other hand, does not suggest new principles that could be leveraged to improve the results.

In Figure 6, the time measurements are presented. The time measurements show very similar results and verify that the event executions and resets indeed dominate the crawling time. Hence, the number of events and the number of resets are valid metrics to assess the performance of a crawling strategy. The major source of difference between the time measurements and our cost measurements is the simplifying assumption we are making (when calculating the cost) that each event execution takes the same average time. As a result of this simplification, for Clipmarks, our measured value for cost of reset (c_r) seems a bit higher than the real time delays incurred by the resets during crawling. This is why, for this application, the breadth-first method is punished too much for using a significantly larger number of resets when calculating its cost.

We also present the exact numbers for when all the states are discovered. Table II shows the number of events executed, the number of resets used, the cost (calculated as explained in Section 7.1), and the time it takes for each strategy to find all the states of an application (i.e., the data in this table corresponds to the maximum points on the box plots). The numbers show that the breadth-first strategy needs significantly more resets whereas the depth-first one executes significantly more events.

7.5.2. Completing the Crawl. The previous results show the measurements until all the states are discovered by the strategies. However, the crawl usually does not end at this point, since the crawler does not know that all the states are discovered. The crawler has to first extract the complete model (explore all transitions), before it can conclude that all states are discovered. In this section, we present the measurements when the complete model is extracted. As discussed earlier, this data has no significance in our definition of crawling strategy efficiency.

The results presented in Table III show the number of events executed, the number of resets used, the cost (calculated as explained in Section 7.1), and the time it takes for each strategy to crawl the applications. It can be seen that two of the model-based strategies, the hypercube and the probability, as well as the greedy, finish crawling significantly faster than the standard crawling strategies. The menu strategy is also more efficient in terms of event executions and resets, but in terms of time, the menu strategy takes slightly more time than the breadth-first version for Bebop and Elfinder. This is because the transition exploration phase for the menu strategy is not implemented in the most optimized way.

8. CONCLUSION AND FUTURE WORK

The main contributions of this article can be summarized as follows.

—We provide a discussion of the important concepts and challenges for crawling of RIAs.
—We present model-based crawling as a general approach to design efficient crawling strategies for RIAs.
—We explain the hypercube metamodel and its corresponding strategy as an example strategy designed using model-based crawling methodology. Also, proof of optimality of this strategy for applications that follow the hypercube metamodel is given.
—We detail an experimental study where the performances of the model-based strategies, the hypercube, the menu, and the probability, are compared with existing crawling strategies on five real RIAs as well as three test applications.

The results we have obtained show that more efficient crawling strategies are possible using a model-based crawling method. Although the hypercube strategy is a good example to show how the model-based crawling works and has a good performance compared to the existing strategies used so far in the related works, the experimental study made it evident to us that the hypercube assumptions are too strict for most real RIAs; they are violated too often. The model-based crawling strategies designed after

Table II. Measurements for Strategy Efficiency

Application	Type	Depth-First	Breadth-First	Greedy	Hypercube	Menu	Probability	Optimal
Bebop	Events	12,542,684	933,696	803,966	796,356	32,960	23,451	1,799
	Resets	1	8,681	27	27	1	27	1
	Cost	12,542,686	951,058	804,020	796,410	32,962	23,505	1,781
	Time (HH:MM:SS)	93:42:14	09:00:31	08:17:07	08:10:36	00:18:49	00:12:50	
Elfinder	Events	1,999,241	116,562	66,806	66,886	41,915	39,526	1,359
	Resets	199	7,178	185	184	201	200	1
	Cost	2,001,231	188,342	68,656	68,726	43,925	41,526	1,369
	Time (HH:MM:SS)	55:15:56	04:42:47	02:18:56	02:19:29	02:01:29	01:09:14	
FileTree	Events	105,343	21,654	17,134	16,243	253	779	213
	Resets	1	1495	13	13	1	1	1
	Cost	105,345	24,644	17,160	16,269	255	781	215
	Time (HH:MM:SS)	01:43:06	00:38:29	00:29:57	00:29:10	00:00:15	00:00:36	
Periodic Table	Events	860,312	28,826	29,574	29,574	16,846	28,904	239
	Resets	7	7,495	7	7	4	2	1
	Cost	860,368	88,786	29,630	29,630	16,878	28,920	247
	Time (HH:MM:SS)	05:37:45	00:21:48	00:14:44	00:15:20	00:08:49	00:14:15	
Clipmarks	Events	18,617	12,608	10,748	10,732	3,106	11,114	128
	Resets	46	900	7	7	9	33	2
	Cost	19,445	28,808	10,874	10,858	3,268	11,708	132
	Time (HH:MM:SS)	00:14:44	00:07:17	00:04:11	00:04:12	00:01:55	00:04:32	
Altoro Mutual	Events	6,790	1,635	1,842	906	104	172	72
	Resets	25	352	25	32	7	7	1
	Cost	6,840	2,339	1,892	970	118	186	74
	Time (HH:MM:SS)	00:02:37	00:00:54	00:00:44	00:00:24	00:00:04	00:00:06	
TestRIA	Events	1,302	1,097	884	882	96	127	57
	Resets	1	54	1	1	1	1	1
	Cost	1,304	1,205	886	884	98	129	59
	Time (HH:MM:SS)	00:00:27	00:00:25	00:00:20	00:00:19	00:00:03	00:00:04	
Hypercube10D	Events*	23,033	28,070	7,093	2,077	7,093	7,335	1,646
	Resets	4,090	5,111	972	252	972	1,012	252
	Cost	31,213	38,292	9,037	2,581	9,037	9,359	2,402
	Time (HH:MM:SS)	00:11:59	00:15:05	00:03:34	00:01:03	00:03:33	00:03:44	

The number of events executed, the number of resets used, the cost (as defined in Section 7.1), and the time (in hours:minutes:seconds) measurements for discovering all the states.

*The difference in number of events between the hypercube and the optimal is because our implementation keeps exploring events when the end of an MCD chain is reached, rather than immediately resetting and following another MCD chain. If this is not done, these numbers would be the same as explained in Section 6.5.

Table III. Measurements for Complete Crawl

Application	Type	Depth-First	Breadth-First	Greedy	Hypercube	Menu	Probability
Bebop	Events	13,006,646	943,035	827,830	816,187	814,148	817,044
	Resets	27	8,686	27	27	27	27
	Cost	13,006,700	960,407	827,884	816,241	814,202	817,098
	Time (HH:MM:SS)	97:05:22	09:04:57	08:32:04	08:22:54	12:48:55	07:54:03
Elfinder	Events	2,000,239	121,735	68,534	68,481	75,346	68,152
	Resets	209	7,258	209	209	213	278
	Cost	2,002,329	194,315	70,624	70,571	77,476	70,932
	Time (HH:MM:SS)	55:17:30	04:53:40	02:21:47	02:22:05	08:23:26	02:03:43
FileTree	Events	105,645	26,370	20,755	19,887	19,741	19,360
	Resets	13	1,638	13	13	13	13
	Cost	105,671	29,646	20,781	19,913	19,767	19,386
	Time (HH:MM:SS)	01:43:28	00:43:05	00:33:42	00:33:02	00:33:40	00:31:37
Periodic Table	Events	860,661	64,853	29,923	29,923	41,082	29,525
	Resets	236	14,634	236	236	236	236
	Cost	862,549	181,925	31,811	31,811	42,970	31,413
	Time (HH:MM:SS)	05:38:02	00:42:19	00:15:06	00:15:42	00:20:06	00:14:42
Clipmarks	Events	18,799	15,327	11,390	11,363	11,623	11,492
	Resets	71	925	55	56	69	62
	Cost	20,077	31,977	12,380	12,371	12,865	12,608
	Time (HH:MM:SS)	00:14:52	00:08:14	00:04:28	00:04:29	00:04:22	00:04:39
Altoro Mutual	Events	6,802	3,076	2,521	2,490	2,468	2,452
	Resets	34	353	34	34	34	35
	Cost	6,870	3,782	2,589	2,558	2,536	2,522
	Time (HH:MM:SS)	00:02:38	00:01:25	00:01:00	00:01:00	00:01:00	00:00:59
TestRIA	Events	1,460	1,229	1,006	995	974	973
	Resets	1	55	1	1	1	1
	Cost	1,462	1,339	1,008	997	976	975
	Time (HH:MM:SS)	00:00:34	00:00:27	00:00:21	00:00:21	00:00:21	00:00:21
Hypercube10D	Events	23,050	28,160	8,865	8,860	8,865	9,434
	Resets	4,098	5,120	1,260	1,260	1,260	1,364
	Cost	35,344	43,520	12,645	12,640	12,645	13,526
	Time (HH:MM:SS)	00:12:00	00:15:07	00:04:28	00:04:19	00:04:24	00:04:46

The number of events executed, the number of resets used, the cost (as defined in Section 7.1), and the time (in hours:minutes:seconds) measurements for complete crawls.

the hypercube model, the menu and the probability, have more realistic assumptions. Thus, they are much more efficient for real application than any other known crawling strategy. However, there are still many opportunities for improvement. We conclude the article with a discussion of some future research directions.

8.1. Adaptive Model-Based Crawling

One important aspect of model-based crawling is to decide on a metamodel for which the crawling strategy will be optimized. However, it is often difficult to predict a good metamodel for an arbitrary application before crawling. A possible solution to this problem might be using an adaptive model-based crawling approach. Instead of fixing a metamodel before crawling, the crawler could start exploring the application using a generic and relatively efficient strategy. Once some initial information is collected using this strategy, the partially extracted model could be analyzed by the crawler and a model-based strategy that would suit the application could be chosen. However, this requires to have a set of metamodels available. The hypercube strategy we have presented is a first step towards this end and it has been followed by the probability [Dincturk et al. 2012] and the menu [Choudhary et al. 2013] strategies.

This idea of dynamically choosing the metamodel can even be developed further, so that a suitable metamodel could be constructed during the crawl. This would be possible when the model of the application has some repeating patterns. For example, we might detect that the instances of the same subgraph repeat themselves in the partially extracted model (as an example, we can think of a large application that uses the same navigational pattern to present different content, like a Web-based product catalog). In that case, it could even be possible to generate an optimal strategy for such subgraphs. Whenever the crawler can predict that some portion of the application is likely to follow this same structure, we can apply this dynamically generated, optimized strategy for exploring that portion.

A recent paper [Faheem and Senellart 2013] applies the adaptive crawling idea to Web sites generated using content management systems like WordPress, vBulletin, etc. When crawling the Web, if they recognize that a Web site is following the template (metamodel) of one of these systems then they apply a crawling strategy optimized for that template.

8.2. Avoiding New States without New Information

A Web page in an RIA often consists of portions that can be interacted independently. For example, each widget in a widget-based application, or each container that mimics the functionality of a window of a desktop application, usually behave independent of each other. Normally, every different combination of the contents of such independent parts will be considered as a new state. However, the majority of such states will not contain any new information. To be able to efficiently crawl these kind of applications, the crawler should be able to predict those events that will lead to new states without new information and assign them lower priorities. The key to achieve this is the ability to detect such independent parts and crawl them separately.

8.3. Greater Diversity

For a large RIA, it is not feasible to wait until crawling finishes to analyze the pages discovered. The analysis of the discovered pages usually takes place while crawling still continues. This is why we stress the importance of the ability to find new pages as soon as possible. In addition to that, rather than exploring one part of the application exhaustively only to keep discovering new but very similar pages, we would like discover dissimilar pages as much as possible earlier on during the crawl. For example, consider a Web page that has a long list of events where each event leads to a similar page. It is not smart to first exhaustively explore each of these events, before trying something outside this list. This is true especially for testing, since the similar pages would probably have the same problems. It is not of much use to find a thousand instances of the same problem when finding one of them would suffice to fix all instances. For this reason, new techniques are needed that would diversify the crawling and provide a bird's-eye view of the application as soon as possible. To this end, crawling strategies may benefit from algorithms that will help in detecting similar pages, and events with similar functionality.

8.4. Relaxing the Determinism Assumption

Another common limitation of the current RIA crawling approaches is the determinism assumption, that is, the expectation that an event will lead to the same state whenever it is executed from a given state. This is not very realistic since most real Web applications may react differently at different times. Crawling strategies should be improved in order to handle such cases.

8.5. Distributed Crawling

Another promising research area is to crawl RIAs using multiple concurrently running processes to reduce the crawling time. The existing distributed crawling techniques for traditional applications distribute the workload based on URLs. However, this would not be sufficient in the context of RIA crawling, so new distributed crawling algorithms are required for RIAs.

REFERENCES

M. Aigner. 1973. Lexicographic matching in boolean algebras. *J. Combin. Theory* 14, 3, 187–194.

D. Amalfitano, A. R. Fasolino, and P. Tramontana. 2008. Reverse engineering finite state machines from rich Internet applications. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE'08).* IEEE Computer Society, 69–73.

D. Amalfitano, A. R. Fasolino, and P. Tramontana. 2010. Rich Internet application testing using execution trace data. In *Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation Workshops (ICSTW'10).* IEEE Computer Society, 274–283.

I. Anderson. 1987. *Combinatorics of Finite Sets*. Oxford University Press, London.

Apache. 2004. Apache flex. http://incubator.apache.org/flex/.

A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan. 2001. Searching the web. *ACM Trans. Internet Technol.* 1, 1, 2–43.

J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. 2010. State of the art: Automated black-box web application vulnerability testing. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'10).* IEEE Computer Society, 332–345.

K. Benjamin. 2010. A strategy for efficient crawling of rich Internet applications. M.S. thesis, EECS - University of Ottawa. http://ssrg.eecs.uottawa.ca/docs/Benjamin-Thesis.pdf.

K. Benjamin, G. V. Bochmann, G.-V. Jourdan, and I.-V. Onut. 2010. Some modeling challenges when testing rich Internet applications for security. In *Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation Workshops (ICSTW'10).* IEEE Computer Society, 403–409.

K. Benjamin, G. Von Bochmann, M. E. Dincturk, G.-V. Jourdan, and I. V. Onut. 2011. A strategy for efficient crawling of rich Internet applications. In *Proceedings of the 11th International Conference on Web Engineering (ICWE'11).* Springer, 74–89.

C.-P. Bezemer, A. Mesbah, and A. Van Deursen. 2009. Automated security testing of web widget interactions. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'09).* ACM Press, New York, 81–90.

N. Bruijn, C. Tengbergen, and D. Kruyswijk. 1951. On the set of divisors of a number. *Nieuw Arch. Wisk.* 23, 191–194.

G. Carpaneto, M. Dellamico, and P. Toth. 1995. Exact solution of large-scale, asymmetric traveling salesman problems. *ACM Trans. Math. Softw.* 21, 4, 394–409.

J. Cho and H. Garcia-Molina. 2003. Estimating frequency of change. *ACM Trans. Internet Technol.* 3, 3, 256–290.

S. Choudhary. 2012. M-crawler: Crawling rich Internet applications using menu meta-model. M.S. thesis, EECS - University of Ottawa. http://ssrg.site.uottawa.ca/docs/Surya-Thesis.pdf.

S. Choudhary, M. E. Dincturk, G. V. Bochmann, G.-V. Jourdan, I. V. Onut, and P. Ionescu. 2012. Solving some modeling challenges when testing rich Internet applications for security. In *Proceedings of the International Conference on Software Testing, Verification, and Validation*. 850–857.

S. Choudhary, M. E. Dincturk, S. M. Mirtaheri, G.-V. Jourdan, G. Bochmann, and I.-V. Onut. 2013. Building rich Internet applications models: Example of a better strategy. In *Proceedings of the 13th International Conference on Web Engineering (ICWE'13).* Lecture Notes in Computer Science, vol. 7977, Springer, 291–305.

E. G. Coffman, Z. Liu, and R. R. Weber. 1998. Optimal robot scheduling for web search engines. *J. Schedul.* 1, 1, 15–29.

R. P. Dilworth. 1950. A decomposition theorem for partially ordered sets. *Ann. Math.* 51, 1, 161–166.

M. E. Dincturk. 2013. Model-based crawling - An approach to design efficient crawling strategies for rich Internet applications. Ph.D. thesis, EECS - University of Ottawa. http://ssrg.site.uottawa.ca/docs/Dincturk_MustafaEmre_2013_thesis.pdf.

M. E. Dincturk, S. Choudhary, G. Bochmann, G.-V. Jourdan, and I. V. Onut. 2012. A statistical approach for efficient crawling of rich Internet applications. In *Proceedings of the 12th International Conference on Web Engineering (ICWE'12).* Springer, 74–89.

C. Duda, G. Frey, D. Kossmann, R. Matter, and C. Zhou. 2009. Ajax crawl: Making Ajax applications searchable. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'09).* IEEE Computer Society, 78–89.

M. Faheem and P. Senellart. 2013. Intelligent and adaptive crawling of web applications for web archiving. In *Proceedings of the 13th International Conference on Web Engineering (ICWE'13).* F. Daniel, P. Dolog, and Q. Li, Eds., Lecture Notes in Computer Science, vol. 7977, Springer, 306–322.

G. Frey. 2007. Indexing Ajax web applications. M.S. thesis, ETH Zurich. http://e-collection.library.ethz.ch/eserv/eth:30111/eth-30111-01.pdf.

J. J. Garrett. 2005. Ajax: A new approach to web applications. http://www.adaptivepath.com/publications/essays/archives/000385.php.

Google. 2009. Making Ajax applications crawlable. http://code.google.com/web/ajaxcrawling/index.html.

C. Greene and D. J. Kleitman. 1976. Strong versions of Sperner's theorem. *J. Combin. Theory A20,* 1, 80–88.

J. Griggs, C. E. Killian, and C. Savage. 2004. Venn diagrams and symmetric chain decompositions in the boolean lattice. *Electron. J. Combin.* 11, 2.

J. Lu, Y. Wang, J. Liang, J. Chen, and J. Liu. 2008. An approach to deep web crawling by sampling. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT'08),* Vol. 1. 718–724.

A. Mesbah, E. Bozdag, and A. V. Deursen. 2008. Crawling Ajax by inferring user interface state changes. In *Proceedings of the 8th International Conference on Web Engineering (ICWE'08).* IEEE Computer Society, 122–134.

A. Mesbah and A. Van Deursen. 2009. Invariant-based automatic testing of Ajax user interfaces. In *Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE'09).* 210–220.

A. Mesbah, A. Van Deursen, and S. Lenselink. 2012. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Trans. Web* 6, 1.

Microsoft. 2007. Silverlight. http://www.microsoft.com/silverlight/.

A. Ntoulas, P. Zerfos, and J. Cho. 2005. Downloading textual hidden web content through keyword queries. In *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL'05).* ACM Press, New York, 100–109.

C. Olston and M. Najork. 2010. Web crawling. *Found. Trends Inf. Retr.* 4, 3, 175–246.

L. Page, S. Brin, R. Motwani, and T. Winograd. 1998. The PageRank citation ranking: Bringing order to the web. Tech. rep., Standford University. http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf.

Z. Peng, N. He, C. Jiang, Z. Li, L. Xu, Y. Li, and Y. Ren. 2012. Graph-based Ajax crawl: Mining data from rich Internet applications. In *Proceedings of the International Conference on Computer Science and Electronics Engineering (ICCSEE'12).* Vol. 3, 590–594.

D. Roest, A. Mesbah, and A. Van Deursen. 2010. Regression testing Ajax applications: Coping with dynamism. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'10).* IEEE Computer Society, 127–136.

W3C. 2005. Document object model (dom). http://www.w3.org/DOM/.

P. Wu, J.-R. Wen, H. Liu, and W.-Y. Ma. 2006. Query selection techniques for efficient crawling of structured web sources. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06).* IEEE Computer Society, 47.

Received May 2012; revised August 2013; accepted December 2013