Verifying Interactive Web Programs

by
Daniel R. Licata

A Thesis submitted in partial fulfillment of the requirements for Honors
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2004

© Copyright 2004 by Daniel R. Licata

This thesis by Daniel R. Licata is accepted in its present form by
the Department of Computer Science as satisfying the research requirement
for the awardment of Honors.

Date ______________ ______________________________
Shriram Krishnamurthi, Reader

Date ______________ ______________________________
Philip N. Klein, Reader

Acknowledgements

I would like to thank my advisor, Shriram Krishnamurthi, my reader, Philip Klein, and the Brown Computer Science Department.

Abstract

Web programs are important, increasingly representing the primary public interfaces of commercial organizations. Unfortunately, Web programs also exhibit numerous flaws. In addition to the usual correctness problems faced by software, Web programs must contend with numerous subtle user operations such as clicking the Back button or cloning and submitting a page multiple times. Many existing Web verification tools fail to even consider, much less effectively handle, these operations.

This paper describes a model checker designed to identify errors in Web software. We present a technique for automatically generating novel models of Web programs from their source code; these models include the additional control flow enabled by these user operations. In this technique, we exploit a constraint-based approach to avoid overapproximating this control flow; this approach allows us to evade exploding the size of the model. Further, we present a powerful base property language that permits specification of useful Web properties, along with several property idioms that simplify specification of the most common Web properties. Finally, we discuss the implementation of this model checker and a study of its effectiveness.

Contents

⋆ Parts of this text have been submitted in a conference paper co-authored with Shriram Krishnamurthi.

Chapter 1

Introduction

The interactive Web is here to stay. Not only are Web sites generated by programs, but they are increasingly playing the role of "services", accepting inputs from users, combining these with information in databases, and dynamically computing results. Indeed, from most users' perspectives, a corporation such as Amazon.com or eBay *is* a Web site: the browser is their principal, often only, means of interacting with the organization. As the number of such organizations increases, the robustness of Web software takes on increasing importance.

Web applications operate in a world of complex user operations. Users can click the Back button, or clone a window and submit a request from each clone. The Back button forces the computation to resume at a prior interaction point; submitting multiple clones causes computation at the same interaction point to resume multiple times. Worse, these user operations are *silent*: they occur in the browser only, and are not reported to the Web application.

The consequence of these complex and silent user operations is that Web programs manifest numerous subtle errors; Graunke et al. have outlined some of these [10]. For instance, travel sites reserve the wrong flights or hotels. Furthermore, programmers who have not considered a sequence of operations are likely neither to develop defensively against it, nor to subsequently test for it. User experience demonstrates that even the professionally-developed Web sites of commercially successful companies are not immune to these errors.

User operations are prevalent: (somewhat outdated) studies have shown that the Back button accounts for a significant percentage of user actions [4]. Even attempts to program defensively against them may go awry. A recent New York Times article [20] describes such a situation:

> But when I clicked on the National [car rental] price[. . .], the site responded with this message: "You may have back-buttoned too far." This was my first experience with "back-button" as a verb. I first translated the phrase as, "You may have pushed the back button too many times." Since that was patently untrue, I decoded its true meaning: "We ran out."

In short, any verification tool for the Web that does not account for user operations is not only incomplete, but potentially even misleading.

In this work, we present a verification technique for Web software that does account for user operations; designing this technique has produced several significant technical contributions. First, we have designed a Web-aware control-flow analysis that generates a model of a Web program from its source code; this model captures the control flow engendered by user operations. Secondly, we have developed a powerful data-flow-analysis-based property language useful for specifying Web properties, along with several property idioms that simplify specification of the most common Web properties. Finally, we have specialized a model checker with Web domain knowledge for precise verification.

Chapter 2

Motivation and Foundations

We begin by presenting several typical Web program properties that we would like to verify; these properties drive our choice of a particular verification approach.

2.1 Example Properties

A sequence of user operations that exposes an actual bug in the flight-reservation program of Orbitz.com (a travel website) is described in Figure 2.1. The *Orbitz property* asserts the absence of this bug: the flight described on the page that the user submits in Step 6 (which we will call the `flight-displayed`) should be the same as the actual flight for which his reservation is made (the `flight-reserved`). The Orbitz bug was documented by Graunke et al. [10]; at the time of this writing, the Orbitz developers seem to have programmed defensively against it by disallowing the user from reserving any flight at all when he returns to the page displaying Flight A and submits it. That is, they have dealt with the complexities of user operations by restricting the user's allowed behavior—not an especially satisfying solution.

One might conclude from this example that all Web sites should have something like the Orbitz property—that the data used for computation should always correspond to what the user saw on the last page he submitted. However, sometimes it is more desirable to have the *Amazon property*, which is drawn from a desired property of Amazon.com: once the user selects a book to purchase, it should be contained in his shopping cart. In particular, the user should be able to select books in two different browser windows and have both appear in his cart—but this means that the cart will not also satisfy an analogue of the Orbitz property.

Finally, the *password-page property* prescribes that an authentication page should always be visited before accessing a certain controlled page—starting at page A, you must go through an access-control page B to reach page C.

Each of these three properties involves a notion of temporal sequencing of events. Further, each prescribes that only certain sequences of events should occur on all executions of a system. These qualities suggest the application of model checking.

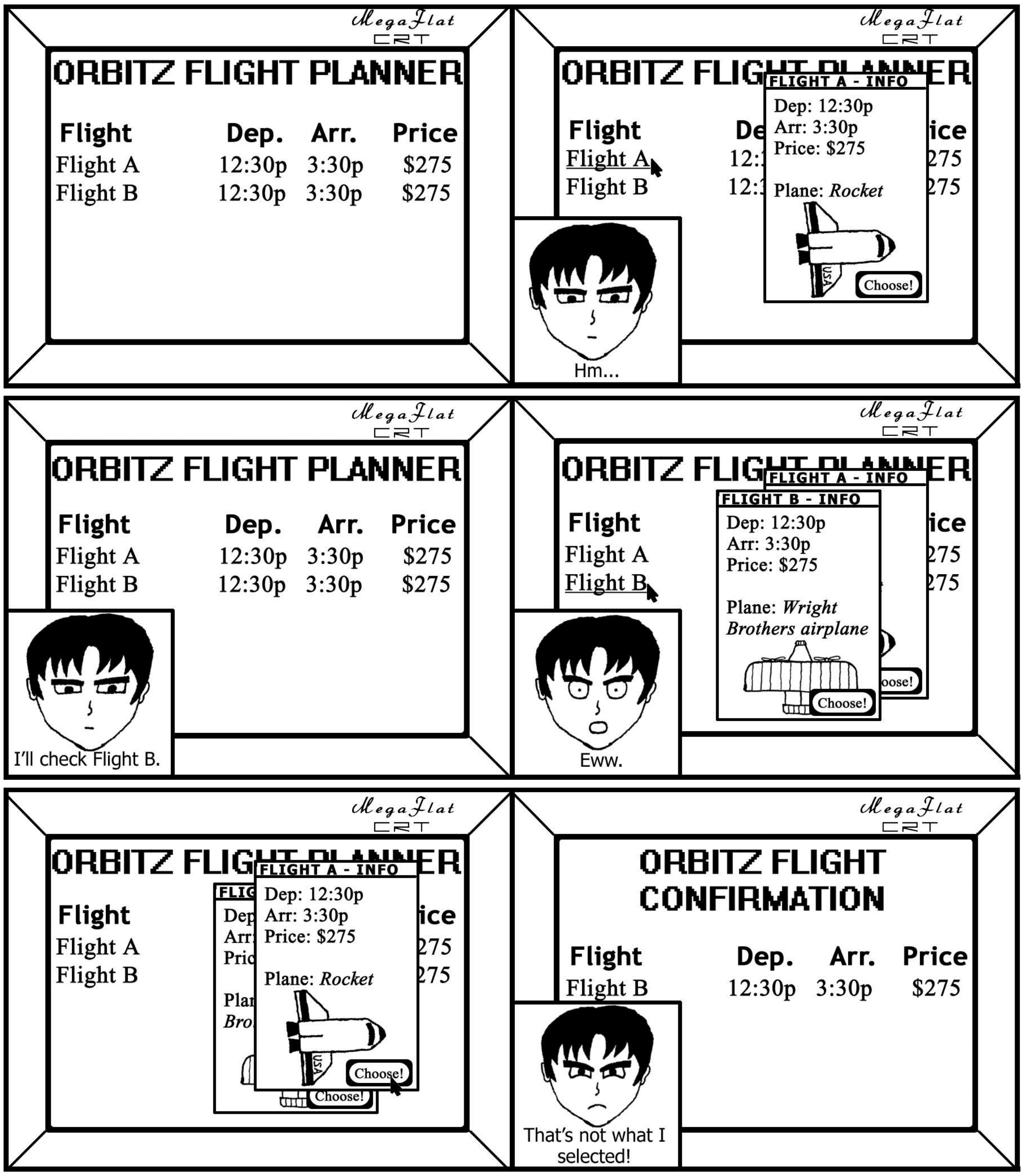


Figure 2.1: The Orbitz Bug; comic by Matt Licata (used with permission)

The comic above shows the result of each of the following steps:

[Step 1] A user enters the desired dates and destination of his flight; he is then presented with a page listing possible flights, including Flight A and Flight B.

[Step 2] He clicks a link to open the description of Flight A in a new browser window.

[Step 3] Not being particularly enthused about that flight, he returns to the list of flights . . .

[Step 4] and clicks a link to load the description of Flight B, again in a new browser window.

[Step 5] Deciding that Flight A was better after all, he switches back to the window still on the screen showing Flight A . . .

[Step 6] and submits the form, causing a page confirming his reservation to be displayed.

[Result] Orbitz incorrectly makes a reservation on Flight B.

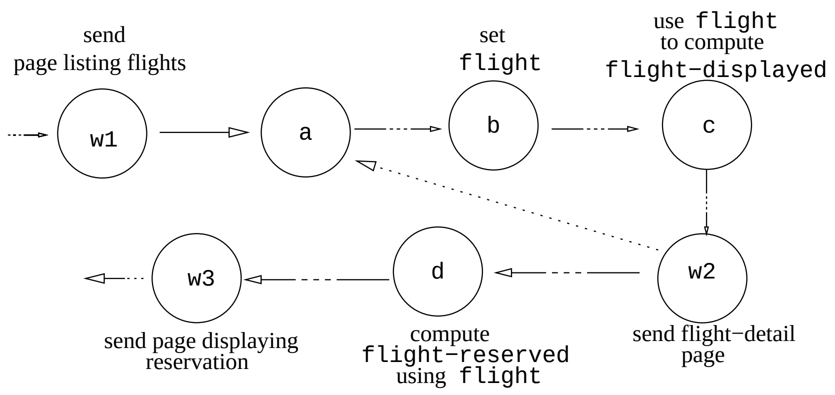


Figure 2.2: Orbitz control-flow graph
Solid denotes immediate successor; solid-with-ellipsis denotes elided nodes; dotted denotes Web control-flow edge.

2.2 Verification Approach: Model Checking

To apply model checking, a developer first creates a model of the system being verified and then writes down the correctness properties with respect to which he would like to verify the model; he then applies a suitable model checking algorithm, which consumes the model and the properties and tells him whether the properties hold for the model.

We now informally work through this methodology for the Orbitz example given above. The desired property is that `flight-reserved` equals `flight-displayed`. Prior work has shown what code for a Web program that exhibits the Orbitz bug would look like [10]; we must extract a model from this code. We could first try the most straightforward technique: from the source code, generate a control-flow graph. A sketch of a control-flow graph is depicted in Figure 2.2 (ignore the dashed line for now—all the other edges correspond to control-flow from the actual code); though we have elided many details for the sake of presentation, all relevant events (in particular, all assignments to the variable `flight`) are shown. Looking at this model, we notice something interesting: there is no error! Once `flight-displayed` has been computed using the value of the variable `flight`, the program must use the same value to compute `flight-reserved`.

Our analysis failed because the sequence of operations (listed in Figure 2.1) that exposed the bug included several instances of the user using his browser to return to and resubmit a previously visited Web page. These actions exploited additional control flow not present in the standard control-flow graph; we must therefore augment our model. The dotted line in Figure 2.2 shows the control-flow-graph edge necessary to capture the user's ability to return to and resubmit the list-of-flights page (Steps 3 and 4 in Figure 2.1). (No edge needs to be added corresponding to the browser window switch in Step 5 because the two pages were generated by the same program expression.) This added control flow exposes the bug—now, we can see that the user might set `flight` to a new value and then submit the page generated using the old one. To develop a sound model of Web programs, we will need to add similar edges for all possible user operations.

2.3 User Operation Calculus

Rather than accounting for each individual operation that a Web browser provides, we use a calculus of primitive user operations due to Graunke, et al. [10]; all traditional browser operations can be expressed in this calculus. Consequently, these primitives are the only user operations about which our verification tool needs to reason.

Like Graunke et al. [10], we distill the Web to a single server and a single user client. The user's client displays Web pages and accepts input; each page includes some text and provides a single form that can be submitted. The client stores a currently active page and a cache of previously visited Web pages (which initially contains some start page), and at each step the client can either *submit* the current page's form or *switch* to a previously visited, cached page. When the user *submit*s a form, the server dispatches the request to the correct Web program, which generates and returns a new page based on the client's input. When he *switch*es the client's currently active page, the client does not communicate this change to the server. It is the Web program's lack of knowledge about these *switch*es that causes so many subtle bugs.

In the present work, we make two simplifying assumptions about user operations: we do not account for the user typing in a URL, and we assume that a Web browser's cache contains all previously visited Web pages. The first is equivalent to the assumption that all relevant pages can reached by *submit*s starting from some start page, so we will refer to it as the *start-page assumption*. We will refer to the second as the *cache assumption*. The cache assumption is reasonable because servers usually expire any computations that have been started but not completed after a certain amount of time (minutes to hours); it is reasonable to assume that the user's browser cache will contain all pages visited in this same period of time. We assume that all interactions with the programs we analyze take place before the server session expires; this has the slight disadvantage of not allowing us to verify properties about the session expirations themselves.

Given these assumptions, we can express the following user operations as combinations of *switch*es and *submit*s:

- **Form Submit, Link:** These actions are modelled as a *submit*; a link is simply a form with no attached data.

- **Back Button, Forward Button, History, Bookmark, Browser Window Switch:** By the start-page assumption, each of these actions involves loading a previously-visited page, which, by the cache assumption, will be in the cache. It can thus be modelled as a *switch* to the indicated page.

- **Refresh:** Refreshing is the act of re*submit*ting to the server the request that generated the client's currently-active page. By the start-page assumption, a page B is either the start page or the result of *submit*ting a form on page A with certain data. We assume for simplicity that the start page is static, so refreshing the start page is a no-op. In the other case, refreshing page B can be modelled as a *switch* to page A followed by a *submit* of the same data that was included in the *submit* that generated B.

- **Clone:** Cloning is the act of opening a copy of the current browser window on the screen. Cloning itself does not affect the course of Web interactions: if the user clones page A but then only *submit*s from one of the clones, then it is just as if he had not cloned at all. However, cloning enables the following situation: given a page A with successors B and C (which correspond to *submit*ting different data from page A), the user can go forward from A to B and from the clone of A to C. In a real browser, this sequence of events might produce different results than *submit*ting A to get to B, going Back to A, and then *submit*ting to get to C. In the first sequence, A is guaranteed to still be in the cache (indeed, it is still on the screen), whereas in the second sequence it is not. However, with the cache assumption, these two sequences of actions are equivalent. The situation enabled by cloning can therefore be modelled as a *submit*, a *switch*, and then another *submit*.

In addition to simplifying the task, verifying using this calculus provides some robustness in the face of new operations that browsers might someday provide—as long as they can be expressed in terms of *switch* and *submit*, we will not need to change our technique.

Chapter 3

Generating Models From Source

In this chapter, we describe how we generate a model of the control flow of a Web program from its source code. We assume that the programs we verify are written using special *Web-interaction procedures*. In the present work, we analyze programs written in PLT Scheme that exploit the Web programming procedure `send/suspend` [12]. This procedure consumes a representation of a Web page and sends that page to the user; when he submits the form on the provided page, the Web program resumes computation with the values submitted by the user. In PLT Scheme, this primitive is implemented using continuations (following the lead of Queinnec [18]). When a Web program is written in this form (rather than as a collection of independent scripts), it is not necessary to reason about the marshalling and unmarshalling of data at every Web-interaction point.

We model a Web program P by its *Web control-flow graph* (*WebCFG*). The WebCFG is an augmented control-flow graph (CFG).

3.1 The Control-Flow Graph

A control-flow graph describes all possible sequences of source expression evaluations during all executions of a program. An ideal control-flow graph of a program P would be a graph with one node for each evaluation of each expression in P. However, if P does not halt, this ideal control-flow graph will not have a finite number of states. Model checking requires a finite-state model, so some form of approximation is necessary.

We approximate the control-flow of a program P by defining its CFG (N, n_0, E) as follows:

- The node set N contains one node corresponding to each expression in the source code of P. We denote by $expr(n)$ the source expression corresponding to the CFG node n.

- $n_0 \in N$ is a unique start node

- The edge set $E \subseteq N \times N$ contains an edge (n_1, n_2) iff $expr(n_2)$ might be the next expression to be evaluated after evaluating $expr(n_1)$.

This CFG collapses all those nodes in the ideal graph that correspond to different executions of the same source expression into a single node (which agglomerates all of their edges). This collapsing corresponds to Shivers's 0CFA [19].

The "might be" in the definition of the edge set is necessary because the control flow of a program often varies from one execution to the next, but we want the control-flow graph to model the control flow of all possible executions. The control flow can vary at a conditional (different branches may be taken on different executions) and at the last expression of a procedure body (different call sites may be returned to). Additionally, in a higher-order language, the control flow can vary at a procedure application (different procedures may be called in different executions). Exceptions and continuations create additional branching.

3.2 The Web Control-Flow Graph

As we saw in the Orbitz example, the control-flow graph of a program does not capture the control flow engendered by user browser operations. We must add this control flow to our model if we are to build a sound verification tool. By the reduction to primitive user operations in Section 2.3, we must only account for *switch* and *submit*. When the user performs a *switch* and then a *submit*, *he causes the program to return from a different Web interaction expression than normal control flow would predict*. That is, when the user performs a *switch*, he changes which Web-interaction call will be returned from on the next *submit*; the program will return from the one that generated the page to which he switched. A *submit* without any *switch*es introduces no new control flow—control proceeds as expected to the successors of the Web interaction expression that generated the submitted page.

To model the control flow enabled by *switch*, we should add a transition from each Web-interaction node w to the successor nodes of each Web-interaction node that the user passed through before reaching w; this corresponds to the user being able to return from (i.e., *switch* to and re*submit*) any previously visited page. Because we cannot add this exact set of edges without exploding the size of the model (see Appendix **??**), we overapproximate the `switch` control flow by adding to the CFG an edge from each Web interaction node to the successors of every other Web-interaction node.

Formally, we can define the WebCFG $(N^{WebCFG}, n_0^{WebCFG}, E^{WebCFG})$ of a Web program P with CFG $(N^{CFG}, n_0^{CFG}, E^{CFG})$ as follows. Let W be the set of Web interaction nodes in P. Let X be a set of fresh nodes, called *post-Web-interaction nodes*, where $|X| = |W|$ and $x_i \in X$ is called the *post-Web-interaction node* corresponding to w_i. Then:

- The set N^{WebCFG} of nodes is given by $N^{CFG} \bigcup X$.
- $n_0^{WebCFG} = n_0^{CFG}$.
- The set E^{WebCFG} of edges is the union of
  - $\{(n_1, n_2)$ where $n_1, n_2 \in N^{CFG} - W$ and $(n_1, n_2) \in E^{CFG}\}$
  - $\{(x_i, n)$ where $(w_i, n) \in E^{CFG}\}$
  - $\{(w_i, x_j)$ for all $w_i \in W$ and $x_j \in X\}$.

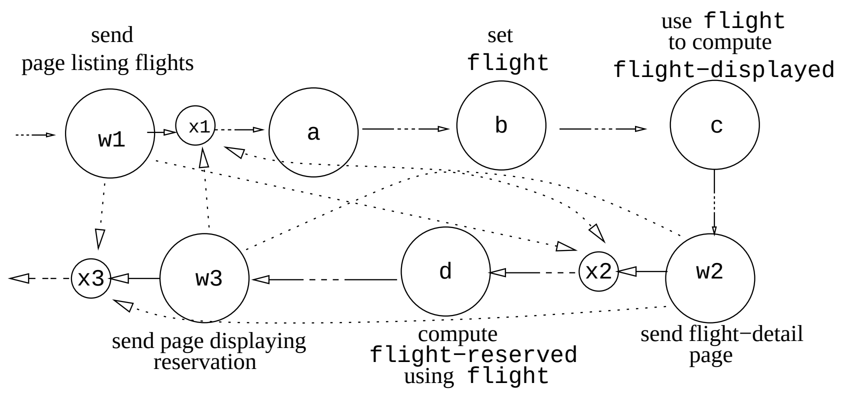


Figure 3.1: Orbitz WebCFG
This figure uses the same arrow convention as Figure 2.2.

We could have directly added edges from each Web-interaction node to the successors of every other Web-interaction node, but we have instead introduced the post-Web-interaction nodes to collect these edges—the reasons for this are detailed in Section 5.2.

We construct the WebCFG completely automatically from the source of a Web program using a standard CFG construction technique (Set-Based Analysis [9, 13] approximates the values of procedure-call positions; the CFG can then be constructed by traversing the program's syntax) followed by a simple graph traversal to add the post-Web-interaction nodes and the Web-interaction edges. Figure 3.2 presents the WebCFG corresponding to the Orbitz CFG considered in Figure 2.2.

Chapter 4

Properties

If we annotate the nodes of the WebCFG with elements of some set of atomic propositions, then the graph will describe all traces (atomic proposition sequences) that might occur during execution. The developer formulates a desired property as a set of traces that should occur. Verification then reduces to containment of the former in the latter [21].

Developers specify a set of traces as an automaton whose input alphabet is the set of atomic propositions. Following Naumovich, et al. [17], whose algorithms we employ, we require developers to write separate automata for safety and liveness properties (because any property described in terms of traces can be decomposed into a safety property and a liveness property [1], this separation does not disallow any properties). Informally, a safety property prescribes that "something bad never happens", a liveness property that "something good eventually happens". Because safety properties are refutable by finite traces, they are expressed as finite-state, finite-word automata with a designated violation (non-accept) state. Liveness properties are written as deterministic Büchi automata [5]. The determinism is imposed by the model-checking algorithm we have chosen [17], and in principle slightly limits the class of properties we can verify (though we have not encountered this obstacle in practice).

To state concrete properties about Web programs, we must overcome two more challenges:

- To express the Orbitz property, we must be able to talk about strings (the `flight-displayed` and the `flight-reserved`) that appear on Web pages. How can we identify the program expressions that generate these strings?
- The atomic propositions that label WebCFG nodes must be simple enough to be generated automatically, yet rich enough to enable the expression of interesting properties. What should these be?

The next two sections address each of these problems in turn. We then present some example properties, followed by three common property idioms that ease specification.

4.1 Identifying Web-Page Content

How can we associate Web-page contents with the source expressions that generate them? Parsing the HTML fragments in the program to search for strings is likely to be complex, unwieldy and highly sensitive to data and formatting changes. Forcing the developer to use special language constructs to label source expressions is intrusive and not portable. Finally, using static-distance coordinates is brittle in the face of program evolution.

We create a solution that is lightweight, robust in the face of change, and unintrusive by observing that the association is often already in the Web program's source! Web developers often use Cascading Style Sheets (CSS) to tag important page elements with an ID for which independent style-sheets provide formatting directives. We simply ask developers to associate a CSS ID with any Web page element they want to refer to in a property and then to use that ID in an atomic proposition. This ID allows us to identify the source expression that generates the associated Web page element. We similarly use the names of user-input fields to identify source expressions that extract the values of submitted forms. This solution avoids the complexity and brittleness of the other proposed solutions; it has the additional advantage of being very easy for developers to comprehend—we have essentially integrated Web presentation elements into the property language.

We refer to a source expression that generates a CSS-tagged HTML element or extracts form input as a *tagged expression*. For example, the Orbitz code might contain tagged expressions that generate HTML with CSS IDs `flight-displayed` and `flight-reserved`, while a search engine might contain one that accesses the user input `query`.

4.2 Atomic Propositions

Informal Description

Some of our atomic propositions are designed for reasoning about misuse of two sets of data bindings: the data local to each page the user sees (e.g., hidden form fields) and the data shared by all pages (e.g., session state and cookies). Many common errors in Web programs result from this misuse [10, 11]. The two data sets have different properties in situations where the user performs browser actions between the generation and the submission of a given Web page (for example, Steps 3, 4, and 5 of Figure 2.1): bindings local to each page are guaranteed not to change between page generation and submission, whereas shared bindings may be modified. The Orbitz property can be true if the flight is kept local to each page; it will be violated if the flight is shared by all pages. The Amazon property can be true if the shopping cart is shared by all pages; it will be violated if the shopping cart is local to each page.

Our set of atomic propositions consists of:

- `tagged` propositions that are true on WebCFG states corresponding to Web program expressions with CSS or user-input tags. These allow the developer to check that certain states have certain values and to reason about the value flow from one expression to another.
- `set` and `join` propositions that are true on states corresponding to operations on shared data. `set`

operations replace one value with another, whereas `join` operations add a new value to a collection including all the old ones (for example, mutatively adding to a list is a `join`). These propositions are useful in specifying the Orbitz and Amazon properties.

- `web` and `postweb` propositions that are true at states corresponding to Web-interaction expressions and their successors. These propositions allow the developer to write properties about the sequencing of Web-page generation.

The WebCFG can automatically be annotated with these atomic propositions using the results of a data-flow analysis called Set-Based Analysis [9, 13]. However, this analysis requires knowledge about the potential return values of all primitive operations, which means knowing the sets of values that a user might type into each form field. As in the work of Benedikt, et al. [3], we presume that the developer has written down some approximation of these values. We assume our tool is given an explicit dictionary-style mapping from field names to values; this mapping could be generated from a more sophisticated (and hence less burdensome to create) user input abstraction such as SmartProfiles [3].

Formal Description: AP and Labelling

We now give a formal description of our atomic propositions and the rule for labelling the WebCFG states with them. Except for the nuances of the example property automata that will be presented, the rest of this paper can be read without understanding these details.

Set-Based Analysis produces two useful outputs. First, it computes an overapproximation of the runtime values of each expression in the source of the program. Second, it generates a set of flow variables for each expression in the source; this set contains one flow variable for each potential value of that expression. The value set enables reasoning about the values of an expression, whereas the flow variable set enables reasoning about value flow between expressions. In particular, if the value of one expression flows into the value of another, then all flow variables associated with the first will also be associated with the second.

In following, we will use the term *environment* to refer to the data local to a Web page that the user sees and the term *store* to refer the data shared by all pages. We assume that a Web program has exactly two syntactically identifiable store operations `set` and `join`. A `set` replaces the value of its first argument with the value of the second, whereas a join adds the value of the second to a collection containing all values previously `join`ed to the first.

Notation: $expr(n)$ denotes the source expression corresponding to the WebCFG node n. $Tags$ denotes the set of all tags; $nodes(tag)$ denotes the WebCFG nodes tagged with $tag \in Tags$. V denotes the set of all program values, FV denotes the set of all flow variables, and for $n \in N^{WebCFG}$, $V(n)$ and $FV(n)$ denote the value and flow variable sets corresponding to $expr(n)$. Flow-variable expressions are described by the grammar `FVE ::= s | v | SCv` and `SC ::=` $=$ `|` $\subseteq$ `|` $\not\subseteq$ `|` $\supseteq$ `|` $\not\supseteq$ where $s \subseteq FV$ and v is a flow-variable variable.

AP consists of five kinds of tuples, `tagged`, `set`, `join`, `web`, `postweb`. Their types are as follows (we abuse notation here by using each of the tuple names as a type for that tuple name literal):

- `tagged` $* Tags * V * FV$

- $\texttt{set} * Exprs * FVE * FVE$
- $\texttt{join} * Exprs * FVE * FVE$
- $\texttt{web} * \mathbb{Z}$
- $\texttt{postweb} * \mathbb{Z}$

A labelling function $L : N \rightarrow \mathcal{P}(AP)$ associates each node in the WebCFG with a set of atomic propositions that are true at that node. For a given n, $L(n)$ includes:

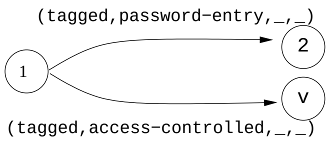
- $(\texttt{value}, tag, V(n), FV(n))$ iff $n \in nodes(tag)$.
- $(\texttt{set}, n_x, FV(n_x), FV(n_v))$ iff $expr(n)$ is an expression that `set`s $expr(n_x)$ to $expr(n_v)$.
- $(\texttt{join}, n_x, FV(n_x), FV(n_v))$ iff $expr(n)$ is an expression that `join`s $expr(n_x)$ to $expr(n_v)$.
- $(\texttt{web}, m)$ iff n is Web-interaction number m
- $(\texttt{postweb}, m)$ iff n is Web-interaction number m. Corresponding Web/post-Web nodes have the same number.

There are a few subtleties in how the developer uses these atomic propositions in properties. When writing `set` and `join` propositions explicitly, the developer must either specify the CFG node by source position, not specify it at all, or use a rule to generate properties for the specific model being checked (in the examples presented later, we will use the second and third approaches). The flow-variable expressions exploit the fact that our verification algorithm is parameterized by the definition of atomic-proposition matching (that is, determining when a propostion labelling a state in the model is the same as a proposition labelling a transition in the property). Other than FVE positions, atomic propositions must exactly match. FVE positions are matched as follows: a literal set in the property matches an identical literal set from the model; a flow-variable variable in the property matches any literal set on the model, and the variable is associated with the literal set in a relation kept by the matching routine; a set-constraint in the property matches a literal set in the model iff the constraint is satisfied for all literal sets related to the constraint's variable. This notion of matching allows us to state value-flow relationships in temporal properties.

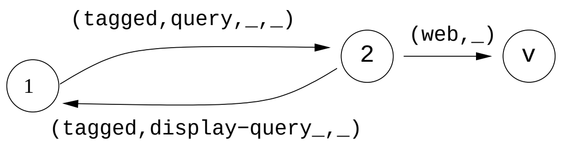
4.3 Example Property Automata

In the following, we label the violation state of a safety property v and the accept states of a liveness property with double concentric circles. Any atomic proposition that is not shown labels a self-loop; any part of a proposition shown with an underscore does not affect atomic proposition matching.

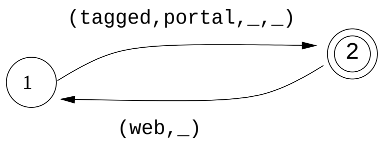
Assuming that there is an expression with CSS tag `password-entry` on the password page and an expression with CSS tag `access-controlled` on the access-controlled page, we can translate the password-page safety property described in Section 2.1 into a property automaton:



Assuming an input tagged `query` and a page element tagged `display-query` that displays the query back to the user on the results page, we can check the property that a search engine always displays the results of a user's query on the next-generated page by verifying both that `display-query` *takes the value of* `query` and that the following automaton is satisfied. This automaton uses the `web` propositions to state that the displayed text is generated before the next page is sent:



Assuming a tag `portal` on a portal page, we can check that a portal page is always eventually reachable by verifying that this automaton is satisfied:



4.4 Property Idioms

Though we could now write the Orbitz and Amazon properties directly as automata, we instead define three property idioms of which they are instances.

So far, we have described the Orbitz property as a relationship between the value of the expression tagged `flight-displayed` and the value of the expression tagged `flight-reserved`: the value of `flight-reserved` must be generated from the value of `flight-displayed` displayed on the page that the user submitted to make his reservation (call this page p_{prev}). This property is implied by the conjunction of two other properties. First, all potential values of `flight-reserved` are also values of `flight-displayed`. Second, no values used in the computation of `flight-reserved` that were present when p_{prev} was generated have changed since its generation. Similarly, we want to capture the Amazon property that once a user selects an item to buy, it it appears in his shopping cart. Assuming appropriate CSS taggings, we state this by saying that the `shopping-cart` contains all values input as `selected-item`.

We offer generalized versions of these properties as idioms in our property language. In the following definitions, let e_1 and e_2 denote expressions in the Web program source.

- We say that e_1 *takes the value of* e_2 iff any potential value of e_1 must also be a value of e_2. The first Orbitz subproperty is an instance of this idiom.

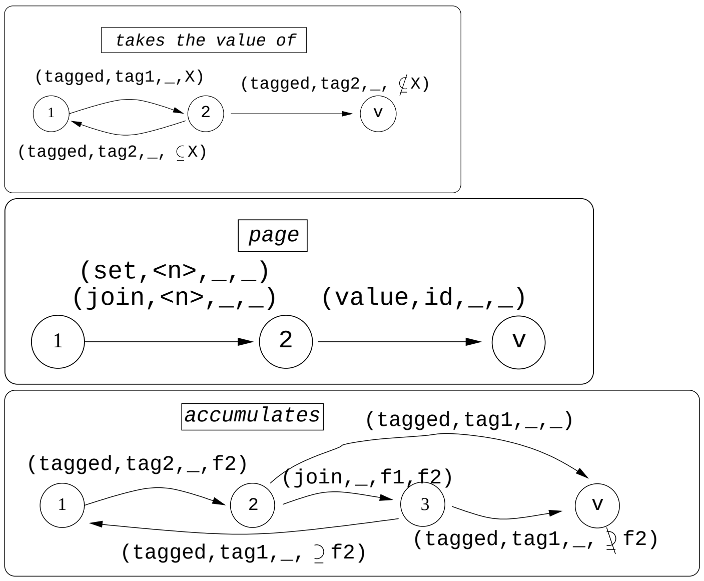


Figure 4.1: Property Idioms as Automata
$expr(tag1)$ *takes the value of* $expr(tag2)$, $expr(tag)$ is *page*, and $expr(tag1)$ *accumulates* the values of $expr(tag2)$. For *page*, we label the transition from state 1 to state 2 with one `set` and one `join` for each node whose value is used to compute $expr(tag)$ (we identify these using the flow variables).

- Let p_{prev} denote the last page the user saw before the evaluation of e_1. Then we say that e_1 is *page* iff no values used in the computation of e_1 that were present when p_{prev} was generated have changed since that page's generation. The second Orbitz subproperty is an instance of this idiom.
- We say that e_1 *accumulates* the values of the e_2 iff if the value of e_1 contains the value that e_2 produces at each evaluation (where the exact notion of containment depends on the type of value that e_2 produces). The Amazon property is an instance of this idiom.

Figure 4.1 shows how to write our idioms as automata over the full set of atomic propositions. *takes the value of* is simple to express with flow-variable set-constraints. *page* requires that no value used to compute the *page* value can be kept in the store (if some value were kept in the store, the Web control flow would allow it to be mutated between page generation and submission). *accumulates* requires that the accumulated value be `join`ed to the accumulating value.

Chapter 5

Verification Process

5.1 Verification Algorithm

The model and property language described above are derived from those used in the FLAVERS toolkit [6]: the FLAVERS algorithms consume a model represented as a graph whose nodes are annotated with certain atomic propositions and a property written as an automaton over those same propositions. We may thus reuse the FLAVERS model checking algorithms in our work. We give only an intuitive description of the algorithms; they are presented formally by Naumovich, et al. [17].

In FLAVERS, a slightly different algorithm is used for verifying liveness properties than for verifying safety properties. Both start with a common subroutine: traverse the model and associate with each model state the property states that are reached at that model state (the atomic propositions reached on model states drive the property automaton, using the particular definition of matching described in Section 4.2). Continue until no model state n can be reached with the property in a state not already associated with n.

Then, to check if a safety property is true for the model, ascertain that no model states are associated with the violation state of the property. Recall that a liveness property is expressed as a deterministic Büchi automaton, and that such an automaton accepts a string iff it reaches an accept state infinitely often. To check a liveness property, form a cross-product graph between each state in the model and the property states that were reached at that state, and prune this graph of all nodes where the property is in an accept state. Then, ascertain that this restricted graph does not contain any strongly connected components. This is a standard model-checking technique (used also in LTL model checking [5, 21]); it relies on the observation that an infinite path where the property never reaches an accept state exists iff such a strongly connected component exists. It is this step that requires the restriction of our property language to deterministic Büchi automata.

At this point, we are able to discover the bug in a model of Orbitz. Using the state labellings from Figure 3.2 (and only mentioning the depicted nodes), we see that the trace $[w_1, x_1, a, b, c, w_2, x_2, d]$ violates the property that the expression tagged `flight-reserved` is *page*.

5.2 Improving Precision and Efficiency

In this section, we present two improvements upon our verification technique. The first reduces the number of spurious errors; the second improves the time efficiency of the verification task.

Constraint Automata for Better Precision

When we discovered the Orbitz bug, we also found that the trace $[w_1, \underline{x_2, d, w_3}, x_1, a, b, c, w_2, x_2, d]$ failed to satisfy the desired property. This corresponds to the user visiting the successors of the second Web-interaction point before he has even gotten to generating the second page, something he clearly cannot actually do. Where did this spurious path come from? When we first defined the WebCFG, we added edges from each Web-interaction node to the successor of each other Web-interaction node. This overapproximates the control flow introduced by Web interactions: in reality, a user can only *switch* to pages he has seen before, not to any page at all. In this case, the overapproximation resulted in a spurious trace being reported for a program that actually was incorrect. In other cases, it will cause correct programs to be deemed incorrect—for example, the password-page property would never hold, as these spurious model paths would make it seem as if the user could always jump directly to the access-controlled page.

We can improve the number of correct programs that we deem correct by eliminating these infeasible paths. The naïve way of accomplishing this would be to redesign the WebCFG, adding many slightly augmented copies of the original graph to represent the enabling of new transitions and adding the appropriate transitions between these copies. This approach would cause an exponential explosion in the size of our model, which in turn would drastically increase the time required to check properties over it.

Fortunately, the FLAVERS algorithm gives us a better option. The full FLAVERS algorithm allows the developer to specify any number of *constraint automata* in addition to the property. A constraint automaton, like a safety property, is a finite-state, finite-word automaton with a single violation state that is driven by the propositions reached on the model states. However, the interpretation of reaching the violation state is different: when a constraint automaton is violated, the model path leading to that violation is no longer considered valid; the property is thus allowed to be violated on such paths. The modified FLAVERS kernel can be used directly for the safety property algorithm; it requires only a slight modification (the cross-product graph is now over the model, the property, and all of the constraints) for the liveness-property algorithm. Constraints thus provide an easy and efficient way to prevent certain paths in the model from affecting the results [6].

Helpfully, the constraints needed to remove the spurious paths we introduced in the WebCFG can be generated automatically. We create one constraint of the following form for each Web-interaction node:

$$\begin{tikzcd} & 2 \\ 1 \arrow[ur, bend left, "{\texttt{(web,m)}}"] \arrow[dr, bend right, "{\texttt{(postweb,m)}}"'] & \\ & v \end{tikzcd}$$

Constraint m will be violated on any path where post-Web-interaction node m is visited before Web-interaction node m; since we have a constraint for each Web-interaction node, at least one constraint will be violated on any path that includes a *switch* to a previously unvisited page. The post-Web-interaction nodes are necessary for specifying these constraints: because we have folded all evaluations of an expression into one WebCFG node, any original CFG node can potentially be reached before any given Web interaction node. Thus, none of these original nodes can be used as the atomic proposition that sends the constraint to its violation state without creating the possibility that the constraint will be violated on a valid path.

Property-language-driven Optimization

Our labelling function associates certain nodes in the WebCFG with the empty set of atomic propositions. Because of the way we have defined the verification process, these unlabeled nodes have no influence on the verification results (since the atomic propositions reached on the model states are the inputs to the property and constraint automata, traversing unlabeled model states will have no effect). Thus, we remove any sequence of unlabeled states from the graph, connecting the predecessors of the sequence directly to its successors. Section 6 shows that this optimization can have quite a dramatic impact in practice.

5.3 Soundness

We can now state a soundness result of our model checker: if the model checker claims that the WebCFG of a Web program P has a certain property, then that property will hold for all executions of P during which the user performs only *switch*es to previously visited pages and *submit*s. This result follows directly from the soundness of the FLAVERS algorithms and the fact that the WebCFG and the atomic proposition labelling are overapproximations. The WebCFG overapproximates the control flow of the Web program (i.e, if a sequence of expressions is evaluated in order in some execution of the program, then the corresponding sequence of nodes appears in the WebCFG) because the standard CFG construction techniques yield an overapproximation and the edges added to form the WebCFG but not disallowed by the constraints exactly reflect the control flow enabled by the user operations. Our atomic proposition labelling is an overapproximation (i.e., if an atomic proposition holds for a program expression, then the WebCFG node corresponding to that expression is labelled with that proposition) because Set-Based Analysis [9, 13] overapproximates runtime values. These two facts imply that the set of atomic proposition traces along paths in the WebCFG is a superset of the set of atomic proposition traces that actually occur at runtime. By the soundness of FLAVERS, if the model checker claims that a given property holds for a WebCFG, then that property is true for all atomic proposition traces along paths through the WebCFG; in particular, it is true for the subset of those traces that occur at runtime.

5.4 Complexity

The complexity of our method is determined by the complexities of the various phases. The data-flow analysis has worst-case time complexity $O(n^3)$ where there are n expressions in the program source; building the WebCFG then takes time $O(n^2)$, so the time for constructing the model is $O(n^3)$. The FLAVERS safety algorithm takes time $O(n^2 \cdot p \cdot k)$ where p is the number of states in the property and k is the product of the numbers of states in each of the constraints. If a developer uses only the constraints we generate for the Web control flow, then there will only be one constant-size constraint for each Web-interaction node. Thus, k is $O(w)$, where w denotes the number of Web-interaction nodes, and therefore k is $O(n)$. In this case, we get an overall worst-case upper bound of $O(n^3 \cdot p)$, but we will often do better—w is likely to be much less than n. Checking liveness properties requires additional time for detecting strongly connected components. Our space complexities are the same as those of FLAVERS.

Chapter 6

Implementation and Results

We have implemented the algorithms described above for constructing the WebCFG and verifying safety and liveness properties. Our implementation accepts Web programs written in PLT Scheme, so we rely on an implementation of Set-Based Analysis called MrFlow [16] for our data-flow analysis. Because our notion of atomic-proposition matching is nuanced (see Section 4.2), we use a quick reimplementation of the relevant algorithms.

Our implementation makes some simple assumptions to aid in data reasoning. MrFlow does not provide useful value set information about strings, which constitute most Web pages. This is because strings can be combined and decomposed in an arbitrary manner. In contrast, many Web applications do not decompose strings; they only combine strings collected from various sources. (The use of structured forms decreases the need to inspect strings for implicit patterns, such as prefixes that determine gender; this information is instead collected explicitly through separate form fields.) These strings therefore closely resemble collection data structures such as lists (about which MrFlow provides rich value-flow information). We therefore map the string primitives onto list primitives, which enables us to trace the flow of strings through the program. This restriction has sufficed for the programs we have verified.

We have begun to verify CONTINUE [14], a conference-management system that has been used for ISSTA 2004 and many other conferences. Preliminary results are encouraging: an initial WebCFG contained 17,200 nodes, but the property-language-driven state space optimization yielded a model with approximately 300 nodes (the exact number depends on how many expressions are tagged for property use).

Chapter 7

Related Work

There have been some past efforts to apply formal verification techniques to the Web. De Alfaro [7] uses model checking techniques to verify properties of static Web pages. He treats the page and link structure of the web as a model and then verifies properties written in a slightly restricted μ-calculus over that model. This technique allows him to check many path properties over static Web sites (such as the password-page property) and to present errors as paths through the Web model that violate a given property. Unlike de Alfaro, we are interested in proving properties of interactive Web sites.

Benedikt et al.'s VeriWeb tool [3] explores interactive Web sites using a special browser that systematically explores all paths up to a specified depth. A user of this tool first makes a model approximating the values that a user might type into the forms of interest. Next, the user specifies properties (such as string containment) about individual Web pages. The verifier then traverses the Web sites of interest and reports errors as sequences of Web operations that lead to a page which violates a property. As in this work, we are concerned with verifying properties of interactive Web sites. However, our work addresses several key limitations of this tool. First, it does not take into account user operations, so his tool is unable to catch errors that occur only in their presence. Our verifier accounts for the control flow enabled by user operations and discovers user-operation-related bugs. Secondly, its verification is limited to single-page properties. In contrast, we provide a method for verifying all-paths properties of interactive Web sites. We can do this because our verifier operates statically on the program's source, whereas Godefroid's tool is dynamic, effectively "running" the Web site as a Web browser would.

Baresi et al. [2] observe a bug in Amazon and extended UML's OCL with assertions to capture it. These assertions roughly correspond to the properties that we have expressed with *page*; in contrast, we provide a much richer property language. Furthermore, it is unclear how to verify a program against their assertions, as the authors provide neither an algorithm nor a mapping to traditional OCL verifiers.

Our formal model for Web operations is given by Graunke, et al. [10]. Using this model, the authors created a type system that statically discovers abuses of the values filled into form fields and devised a strategy for detecting data inconsistency problems such as the Orbitz bug. However, these inconsistency problems are only detected dynamically through changes to the server's run-time system. In contrast, our system is static and can provide guarantees about all possible execution sequences.

Chapter 8

Future Work

In the future, we would like to perform more case studies to further demonstrate the utility of our tool. We expect that these will help us identify more property idioms, eventually resulting in a catalog of verification patterns, akin to that of Dwyer et al. [8], but for the Web.

To verify a larger set of Web applications, we must eventually permit richer reasoning about data. In particular, we must support a broader set of operations on strings (especially string decomposition), as well as arithmetic operations. We expect it will be essential to complement our model checker with a theorem prover: the model checker would output data propositions to the theorem prover, which would determine whether the desired property could be proven from those constraints.

One factor that greatly influences the utility of a model checker is the quality of the error traces it provides when a property is violated. Our technique has the advantage of being able to present traces very intuitively as sequences of Web pages and Web operations that lead to a violation. Indeed, we could potentially even generate this output in the WebVCR format [3] so that a developer could sit back and watch as the error is played out.

We have restricted ourselves to analyzing programs that use the `send/suspend` primitive. Many Web programs, unfortunately, are not written in this style. We conjecture two solutions to this problem. First, based on prior work [11], we conjecture that we can use an "inverse CPS transformation" to convert ordinary CGI programs into a form suitable for our verifier. However, such a tool would have to overcome many engineering obstacles. Secondly, we could treat individual CGI scripts as open features and use techniques [15] for reasoning about their composition.

Currently, we do not address the concurrency issues resulting from multiple simultaneous accesses to a server by different clients (which are different from those resulting from repeated sequential submissions of the same page by a client). Given that many Web sites allow multiple users to interact with the same data, this is an important path for future research. We hope to exploit results on atomicity to reduce the sizes of models involving multiple clients. This process might be abetted by the fact that our current design of the WebCFG includes more knowledge about the particular sequencing of some events than may be necessary (for instance, the order of various operations that occur in between two Web interactions may not matter).

Bibliography

[1] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[2] L. Baresi, G. Denaro, L. Mainetti, and P. Paolini. Assertions to better specify the amazon bug. In *14th Software Engineering and Knowledge Engineering*, 2002.

[3] M. Benedikt, J. Freire, and P. Godefroid. VeriWeb: Automatically testing dynamic web sites. In *International World Wide Web Conference*, Honolulu, 2002.

[4] L. D. Catledge and J. E. Pitkow. Characterizing browsing strategies in the World-Wide Web. *Computer Networks and ISDN Systems*, 27(6):1065–1073, 1995.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[6] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. FLAVERS: A finite state verification technique for software systems. *IBM Systems Journal*, 41(1), 2002.

[7] L. de Alfaro. Model checking the world wide web. *Lecture Notes in Computer Science*, Conference on Computer Aided Verification, 2001.

[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *21st International Conference on Software Engineering*, 1999.

[9] C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):369–415, 1999.

[10] P. Graunke, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Modeling web interactions. In *European Symposium on Programming*, 2003.

[11] P. T. Graunke, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Automatically restructuring programs for the Web. In *IEEE International Symposium on Automated Software Engineering*, pages 211–222, Nov. 2001.

[12] P. T. Graunke, S. Krishnamurthi, S. van der Hoeven, and M. Felleisen. Programming the Web with high-level programming languages. In *European Symposium on Programming*, pages 122–136, Apr. 2001.

[13] N. Heintze. Set-based analysis of ML programs. In *LISP and Functional Programming*, 1994.

[14] S. Krishnamurthi. The CONTINUE server. In *Symposium on the Practical Aspects of Declarative Languages*, 2003.

[15] H. C. Li, S. Krishnamurthi, and K. Fisler. Modular verification of open features through three-valued model checking. *Automated Software Engineering: An International Journal*, 2003.

[16] P. Meunier. Selector-based versus conditional-constraint-based data-flow analysis of programs. Master's thesis, Rice University, 2001.

[17] G. Naumovich and L. A. Clarke. Extending flavers to check properties on infinite executions of concurrent software systems. In *Monterey Workshop on Engineering Automation for Software Intensive System Integration*, 2001.

[18] C. Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *ACM SIGPLAN International Conference on Functional Programming*, 2000.

[19] O. Shivers. Control-flow analysis in scheme. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1988.

[20] M. Slatalla. Big, curved and road-ready? Book it. *New York Times*, 07-17-2003.

[21] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Logic In Computer Science*, pages 332–344, 1986.

Appendix A

Alternate WebCFG Construction

We claimed that overapproximating the *switch* and *submit* control flow in the WebCFG and then ruling out the spurious paths with constraints was necessary to avoid exploding the size of the model. In this section, we present a WebCFG construction that directly captures the exact control flow and analyze its affect on the state-space. Thus, our goal in this section is to augment the CFG with all and only edges corresponding to the user's ability to *switch* to and re*submit* any previously-visited page.

Construction

Let n denote the number of nodes in the CFG, and $m < n$ the number of Web interaction nodes. Denote Web-interaction node i by w_i. We still introduce the post-Web-interaction nodes, and we denote post-Web node i by x_i.

Single ordering: Assume to start that the CFG gives a total order on the execution of the Web interaction nodes—that is, the user must progress linearly from w_1 to w_2 …to w_m. This corresponds to the program not presenting the user with any choice about the order in which he visits the Web pages in a site (though of course he recovers this choice through browser operations). In this situation, there are two kinds of `switch` edges that we need to add: those from w_i to x_j when $i > j$ and those when $i < j$. Intuitively, the former correspond to using the Back button (perhaps multiple times), whereas the latter correspond to using the Forward button (perhaps multiple times). For the "Back" edges—for $i > j$—we can add the exact edges to the graph from the start without introducing any new states: because of the total order, any path that reaches w_i must have gone through w_j, and thus w_j corresponds to a previously-visited page. The "Forward" edges are more complicated: an edge from w_i to x_j when $i < j$ should only be present on a path in which the user has gone through w_j. This corresponds to the fact that the user must visit a page once through normal control flow before he can go "Forward" to it. We can mimic the enabling of these edges at the right time as follows: tile m copies of the original graph, where copy k is the same as copy $k-1$, except that it adds the "Forward" edges to w_k from all w_l such that $l < k$ (and copy 1 is simply the original CFG); link these copies with a transition from state w_k in copy $k-1$ to state w_k in copy k. As a path through this graph reaches new Web interaction nodes, it enables the appropriate "Forward" arrows by transitioning to a copy of the graph including them. The number of states produced by this copying can be cut down slightly by eliminating all original successors of w_l from copy l—there is no need to go on in both the copy without the “Forward” edges and the copy with them.

All orderings: The construction in the above paragraph gives the proper control-flow augmentation to reflect any executation trace visits the Web-interaction nodes in order from 1 to n. In some CFGs, the order in which the Web-interaction nodes are visited will change from trace to trace, depending on the data. We can model a trace that visits Web-interaction nodes in a different order by renumbering the Web-interaction nodes and repeating the construction above. To model the control-flow for all possible traces, we must repeat the process above for each ordering of the Web-interacction nodes allowed by the CFG, and then add a transition from the start node of the overall model to the start node of the model for each ordering.

Analysis

We now give a lower bound on the size of the model produced by this process in the worst case. The (unoptimized) single-ordering construction requires m copies of a graph with n nodes, so it requires $\Omega(mn)$ nodes. There exists an input program for which m is linear in n (in particular, m is $\Omega(n)$), so the number of nodes required in the worst case is $\Omega(n^2)$. In this case, the number of nodes required by the optimized single-ordering construction would be $\Omega(\Sigma_{i=1}^{n} i)$ nodes, so it would still be $\Omega(n^2)$. The all-orderings construction requires as many copies of the single-ordering construction as there are orderings of the Web-interaction nodes. In one case, all orderings of the Web-interaction nodes could be allowed by the CFG, which would give $m!$ possible orderings. Thus, the worst case has at least $m!$ orderings. Since m is linear in n, this gives $\Omega(n!)$ orderings, and thus $\Omega(n!)$ copies of a graph with $\Omega(n^2)$ states. Thus, the number of states in the worst case is $\Omega(n! \cdot n^2)$, which is clearly exponential in n.

We conjecture that any WebCFG construction that explicitly captures the control flow will similarly require an exponential state explosion.