In *CAV 01: Proceedings of the 13th Conference on Computer Aided Verification,* 2001.

Model Checking the World Wide Web[⋆]

Luca de Alfaro

Department of Electrical Engineering and Computer Sciences
University of California at Berkeley, Berkeley, CA 94720-1770, USA
Email: dealfaro@eecs.berkeley.edu

Abstract. Web design is an inherently error-prone process. To help with the detection of errors in the structure and connectivity of Web pages, we propose to apply model-checking techniques to the analysis of the World Wide Web. Model checking the Web is different in many respects from ordinary model checking of system models, since the Kripke structure of the Web is not known in advance, but can only be explored in a gradual fashion. In particular, the model-checking algorithms cannot be phrased in ordinary μ-calculus, since some operations, such as the computation of sets of predecessor Web pages and the computations of greatest fixpoints, are not possible on the Web. We introduce *constructive μ-calculus*, a fixpoint calculus similar to μ-calculus, but whose formulas can be effectively evaluated over the Web; and we show that its expressive power is very close to that of ordinary μ-calculus. Constructive μ-calculus can be used not only for phrasing Web model-checking algorithms, but also for the analysis of systems having a large, irregular state space that can be only gradually explored, such as software systems. On the basis of these ideas, we have implemented the Web model checker MCWEB, and we describe some of the issues that arose in its implementation, as well as the type of errors that it was able to find.

1 Introduction

The design of a Web site is an inherently error-prone process. A Web site must be correctly designed both at a local and at a global level. Good design at the local level implies that the pages contain well-formed HTML code, have the intended visual appearance, and have no broken links. Several tools are available for checking such local properties, either on single pages, or more commonly by crawling over an entire Web site: see for example [7, 20, 12, 13, 9, 14, 6, 11, 5, 19, 21, 8]. Good design at the global level requires that the Web site satisfies properties concerning its connectivity and cost of traversal, as well as properties that depend on the path followed to reach the pages, rather than on the pages only. Examples of such global properties are that every page of a Web site is reachable from all other pages, and that all paths from the main page to pages with confidential information must go through an access control page. Current Web verification tools focus essentially on local properties. On the other hand, model checking has proved to be an effective technique for the specification and verification of global properties of the large graphs that correspond to the state-space and transition relation of systems [4, 17]. Hence, it is natural to ask whether model checking can be

[⋆] This research was supported in part by the AFOSR MURI grant F49620-00-1-0327 and the NSF Theory grant CCR-9988172.

applied to the analysis of global properties of Web sites. This paper answers this question affirmatively, by showing how model-checking techniques can be adapted to the analysis of the Web, and by illustrating which types of errors are amenable to discovery with such techniques. In particular, we show that model-checking techniques can be used for the analysis of the following three classes of properties:

– *Connectivity properties.* Connectivity properties refer to the graph structure of a Web site.
– *Frame properties.* Since each link loads only a portion of a frame-based page, the content of a frame-based page depends on the path followed by the browser in a site: hence, frame properties are essentially path properties.
– *Cost properties.* Cost properties refer to the number of links or bytes, that must be followed or downloaded while browsing a Web site. An example consists in the computation of the all-pair longest path in a Web site.

Model-checking methods can be broadly classified in *enumerative* methods and *symbolic* methods. Enumerative methods operate on *states* as the basic entities [4, 17], and represent sets of states and transition relations in terms of the individual states. Symbolic methods operate directly on symbolic representations of sets of states [3, 2]. Our approach to the model checking of the Web is enumerative, in that we represent sets of Web pages as collections of single pages. However, we argue that it is convenient to phrase our model-checking algorithms as symbolic algorithms, based on the manipulation of sets of Web pages. In fact, a set-based approach lends itself better to parallelization: given a set S of Web pages, the computation of the set $\mathit{Post}(S)$ of Web pages that can be reached from S by following one link can proceed largely in parallel, by following simultaneously all links originating from pages in S. Since the page fetch time on the Web is typically dominated by response time, rather than transfer time, such a parallel approach is significantly more efficient than a sequential one. Nevertheless, the model checking of the Web differs in several respects from usual symbolic model checking. In particular, some of the basic operations performed by standard model-checking methods cannot be performed on the Web:

1. Given a predicate P defining a property of Web pages, we cannot construct the set S_P consisting of all the Web pages that satisfy P.
2. Given a set S of Web pages, we cannot construct the set $\mathit{Pre}(S)$ of pages that can reach some page of S by following one link.[1]
3. The set V of all Web pages is not known in advance. Likewise, given a set $U \subseteq V$ of Web pages, we cannot construct its complement $V \setminus U$.

These limitations imply in particular that we cannot phrase our model-checking algorithms in standard μ-calculus [15, 10]. In fact, limitation 3 implies that we cannot evaluate expressions that involve the greatest fixpoint operator ν: in $\nu x.\phi(x)$, we cannot set $x_0 = V$ in order to compute the limit $\lim_{k\to\infty} x_k$ of $x_{k+1} = \phi(x_k)$, for $k \geq 0$. Limitation 1 implies that we must introduce restrictions in the use of predicates, and Limitation 2 prevents us from using the standard predecessor operator *Pre*.

[1] Search engines such as Google do in fact provide such a service, but the answer they provide is only approximate.

We introduce *constructive μ-calculus*, a fixpoint calculus similar to equational μ-calculus [1], but containing only expressions and constructs that can be effectively evaluated within the above limitations. Constructive μ-calculus differs from standard equational μ-calculus in the following respects:

– The greatest fixpoint operator ν is replaced by the operator ν_x, where x is a set of states that must be already known, and that acts as the "universe set" in which the largest fixpoint is computed.
– The predecessor operator *Pre* is replaced by its guarded version $\mathit{GPre}(U, W)$, defined by $\mathit{GPre}(U, W) = U \cap \mathit{Pre}(W)$. Since the pages in U are already known when $\mathit{GPre}(U, W)$ is evaluated, all links from U to W are also known, ensuring that $\mathit{GPre}(U, W)$ can be computed.
– Predicates cannot be used to generate sets of states, but only to select from existing sets the states that satisfy given propositional formulas.

We show that these restrictions are enough to ensure that the expressions of constructive μ-calculus are effectively computable, and we provide a precise characterization of the expressive power of constructive μ-calculus. In particular, we show that in spite of the above differences, the expressive power of constructive μ-calculus is essentially the same as the one of ordinary μ-calculus. We phrase our Web model-checking algorithms in constructive μ-calculus.

We argue that the limitations 1–3 are not peculiar to the Web, but are shared by a large class of systems that have a large or infinite state space without regular structure, among which software programs. In the analysis of complex programs, we often have no way of constructing in advance the set of all states, and we may not know what are the predecessors of a given set of states unless we have already encountered those states in the course of the model checking. Constructive μ-calculus is well-suited for phrasing model-checking algorithms that operate on-the-fly over irregular graphs that can be explored only gradually.

In order to experiment with Web model checking, we have implemented the model checker MCWEB, which enables the analysis of connectivity, frame, and cost properties of Web sites. We report our experience in using MCWEB, and we summarize the most common classes of errors that we were able to find using it.

2 The Graph Structure of the Web

As a first step in the application of model-checking techniques to the Web consists in fixing a graph structure of the Web. The simplest choice consists in disregarding the frame structure of the Web, and in modeling the Web as a graph of pages, with links due to both `a` (anchor) and `frame` (sub-frame) tags as edges. We call this the *flat* model of the Web. The flat model suffices for many purposes, among which broken link detection and HTML consistency analysis, and indeed many current tools for Web analysis rely on the flat model. Nevertheless, some reachability properties cannot easily be checked on the flat model. For example, the property that the home page of a site is reachable from all pages of the site is often not true in the flat model of a frame-based site, since the link to the home page may be in a separate frame (and thus, a separate graph node) than the main content of the page. For this reason, our graph model of the Web takes into account the frame structure of the pages, unlike the flat model.

2.1 URLpages and webnodes

An *URL* a is a string that uniquely specifies a document on the Web; it is composed of a protocol field (such as HTTP), a domain name, and a document locator on the domain. In this paper, we restrict our attention to URLs that refer to the HTTP protocol. Given an URL a, we can fetch the corresponding document $s = \mathit{GetUrl}(a)$; we call s the *URLpage* corresponding to the URL a. The URLpage $s = \langle g_s, h_s, F_s, A_s \rangle$ consists in the URL g_s from which the document is retrieved, the textual content h_s, a set of *frame tags* F_s, and a set of *anchor tags* A_s. The URL g_s may be different from s due to automatic redirection, as effected by the HTTP protocol. Since images are loaded automatically by most current browsers, we consider them to be an integral part of h_s, even though they are specified by separate anchors. A *frame tag* $\langle b, \ell \rangle$ consists of the URL b to be loaded into the subframe, and of a name ℓ used to label the subframe. An *anchor tag* $\langle b, \ell \rangle$ consists of the URL b specifying the link destination, and of a *target name* ℓ, which specifies in which subframe the new URL should be loaded [16, 18] (if no target is specified, we take ℓ to be the empty string ε). While this is only a partial subset of the tags and attributes that occur in HTML documents, it will suffice for our purpose of defining the graph structure of the Web.

The nodes of our graph model of the Web, called the *webgraph*, consist in *webnodes*. A webnode w is a tree with URLpages as nodes; the edges of the tree are labeled by frame names. We write $s \in w$ to denote that an URLpage s is a node in the tree w. If $s \in w$ and $F_s = \{\langle a_1, \ell_1 \rangle, \ldots, \langle a_n, \ell_n \rangle\}$, then the URLpage s has n URLpages $t_1, \ldots, t_n$ as children in w; for $1 \leq i \leq n$, the edge from s to t_i is labeled with ℓ_i. Given an URLpage t, the webnode $w = \mathit{WebNode}(t)$ is obtained by "loading" recursively all the subframes of t. Precisely, w consists in a tree with root t, where each URLpage $s \in w$ has as descendants the set $\{\mathit{GetUrl}(a) \mid \langle a, \ell \rangle \in F_t\}$ of URLpages corresponding to subframes of t. For brevity, given an URL a we define $\mathit{GetWeb}(a) = \mathit{WebNode}(\mathit{GetUrl}(a))$.

The edges of the webgraph correspond to page links; the precise definition takes into account the way in which pages are loaded into subframes. Given a webnode w, and an URLpage $s \in w$, we denote by $\mathit{subtree}(w, s)$ the subtree of w with root in s. Given a webnode w, an URLpage $s \in w$, and a link $\langle a, \ell \rangle \in A_s$, we denote by $\mathit{target}(w, s, \ell)$ the subtree of w that will be replaced by the webnode $\mathit{GetWeb}(a)$ when the link $\langle a, \ell \rangle$ is followed, defined according to the HTML standard [16, 18]. Precisely, if $\ell = \texttt{\_blank}$ or $\ell = \texttt{\_top}$, we have $\mathit{target}(w, s, \ell) = w$; if $\ell = \texttt{\_self}$ or $\ell = \varepsilon$ then $\mathit{target}(w, s, \ell) = \mathit{subtree}(w, s)$, if $\ell = \texttt{\_parent}$ then $\mathit{target}(w, s, \ell) = \mathit{subtree}(w, t)$, where t is the parent of s in the tree w. Finally, if ℓ is any other string, then $\mathit{target}(w, s, \ell) = \mathit{subtree}(w, t)$, where t is the unique URLpage such that the link in w from the parent of t to t is labeled ℓ; if there is no such link, or if the link is not unique, then we treat the link as a "broken link", and we take as its destination a special error webnode. Given a webnode w, a subtree u of w, and another webnode v, we denote by $w[v/u]$ the result of replacing u by v in w. Given a webnode w and an URL-

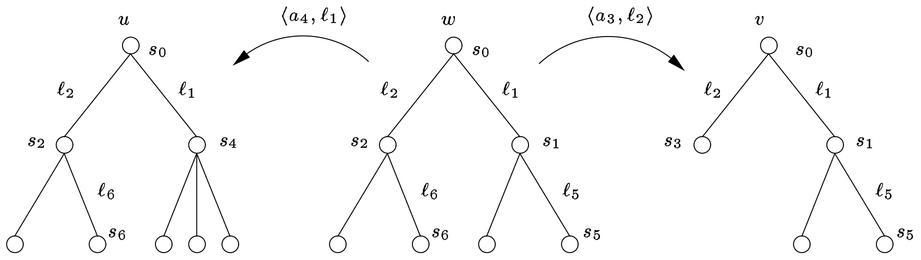

Fig. 1. A primary webnode $w = \mathit{WebNode}(s_0)$, and two secondary webnodes u, v.

page $s \in w$, the destination of a non-broken link $\langle a, \ell \rangle \in A_s$ consists in the webnode $\mathit{dest}(w, s, \langle a, \ell \rangle) = w[\mathit{GetWeb}(a)/\mathit{target}(w, s, \ell)]$.

Example 1 In Figure 1, we depict a webnode $w = \mathit{WebNode}(s_0)$, and two webnodes u, v reachable from w by following links. We have $F_{s_0} = \{\langle a_1, \ell_1 \rangle, \langle a_2, \ell_2 \rangle\}$; the children of s_0 in w are s_1, s_2: by convention, we denote $\mathit{GetUrl}(a_i) = s_i$ for all $i \geq 0$. Only some of the edge labels and URLpages are indicated, to avoid clutter. URLpage s_1 contains the anchor tag $\langle a_3, \ell_2 \rangle$; taking this link leads to webnode v. URLpage s_2 contains the anchor tag $\langle a_4, \ell_1 \rangle$; taking this link leads to webnode u. URLpage s_5 contains the anchor tag $\langle a_7, \ell_6 \rangle$. Note that the link corresponding to the anchor tag $\langle a_7, \ell_6 \rangle$ is broken in v, since there is no label ℓ_6 in v; the link is not broken in w, and it is not present in u. This illustrates how links can become broken in secondary pages.

2.2 The webgraph

In order to fix the structure of the webgraph, we need to establish a criterion for webnode equality. Two webnodes w and u are equal, written $w \cong u$, if their trees of URLpages are equal: hence, webnode equality is defined in terms of URLpage equality. There are several possible definitions for URLpage equality; to understand the issue, we need to explain in more detail how URLpages are fetched from the Web. Given an URL a_0, we can issue an HTTP request for a_0. The result can either be the URLpage $\mathit{GetUrl}(a_0)$, or a redirection URL a_1. In the latter case, we can issue a page request for a_1, obtaining either $\mathit{GetUrl}(a_1)$, or a redirection URL a_2. The process continues until either an error occurs, or until we reach a $k \geq 0$ such that a request for a_k returns an URLpage s; we set then $\mathit{GetUrl}(a_0) = \cdots = \mathit{GetUrl}(a_k) = s$. Consider two sequences $a_0, \ldots, a_k$, $\mathit{GetUrl}(a_0)$ and $b_0, \ldots, b_n$, $\mathit{GetUrl}(b_0)$ of redirections and final pages, and let $s = \mathit{GetUrl}(a_0)$ and $t = \mathit{GetUrl}(b_0)$; note that we have $g_s = a_k$, and $g_t = b_n$. We can define whether s is equal to t, written $s \cong t$, in several ways.

- **Textual comparison.** We can define $s \cong t$ when $h_s = h_t$, i.e., when the texts of the two URLpages s and t are identical. According to this definition, however, different domains containing two textually identical pages (for instance, two empty pages) would share a webnode in the webgraph, leading to unexpected results when reachability analysis is performed. In addition, textual comparison is sensitive to minor differences in the pages retrieved, such as visitation counter updates.

– **Final URL comparison.** Another possibility consists in defining $s \cong t$ when $a_k = b_n$, or equivalently when $g_s = g_t$. Occasionally, however, a request to an URL a is redirected to any of a large number of URLs $c_1, \ldots, c_m$, in order either to distribute the load between machines, or to provide slightly different content in terms of advertising. Final URL comparison would consider *GetUrl*$(c_1) \neq \cdots \neq$ *GetUrl*(c_m), in spite of the fact that those pages are essentially the same page.
– **Redirection sequence comparison.** Finally, we can define $s \cong t$ when $\{a_0, \ldots, a_k\} \cap \{b_0, \ldots, b_n\} \neq \emptyset$; this criterion is more robust than final URL comparison with respect to load-balancing and page-customization techniques.

The Web checker MCWEB adopts redirection sequence comparison as the URLpage equality criterion, with additional heuristics used to cope with features such as automatic `index.html` extensions.

Once a notion $\cong$ of webnode equality has been fixed, we can define precisely the webgraph. Let V be the set of all webnodes, and let $E = \{(w, u) \mid w \in V \wedge \exists s \in w. \exists \langle a, \ell \rangle \in A_s. u = \mathit{dest}(w, s, \langle a, \ell \rangle)\}$ be the set of all edges between webnodes. The *webgraph* $(V/\cong, E/\cong)$ is the quotient of (V, E) with respect to the equality notion $\cong$. We note that this definition is not completely precise, as it depends on the function *GetUrl*, that given an URL returns the corresponding URL. This definition also does not capture the fact that the true connectivity Web is time-varying. Nevertheless, this definition formalizes the structure of the Web to a sufficient degree for the development of our model-checking algorithms.

We say that a webnode $w \in V$ is *primary* if there is an URL a such that $w =$ *GetWeb*(a), and that w is *secondary* otherwise. Primary webnodes correspond to Web pages that can be obtained by loading an URL with a browser. Secondary webnodes cannot be loaded directly; they are reached by traversing links and updating the frame structure starting from primary webnodes. Most current tools for Web analysis only consider primary webnodes. Yet, many errors arise only in secondary webnodes, as illustrated by Example 1. Our experience with MCWEB indicates that the difficulty of examining all secondary webnodes is a common cause of errors on the Web.

3 Model Checking the Web

As remarked in the introduction, the ordinary μ-calculus is not suited for the model checking of the Web, since it includes several operations that are not effectively computable on the Web. We introduce constructive μ-calculus, a fixpoint calculus similar to μ-calculus, but containing only expressions that can be effectively computed.

3.1 Constructive μ-calculus

Syntax. Constructive μ-calculus ($C\mu C$) is derived from the equational μ-calculus of [1]. A $C\mu C$ formula $\langle\langle B_1, \ldots, B_n\rangle, x_m\rangle$ consists of $n > 0$ *blocks* $B_1, \ldots, B_n$, and of an *output variable* x_m, with $m \in \{1, \ldots, n\}$. Each block B_i, for $1 \leq i \leq n$, has the form $\lambda_i . x_i = e_i$, where x_i is a variable, e_i a *set expression*, and λ_i is a quantifier tag equal to either μx_i, or to $\nu x_i \subseteq x_j$ for some $j > i$. Hence, the quantifier tag of the outermost block B_n must be μx_n. Each set expression e_i is defined according to the following grammar:

$$\begin{aligned} \Phi \; ::= \; & x \mid \Phi \cup \Phi \mid \Phi \cap \Phi \mid \Phi \setminus \Phi \mid \Phi \cap \Theta \mid a \\ & \mid \mathit{Post}(\Phi) \mid \widetilde{\mathit{GPost}}(\Phi, \Phi) \mid \mathit{GPre}(\Phi, \Phi) \mid \widetilde{\mathit{GPre}}(\Phi, \Phi) \end{aligned}$$

where a is a constant, x is one of $x_1, \ldots, x_n$, and Θ is a *predicate expression.* Predicate expressions are defined by the grammar

$$\Theta ::= \Theta \vee \Theta \mid \neg\Theta \mid P,$$

where P is a predicate belonging to some basic set of predicates $\mathcal{P}$. The use of the set difference operator in set expressions is subject to the following restriction. For $i, j \in \{1, \ldots, n\}$, we say that the block B_i depends directly on block B_j, written $B_i \succ B_j$, if x_j appears in e_i, and we let $\overset{*}{\succ}$ be the reflexive transitive closure of $\succ$. For $i, j \in \{1, \ldots, n\}$, we say that that the variable x_j occurs with negative polarity in e_i if it occurs within an odd number of right-hand sides of the set-difference operator $\setminus$. Then, we require that for all $i, j \in \{1, \ldots, n\}$, the variable x_j occurs with negative polarity in e_i only when $B_i \overset{*}{\not\succ} B_j$. We say that a $C\mu C$ formula is *negation-free* if it does not contain occurrences of the set difference operator $\setminus$. We denote by $C\mu C^+$ the negation-free fragment of $C\mu C$.

Syntax of ordinary equational μ-calculus. In order to compare the expressive power of constructive and ordinary μ-calculus, we define also the semantics of the equational μ-calculus of [1], denoted by μC. The formulas of μC have the same structure of those of $C\mu C$, except that the quantifier tag λ_i can be equal to either μx_i, or to νx_i. The syntax of set expressions is given by the grammar

$$\Phi ::= x \mid \Phi \cup \Phi \mid \Phi \cap \Phi \mid \Theta \mid \neg\Theta \mid a \mid \mathit{Post}(\Phi) \mid \widetilde{\mathit{Post}}(\Phi) \mid \mathit{Pre}(\Phi) \mid \widetilde{\mathit{Pre}}(\Phi)$$

Semantics. For conciseness, we define the semantics of a calculus that is a superset of both $C\mu C$ and μC; the semantics of $C\mu C$ and μC are obtained by considering the appropriate fragments of this calculus. The semantics is defined with respect to a *Kripke structure* $\mathcal{K} = (V, E, \mathcal{C}, f^c, \mathcal{P}, f^p)$, where (V, E) is a graph, $\mathcal{C}$ is a set of constants, $f^c : \mathcal{C} \mapsto V$ is the interpretation of the constants, $\mathcal{P}$ is a set of predicates, and $f^p : \mathcal{P} \mapsto 2^V$ is the interpretation of the predicates. In the model checking of the Web, we take V, E as in the webgraph, $\mathcal{C}$ to be the set of valid URLs, f^c to be *GetWeb*, $\mathcal{P}$ to be a set of effectively checkable predicates defined on webnodes, and $f^p(P) = \{w \in W \mid w \models P\}$ for all $P \in \mathcal{P}$. Given such a Kripke structure, all the operators in set and predicate expressions have their standard meanings, except for the predecessor and successor operators. The semantics of the predecessor and successor operators is defined, for all $U, W \subseteq V$, by

$$\begin{aligned} \mathit{Pre}(W) &= \{u \in V \mid \exists v \in W.(u, v) \in E\} & \mathit{GPre}(U, W) &= U \cap \mathit{Pre}(W) \\ \widetilde{\mathit{Pre}}(W) &= \{u \in V \mid \forall v.((u, v) \in E \rightarrow v \in W)\} & \widetilde{\mathit{GPre}}(U, W) &= U \cap \widetilde{\mathit{Pre}}(W) \end{aligned}$$

$$Post(W) = \{u \in V \mid \exists v \in W.(v,u) \in E\}$$

$$\widetilde{Post}(W) = \{u \in V \mid \forall v.((v,u) \in E \rightarrow v \in W)\} \qquad \widetilde{GPost}(U,W) = U \cap \widetilde{Post}(W)$$

The intuition is that in $C\mu C$ we can compute the set of predecessors of a given set W of webnodes only relative to another set U of webnodes; similarly for the other constructive operators. The operational semantics of constructive μ-calculus, given by Algorithm 1 below, will ensure that all the webnodes in U have already been explored when $GPre(U,W)$ is computed (resp. $\widetilde{GPre}(U,W)$, $\widetilde{GPost}(U,W)$), thus ensuring that all the links from U to W are known.

The definition of semantics follows the lines of [1]. Consider a Kripke structure $\mathcal{K}$ and a formula $\langle\langle B_1,\ldots,B_n\rangle, x_m\rangle$ of $C\mu C$, where each block B_i, $1 \le i \le n$, has the form $\lambda_i.x_i = e_i$, for λ_i equal to either μx_i or $\nu x_i \subseteq x_j$. Let $\Gamma = (\{x_1,\ldots,x_n\} \mapsto 2^V)$ be the set of valuations of the variables in the formula. Given $\gamma \in \Gamma$ and $1 \le i \le n$, we indicate with $\gamma \circ (x_i = U)$ the valuation that coincides with γ, except that it associates value $U \subseteq V$ to x_i. Given a valuation γ and a set expression e_i, for $1 \le i \le n$, we denote by $[\![e_i \mid \mathcal{K},\gamma]\!] \subseteq V$ the value of e_i in the Kripke structure $\mathcal{K}$ under valuation γ. Given $\gamma \in \Gamma$ we define recursively, for $i = 0$ to n, the valuation $Eval^i_{\mathcal{K}}(\langle B_1,\ldots,B_i\rangle \mid \gamma) \in \Gamma$. The definition relies on two auxiliary functions $f^i_{\mathcal{K},\gamma}, g^i_{\mathcal{K},\gamma} : \Gamma \mapsto \Gamma$, defined as follows:

$$g^i_{\mathcal{K},\gamma}(\delta) = Eval^{i-1}_{\mathcal{K}}(\langle B_1,\ldots,B_{i-1}\rangle \mid \delta) \circ (x_{i+1} = \gamma(x_{i+1})) \circ \cdots \circ (x_n = \gamma(x_n))$$

$$f^i_{\mathcal{K},\gamma}(\delta) = g^i_{\mathcal{K},\gamma}(\delta) \circ (x_i = [\![e_i \mid \mathcal{K}, g^i_{\mathcal{K},\gamma}(\delta)]\!]) \qquad \textit{if } \lambda_i \textit{ is } \mu x_i \textit{ or } \nu x_i$$

$$f^i_{\mathcal{K},\gamma}(\delta) = g^i_{\mathcal{K},\gamma}(\delta) \circ (x_i = \gamma(x_j) \cap [\![e_i \mid \mathcal{K}, g^i_{\mathcal{K},\gamma}(\delta)]\!]) \qquad \textit{if } \lambda_i \textit{ is } \nu x_i \subseteq x_j$$

We then define $Eval^i_{\mathcal{K}}(\langle B_1,\ldots,B_i\rangle \mid \gamma) = \lambda\delta.(\delta = f^i_{\mathcal{K},\gamma}(\delta))$, where $\lambda = \mu$ if λ_i is μx_i, and $\lambda = \nu$ if λ_i is $\nu x_i \subseteq x_j$. The restrictions on the use of negation, together with the Tarski-Knaster theorem [22], ensure that the fixpoints exist. It can be readily verified that $Eval^n_{\mathcal{K}}(\langle B_1,\ldots,B_n\rangle \mid \gamma)$ does not depend on γ. The meaning of the complete formula is the valuation of the output variable: we define $[\![\langle\langle B_1,\ldots,B_n\rangle, x_m\rangle]\!]_{\mathcal{K}} = Eval^n_{\mathcal{K}}(\langle B_1,\ldots,B_n\rangle \mid \gamma)(x_m)$, for an arbitrary $\gamma \in \Gamma$.

3.2 Expressivity

In order to study the relationship between the expressive power of $C\mu C$ and μC, we consider fixed infinite and countable sets $\mathcal{P}$ and $\mathcal{C}$ of predicates and constants, so that the syntax of the formulas is fixed. Given a class $\mathcal{U}$ of Kripke structures, let $\mathcal{V} = \bigcup\{V \mid (V,E,\mathcal{C},f^c,\mathcal{P},f^p) \in \mathcal{U}\}$ be the set of all states. A formula ϕ of fixpoint calculus defines a function $[\![\phi]\!] : \mathcal{U} \mapsto 2^{\mathcal{V}}$ by $[\![\phi]\!](\mathcal{K}) = [\![\phi]\!]_{\mathcal{K}}$. Given $\mathcal{U}$ and two fixpoint calculi C and C', we say that C is *as expressive* as C' over $\mathcal{U}$, written $C \sqsupseteq_{\mathcal{U}} C'$, if for every formula ϕ' of C' there is a formula ϕ of C such that $[\![\phi]\!]$ and $[\![\phi']\!]$ are the same function. We say that C and C' are *equally expressive* over $\mathcal{U}$, written $C \equiv_{\mathcal{U}} C'$, if both $C \sqsupseteq_{\mathcal{U}} C'$ and $C \sqsubseteq_{\mathcal{U}} C'$ hold, and we say that C is *strictly more expressive* than C' over $\mathcal{U}$, written $C \sqsupset C'$, if $C \sqsupseteq_{\mathcal{U}} C'$ holds but $C \sqsubseteq_{\mathcal{U}} C'$ does not. Let $\mathcal{K}_{\mathit{fin}}$ and $\mathcal{K}_{\mathit{cnt}}$ be the classes of Kripke structures with finite and countable state space, respectively. The following theorem relates the expressive power of μC and $C\mu C^+$.

Theorem 1 *$\mu C \sqsupset_{\mathcal{K}_{\mathit{fin}}} C\mu C^+$, and $\mu C \sqsupset_{\mathcal{K}_{\mathit{cnt}}} C\mu C^+$.*

The difference in expressive power is essentially due to the inability of $C\mu C$ of considering portions of the Kripke structure that are unreachable from named constants. This is confirmed by the following result. We say that a class $\mathcal{U}$ of Kripke structures is *finitely rooted* if there is a finite set of constants $\{a_1, \ldots, a_n\}$ such that for all $(V, E, \mathcal{C}, f^c, \mathcal{P}, f^p) \in \mathcal{U}$, we have that every state of V is reachable in (V, E) from $f^c(a_1) \cup \cdots \cup f^c(a_n)$ in a finite number of steps.

Theorem 2 *For all classes $\mathcal{U}$ of finitely-rooted Kripke structures, we have $\mu C \equiv_{\mathcal{U}} C\mu C^+$.*

Proof. There is a straightforward translation of $C\mu C^+$ into μC. The translation from μC to $C\mu C^+$ is as follows. Consider a μC expression $\langle B_1, \ldots, B_n, x_m \rangle$, where for $1 \leq i \leq n$ the block B_i has the form $\lambda x_i.x_i = e_i$, for $\lambda \in \{\mu, \nu\}$. An equivalent $C\mu C$ expression is $\langle B'_1, \ldots, B'_n, B'_{n+1}, x_m \rangle$, where the block B'_{n+1} is $\mu y.y = \mathit{GetWeb}(a_1) \cup \cdots \cup \mathit{GetWeb}(a_n) \cup \mathit{Post}(y)$, and for $1 \leq i \leq n$, the block B'_i is obtained from B_i by replacing νx_i with $\nu x_i \subseteq y$ and $\mathit{Pre}(\Phi)$ with $\mathit{GPre}(y, \Phi)$, $\widetilde{\mathit{Pre}}(\Phi)$ with $\widetilde{\mathit{GPre}}(y, \Phi)$, and $\widetilde{\mathit{Post}}(\Phi)$ with $\widetilde{\mathit{GPost}}(y, \Phi)$. ∎

The converse is also true, under some general conditions. We say that a Kripke structure is *non-trivial* if it contains at least one predicate symbol.

Theorem 3 *Consider a class $\mathcal{U}$ of non-trivial Kripke structures. If $\mu C \equiv_{\mathcal{U}} C\mu C^+$, then all the structures in $\mathcal{U}$ are finitely rooted.*

The following result follows from the presence of the set-difference operator $\setminus$ in the definition of $C\mu C$, and can be proved similarly to Theorem 2.

Theorem 4 *On finitely-rooted Kripke structures, the fixpoint calculus $C\mu C$ is closed under complementation.*

The difference in expressive power between $C\mu C$ and μC on finitely-rooted structures is due to the fact that μC is not closed under complementation. Let μC^D be the calculus obtaining by adding to μC the operator *DGetWeb*, applicable only to constants, with semantics defined by $\mathit{DGetWeb}(a) = \{w \in V \mid w \neq f^c(a)\}$. The calculus μC^D is then closed under complementation, leading to the following theorem.

Theorem 5 *The following assertions hold.*

1. *$\mu C^D \sqsupset_{\mathcal{K}_{\mathit{fin}}} C\mu C$, and $\mu C^D \sqsupset_{\mathcal{K}_{\mathit{cnt}}} C\mu C$.*
2. *For all classes $\mathcal{U}$ of finitely-rooted Kripke structures, we have $\mu C^D \equiv_{\mathcal{U}} C\mu C$.*

3.3 Evaluation of constructive μ-calculus formulas

While μC and $C\mu C$ have similar expressive power, the calculus $C\mu C$ guarantees that, whenever the interpretations of the variables at the fixpoint consists in finite sets, then the fixpoint itself can be computed in finite time. An algorithm for doing so is given below.

Algorithm 1 **Input:** a Kripke structure $\mathcal{K} = (V, E, \mathcal{C}, f^c, \mathcal{P}, f^p)$, and a $C\mu C$ formula $\phi = \langle\langle B_1, \dots, B_n\rangle, x_m\rangle$, where each block B_i, $1 \leq i \leq n$, has the form $\lambda_i.x_i = e_i$.
Output: $[\![\phi]\!]_{\mathcal{K}}$.
Procedure: Let $\Gamma = \{x_1, \dots, x_n\} \mapsto 2^V$. Given a valuation $\gamma \in \Gamma$, we define recursively, for $i = 0$ to n, the valuation *Compute*$(\langle B_1, \dots, B_i\rangle \mid \gamma) \in \Gamma$. For $i = 0$, we let *Compute*$(\langle\rangle \mid \gamma) = \gamma$. For $i > 0$, the definition is as follows.

```
  Init: If $\lambda_i$ is $\mu x_i$, then let $\gamma'_0 = \gamma \circ (x_i = \emptyset)$; if $\lambda_i$ is $\nu x_i \subseteq x_j$, then let
      $\gamma'_0 = \gamma \circ (x_i = \gamma(x_j))$.
  Update: For $k \geq 0$, let $\gamma''_k = Compute(\langle B_1, \dots, B_{i-1}\rangle \mid \gamma'_k)$,
      $W_k = [\![e_i \mid \mathcal{K}, \gamma''_k]\!]$, and $\gamma'_{k+1} = \gamma''_k \circ (x_i = W_k)$.
  Define: $Compute(\langle B_1, \dots, B_i\rangle \mid \gamma) = \lim_{k\to\infty} \gamma'_k$.
```

Return: *Compute*$(\langle B_1, \dots, B_n\rangle \mid \gamma)(x_m)$, where γ is arbitrary.

The following theorem states that the fixpoints of $C\mu C$, if finite, can be effectively computed. We say that an operation can be *effectively computed* if it involves only finitely many states of the Kripke structure.

Theorem 6 *Consider a $C\mu C$ formula $\langle\langle B_1, \dots, B_n\rangle, x_m\rangle$, and assume that for a variable interpretation γ, we have $|Eval^n_{\mathcal{K}}(\langle B_1, \dots, B_n\rangle \mid \gamma)(x_i)| < \infty$ for all $1 \leq i \leq n$. Then, Algorithm 1 consists of effectively computable steps, and it terminates returning $[\![\langle\langle B_1, \dots, B_n\rangle, x_m\rangle]\!]_{\mathcal{K}}$.*

The result is a consequence of the fact that, if all variables have finite extension at the fixpoint, then only a finite portion of the Kripke structure is explored. Note that the result is independent from the cardinality of V. In contrast, it is well known that when V is infinite, the formulas of μC cannot be evaluated iteratively, even when the fixpoints are finite.

3.4 Predicates for Web analysis

After some experimentation, we have chosen to include in the Web model checker MCWEB the following families of predicates, for all strings α, domains Δ, and $k > 0$:

- predicate $\texttt{contains}_{\alpha_1,\alpha_2,\dots,\alpha_k}$ holds for a webnode w if there is an URLpage $s \in w$ such that h_s contains all the strings $\alpha_1, \dots, \alpha_k$.
- predicate $\texttt{in\_domain}_{\Delta}$ holds for a webnode w if there is an URLpage $s \in w$ such that g_s contains the substring Δ;
- predicate $\texttt{all\_in\_domain}_{\Delta}$ holds for a webnode w if all URLpages $s \in w$ are such that g_s contains the substring Δ;
- predicate $\texttt{http\_error}_k$ holds for a webnode w if the HTTP error k occurred while loading some URLpage in w;
- predicate `frames_error` is a catch-all predicate, that holds for a webnode w if the frame structure at w contains errors. Among the errors currently checked are: duplicated frame names (a name ℓ that occurs in more than one frame tag), frame trees deeper than a fixed threshold, and non-existent link targets (anchors tags $\langle a, \ell\rangle$ such that ℓ does not appear in any frame tag).

3.5 A semi-decision procedure for non-emptiness

Consider a $C\mu C^+$ formula $\phi = \langle\langle B_1, \ldots, B_n\rangle, x_m\rangle$, where each block B_i has the form $\lambda_i.x_i = e_i$, for $1 \leq i \leq n$. During the evaluation of ϕ according to Algorithm 1, we call *checkpoints* the stages where all the variables x_i with quantifier tag equal to $\nu x_i \subseteq x_j$ for some $j > i$ have reached a fixpoint (even if some variables with μ-tag have not). Then, if the interpretation of the output variable x_m at some checkpoint is non-empty, we know that also $[\![\phi]\!]_{\mathcal{K}} \neq \emptyset$. To make this observation precise, we consider a $C\mu C^+$ formula $\phi = \langle\langle B_1, \ldots, B_n\rangle, x_m\rangle$, and we let $\{i \in \{1, \ldots, n\} \mid \lambda_i \text{ is } \mu x_i\} = \{i_1, \ldots, i_j\}$ be the set of indices of the μ-blocks in ϕ; we denote by $\sharp\mu(\phi) = j$ the number of such indices. Given $k_1, k_2, \ldots, k_j \geq 0$, we compute $[\![\phi]\!]_{\mathcal{K}}^{k_1,\ldots,k_j}$ by following Algorithm 1, except that for $1 \leq l \leq j$ we take *Compute*$(\langle B_1, \ldots, B_{i_l}\rangle \mid \gamma)_{i_1,\ldots,i_l} = \gamma'_{k_l}$. Hence, $[\![\phi]\!]_{\mathcal{K}}^{k_1,\ldots,k_j}$ is computed by performing $k_1, k_2, \ldots, k_j$ iterations of the μ-blocks, rather than waiting until the fixpoint is reached.

Theorem 7 *For all $C\mu C^+$ formulas ϕ and Kripke structures $\mathcal{K}$, if $[\![\phi]\!]_{\mathcal{K}}^{k_1,\ldots,k_{\sharp\mu(\phi)}} \neq \emptyset$ for some $k_1, \ldots, k_{\sharp\mu(\phi)} \geq 0$, then $[\![\phi]\!]_{\mathcal{K}} \neq \emptyset$.*

Given a $C\mu C^+$ formula ϕ and a (possibly infinite) finitely-branching Kripke structure $\mathcal{K}$, this theorem provides a semi-decision procedure for $[\![\phi]\!]_{\mathcal{K}} \neq \emptyset$: it suffices to enumerate the lists of non-negative integers $\langle k_1, \ldots, k_{\sharp\mu(\phi)}\rangle$, checking for each list whether $[\![\phi]\!]_{\mathcal{K}}^{k_1,\ldots,k_{\sharp\mu(\phi)}} \neq \emptyset$. As an example, consider the formula

$$\phi = \langle\langle \nu y \subseteq x.y = \widetilde{Pre}(y), \mu x.x = \texttt{in\_domain}_{\Delta} \cap (Post(x) \cup a)\rangle, y\rangle,$$

where Δ is a domain name, and a is an URL in that domain. If $\mathcal{K}$ is the webgraph, then $[\![\phi]\!]_{\mathcal{K}}$ is the set of webnodes in domain Δ that are reachable from the URL a, and that have no link sequence leading outside Δ. The variable x keeps track of the portion of Δ that is reachable from a. If the domain Δ contains infinitely many webnodes (as can be the case in sites with dynamically generated pages), then the evaluation of $[\![\phi]\!]_{\mathcal{K}}$ does not terminate. On the other hand, we can obtain a semi-decision procedure for $[\![\phi]\!]_{\mathcal{K}} \neq \emptyset$ by evaluating $[\![\phi]\!]_{\mathcal{K}}^{k}$ for $k = 0, 1, 2, \ldots$, and by checking for non-emptiness for each value of k. This provides a semi-decision procedure for detecting pages in a Web site that cannot reach the rest of the Web.

4 Web Model Checking in Practice

In order to experiment with Web model checking, we have implemented the model checker MCWEB. The checker MCWEB is written in Python; its input consists in constructive μ-calculus formulas, augmented with the capability of post-processing the output in order to perform quantitative analysis of Web properties.

In some domains, such as hardware, the cost of errors that go undetected until the production stage is very large, and consequently a large effort is done in order to detect them early. Formal verification methods such as model checking are usually called to help in finding error that cannot be found with other methods. Consequently, finding errors with formal methods is a difficult task. In Web model checking, the situation is very different. Due to the lower cost of undetected errors on the Web, many collections of Web pages are checked cursorily, if at all, and in our experience errors are abundant, and come in great variety. The sites where we found the highest density of errors were medium-sized sites: small sites often have a simple structure that limits the number of errors; large commercial sites are usually produced with the help of automated tools, that help in avoiding structural errors. Nevertheless, errors were found even in large sites such as `amazon.com`.

In the course of the experimentation with MCWEB, we have identified some categories of errors and properties that are commonly of interest.

- **Broken links.** Detecting broken links is an ability that MCWEB shares with many other tools. MCWEB implicitly checks for broken links whenever the *Post* operator is applied to a set of webnodes.
- **Duplicated frame names.** MCWEB checks automatically for ill-formed frame structures using the predicate `frames_error` described earlier. For example, to check that no webnode with ill-formed frame structure is present in the site $\varDelta$ with home page a, it suffices to evaluate the formula $\langle\langle \mu x.x = a \cup (\mathit{Post}(x) \cap \texttt{in\_domain}_{\varDelta}), \mu y.y = x \cap \texttt{frames\_error}\rangle, y\rangle$.
- **Non-hierarchical frame content.** If an URLpage t of webnode w is not in the same domain as the root URLpage s of w, then the content and the links in t are typically not under the control of the author of s. Moreover, if s can be reached from t, then this usually leads to a webnode containing two instances of the same URLpage s. We can check that all webnodes in domain $\varDelta$ are composed only of URLpages in $\varDelta$ by evaluating the formula $\langle\langle \mu x.x = a \cup (\mathit{Post}(x) \cap \texttt{in\_domain}_{\varDelta}), \mu y.y = x \cap \neg\texttt{all\_in\_domain}_{\varDelta}\rangle, y\rangle$, where a is the home page of $\varDelta$.
- **Reachability.** Suppose that A is a set of webnodes containing publicly available information, B is a set of webnodes with private content, and C is a set of access control webnodes, We can check that all paths from A to B in domain $\varDelta$ contain a webnode in C by checking the emptiness of the formula $\langle\langle \mu y.y = (x \cap A) \cup (\mathit{Post}(y) \cap \texttt{in\_domain}_{\varDelta} \cap \neg C), \mu x.x = a \cup (\mathit{Post}(x) \cap \texttt{in\_domain}_{\varDelta}), \mu z.z = y \cap B\rangle, z\rangle$, where we assume that A, B, C are definable in terms of predicate `contains`, and a is the home page of $\varDelta$.
- **Repeated reachability.** To compute which pages of a Web site $\varDelta$ cannot reach the home page a without leaving $\varDelta$, we can evaluate the formula $\langle\langle \mu z.z = a \cup \mathit{Pre}(x, z), \mu x.x = a \cup (\mathit{Post}(x) \cap \texttt{in\_domain}_{\varDelta}), \mu y.y = x \setminus z\rangle, y\rangle$.
- **Longest paths.** MCWEB also contains an extension that enables the computation of the longest and shortest paths in a set of webnodes, where the "length" of a path consists in the number of bytes, or the number of links, that must be downloaded in order to follow it. For example, to find the all-pair longest path between webnodes of a domain $\varDelta$, MCWEB post-processes the output of the formula $\langle\langle \mu x.x = a \cup (\mathit{Post}(x) \cap \texttt{in\_domain}_{\varDelta})\rangle, x\rangle$. The computation of the all-pair longest path can provide information about the bottlenecks in the navigation of a site.

Acknowledgments. I would like to thank Tom and Monika Henzinger, Jan Jannink, and Freddy Mang for many helpful discussions and suggestions.

References

1. G. Bhat and R. Cleaveland. Efficient model checking via the equational μ-calculus. In *Proc. 11th IEEE Symp. Logic in Comp. Sci.*, pages 304–312, 1996.
2. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th IEEE Symp. Logic in Comp. Sci.*, pages 428–439. IEEE Computer Society Press, 1990.
3. J.R. Burch, K.L. McMillan, E.M. Clarkes, and D.L. Dill. Sequential circuit verification using symbolic model checking. In *Proc. of the 27th ACM/IEEE Design Automation Conference*, pages 46–51, Orlando, FL, USA, June 1990.
4. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, volume 131 of *Lect. Notes in Comp. Sci.*, pages 52–71. Springer-Verlag, 1981.
5. Electronic Software Publishing Co. Linkscan. http://www.elsop.com/linkscan/.
6. Watchfire Co. Linkbot. http://www.watchfire.com/products/linkbot.htm.
7. R.T. Fielding. Maintaiing distributed hypertext infostructures: Welcome to MOMspider's web. In *Proceedings of First Intl. Conference on the World-Wide Web (WWW 94)*, 1994.
8. Voget Selbach Enterprises GmbH. Link tester. http://vse-online.com/link-tester/.
9. Tilman Hausherr. Link sleuth. http://home.snafu.de/tilman/xenulink.html.
10. T.A. Henzinger, O. Kupferman, and S. Qadeer. From *pre*historic to *post*modern symbolic model checking. In A.J. Hu and M.Y. Vardi, editors, *CAV 98: Computer-aided Verification*, Lecture Notes in Computer Science 1427, pages 195–206. Springer-Verlag, 1998.
11. Biggbyte Software Inc. Infolink. http://www.biggbyte.com/infolink/index.html.
12. Link Alarm Inc. Link alarm. http://www.linkalarm.com/.
13. NetMechanic Inc. Html toolbox. http://www.netmechanic.com/.
14. InContext. Web analyzer 2.0. http://www.incontext.com/WAinfo.html.
15. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.
16. C. Musciano and B. Kennedy. *HTML: The Definitive Guide*. O'Reilly & Associates, Inc., 1998. Third Edition.
17. J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symposium on Programming*, volume 137 of *Lect. Notes in Comp. Sci.*, pages 337–351. Springer-Verlag, 1981.
18. D. Raggett, A. Le Hors, and I. Jacobs. HTML 4.01 specification, 1999. W3C Recommendation 24 December 1999.
19. Internet Software Services. Theseus. http://www.matterform.com/theseus/.
20. IXActa Visual Software. Ixsite web analyzer. http://ixacta.com/products/ixsite/.
21. DACPro Computer Solutions. Webtester. http://awsd.com/scripts/webtester/.
22. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 25(2):285–309, 1955.