Run-time verification of web applications

Roberto Tonino

May 21st, 2025

Contents

1 Introduction

This report summarizes the paper "A formal approach for run-time verification of web applications using scope-extended LTL". The authors present a solution that uses finite automata, LTL and the model checker Spin to formally verify properties on web applications.

The paper begins explaining how to build the automata that are used to model the behaviour of the user in a web application. After that, the authors focus on LTL, presenting a new operator that allows formula writers to define an LTL formula scoped to a subset of states. Eventually, the authors conclude the paper showing empirical results, together with a prototype of a tool to apply all the steps described in the paper.

2 Definitions

Some definitions are now presented, which will help the reader understand the technical jargon discussed in the paper:

- *Web Application Under Test (WAUT)*: the web application taken in consideration in a particular definition, discussion, etc...
- *request*: string l that represents a web request performed by a WAUT
- *response*: tuple $\langle u, c, I, L, V \rangle$ which represents the response that the web server sent to the WAUT
  - $u = l$
  - c represents the status code of the response [2]
  - $I =$ "target" attribute of the forms contained in the response
  - $L =$ URLs of the links contained in the response
  - $V =$ <$v_1, \ldots, v_k$> vector where v_i is the valuation of the page attribute i
- *browsing session*: denoted RRS, it is a recorded sequence of request-response exchanges that a user performs when visiting a WAUT
- *local browsing session*: denoted RRS as well, it is a recorded sequence of request-response exchanges that a user performs in a single browser window or frame

3 Automata

In order to represent the behaviour of a user in a web application, the authors propose a *communicating-automata*-based model of the WAUT. An automaton represents the "journey" that a user takes when utilizing the WAUT: this journey is identified by the links that the user clicks or the forms that they submit, and the pages that are loaded subsequentially. For an easier understanding, the authors present an incremental approach to the communicating-automata model. A *single-display* application model is first proposed, then to be followed by a *multi-display* application model.

3.1 Single-display applications model

The automaton that represents a single-display application is built as follows.

Procedure 1. Convert a browsing session of a single-display application into an automaton.

1. the inactive state $s_0 = \langle u_0, c_0, I_0, L_0, V_0 \rangle$ is defined;
2. the set of states is defined by the set of *responses*, a response being $\langle u_i, c_i, I_i, L_i, V_i \rangle$
   (a) when only the links in two responses are different, the responses are mapped to the same state. The authors provide a proof that this compression does not alter the recorded behaviour of the WAUT;
3. the alphabet is built from the union of the requests (Req), the URIs associated with links in the observed responses (Γ), and the actions that correspond to the unexplored forms in the observed responses Δ. $\Sigma = Req \cup \Gamma \cup \Delta$;
4. there is a transition (s_i, l_{i+1}, s_{i+1}) from state s_i to state s_{i+1} if there is a link or a form action that goes the page represented by s_i to the page represented by s_{i+1};
5. requests corresponding to explored forms or links define a transition that goes from the state where the request occurs to the state mapped to the response;
6. for each unexplored link $l \in L_i$ or form $a \in I_i$, the automaton has a transition from the state representing the page $\langle u_i, c_i, I_i, L_i, V_i \rangle$ to a so-called *trap* state $t \in T$.

The construction allows to define *deduced* links: they are links that are **not** visited during the browsing session, but are contained in one or more of the responses of the browsing session. Deduced links extend the automaton, making it a little more complete, enhancing property verification and improving reachability of certain states.

In Fig. 1 it is possible to see an example of a constructed session automaton. The links "URL1", "URL2", and "URL3" are unexplored links which transition to the trap state. The transitions from s_2 to s_1, from s_3 to s_2, and from s_4 to s_1 represent deduced links. Notice that deduced links are undistinguishable from regular links in the automaton representation.

3.2 Multi-display applications

The model presented in Section 3.1 is extended to handle multi-window and multi-frame applications. Such applications intrinsically possess concurrency because of how browsers load them: in the case of a web page with several frames, it is not possible to know in advance what will be the loading order of the pages. The authors note how it is theoretically possible to represent multi-display as single-display applications, but it is discouraged because it is cumbersome to represent multiple, parallel behaviours in a single automaton. Some of the definitions are extended from the ones of single-display applications:

- *response:* $\langle u, c, I, F, L, V \rangle$ with F being a set of frames in the page. The target t is defined; if no target is present $t = \varepsilon$. Additional changes are:
  - $\langle i, t \rangle \in L$
  - $\langle a, t \rangle \in I$
  - $\langle f, b \rangle \in F$

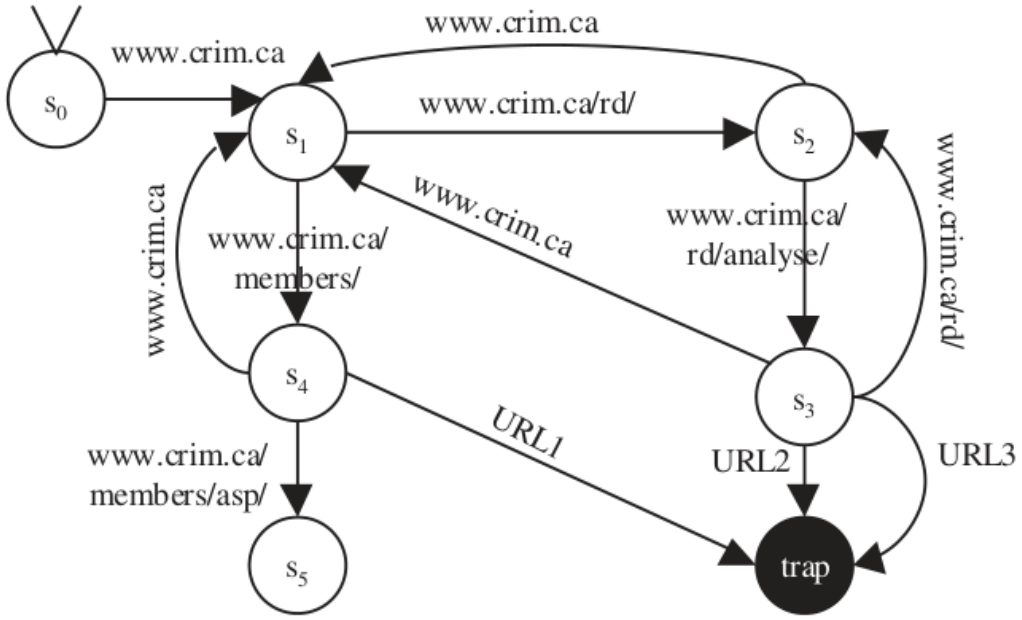

Figure 1: Example of a session automaton.

- the requests are now made of the link as before, with the addition of the referer r (link from which the request started) and the target t. They are denoted as $\langle r,l,t\rangle$

The procedure for building a single-display automaton is extended to build a communicating automata model.

Procedure 2. Convert a browsing session of a multi-display application into a communicating automata model.

1. a browsing session is split into a local browsing session $(RRS_1, \ldots, RRS_k)$, one for each window and frame;
2. convert each local browsing session RRS_i into an automaton;
   (a) use Procedure 1 to convert RRS_i to an automaton;
   (b) the alphabet Σ_i is extended with the source pages of the frames (src attribute), $\Sigma_i := \Sigma_i \cup \Phi_i$;
   (c) the case in which the user clicks on a link or submits a form while a frame is loading is handled by adding a transition from each state of the local automaton to the response state;
   (d) each unexplored link $\langle r_i,l_i,t_i\rangle \in \Gamma_i$ is mapped to a loop in the state it targets (self-loop);
3. create the communicating automata via the *parallel composition operator*, denoted $A_1 \mid\mid A_2$. The compositions of multiple automata is denoted $A_1 \mid\mid \cdots \mid\mid A_k$

A detailed explanation is presented in [3].

4 Extension of the automata model

In the communicating automata model described above, it is possible to characterize *transient* and *stable* states. Transient states represent situations where a multi-frame page is loaded, and the browser performs the requests for the frames in that page **without user intervention**.

The authors propose an *extended automata model* by adding a context variable to each state of each automaton. The context variable represents the number of frames that have to be loaded when in a certain state. If the context variable equals 0, the state is denoted stable, i.e. there are no more frames to load. Otherwise, the state is denoted transient.

In Fig. 2 an example of communicating automata is presented. The automaton in part (d) is not in its extended version, but it is possible to denote the transient and stable states already. The transient states are all the states that have an outgoing transition f_i, i.e. a transition that represents a frame loaded by the browser without user intevention. The stable states are therefore (s_0, u_0, w_0), (s_1, u_1, w_1), (s_1, u_2, w_1), (s_2, u_0, w_0).

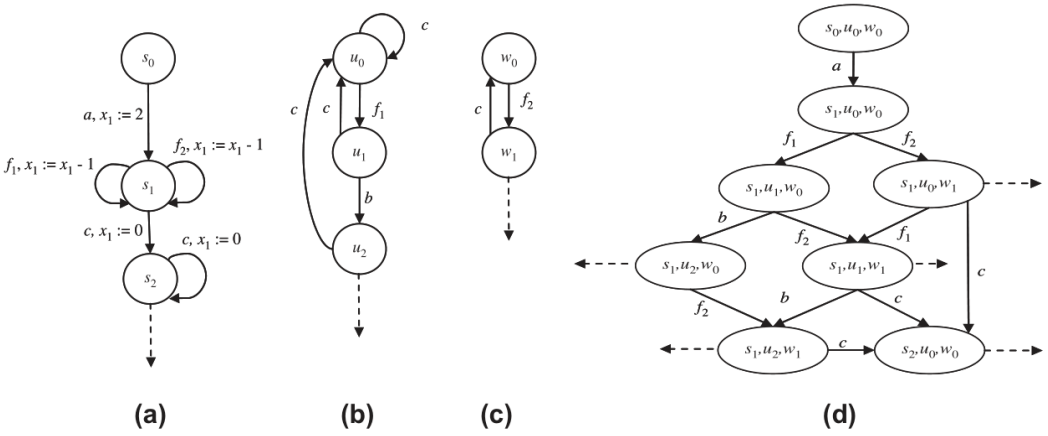

Figure 2: Example of communicating automata.

Each component automaton gets a context variable in each state. When all the component automata are in a state where the context variable is equal to 0, then the global state is considered stable.

4.1 Extended single-display automaton

The definition of an Extended Automaton for single-display applications follows:

Procedure 3. An automaton A_i is extended to an automaton Q_i as follows.

1. the set of states S_i, alphabet Σ_i and initial state s_{0i} are unchanged;
2. x_i is the context variable of Q_i, x_{0i} is the context variable's initial state;
3. for each transition $(s, a, s') \in T_i$, $s, s' \in S_i$, $a \in \Sigma_i$:
   (a) if $s = s'$ and $a \in \Sigma_i^d$, then $(s, a, x_i := x_i - 1, s)$ is a transition in Q_i, where $x_i := x_i - 1$ is the update of the transition; or
   (b) $(s, a, x_i := |init(s') \cap \Sigma_i^d|, s')$ is a transition in Q_i, where $x_i := |init(s') \cap \Sigma_i^d|$ is the update of the transition.

The designated set of transitions Σ_i^d is the set of those transitions who cause the automaton to pass through a transient state. In the case of browser sessions, the elements belonging to this set are the **browser triggered events**.

4.2 Extended multi-display automaton

The definition of a *communicating extended automata* model follows:

1. build the single-display automata;
2. apply Procedure 3 to get extended automata;
3. the set of designated events Σ_i^d is the set of frames of the browsing session;
4. x_i is initially set to 0;
5. at each state, x_i is assigned the number of frames that have to be loaded by the browser, or it is decremented;
6. each automaton is unfolded (transformed to its equivalent non-extended version);
7. the unfolded automata are composed using the composition operator.

The communicating extended automata model built as such is called stable if all its x_i variables are set to 0. Otherwise, it is called transient.

5 LTL and the **In** operator

To ease the definition of properties in a setting with automata possessing transient and stable states, the authors introduce new operators to increase the succintness of LTL. The operators allow to specify LTL properties over a subset of the state space offered by the system in consideration. For example, operators can be used to specify properties that hold only on the main page, or only in a subset of the pages of the application.

Over propositional logic expressions, the $\Im$-scope operator is introduced. The authors re-define LTL's $\neg$, $\wedge$, $\vee$ U, X, F, and G operators to use $\Im$ scopes, defining $\neg_\Im$, $\wedge_\Im$, $\vee_\Im$, $U_\Im$, $X_\Im$, $F_\Im$, and $G_\Im$. The scope is in itself a logical formula. A formula that is satisifed in a $\Im$ scope is not said to be satisfied outside of the scope.

Over logical formulas, instead, the **In** operator is introduced, which makes use of the $\Im$-scope operator. The full specification is detailed in [5].

An example of a simplication allowed by the **In** operator is the following.

Example 1.

$$G(((\neg Home \wedge \neg Shopping) \rightarrow (Promotions = 0)) \wedge ((Home \wedge Shopping) \rightarrow (Promotions \leq 2)))$$

Example 2.

$$G(((Promotions \leq 2)\ \mathbf{In}\ (Home \vee Shopping)) \vee (Promotions = 0))$$

6 Evaluation of the approach

6.1 Theoretical evaluation

The authors propose a theoretical evaluation that assume that all pages are static, i.e. there are no scripts running in them, the WAUT is static, i.e. during the observation it doesn't variate, that there is a one-to-one mapping between an URI and a page, and that always $c = 200$.

The definition of a (finite) web app automaton is then given.

Definition 1. Given a web application with the set of all the reachable pages **P**, the (finite) web application automaton, denoted A_{WA}, is a tuple $\langle S,s_0,\Sigma,T\rangle$ where:

- $S = \mathbf{P} \cup \langle u_0, c_0, I_0, L_0, V_0\rangle$ is the set of all pages;
- $s_0 \in S$ is the initial page $\langle u_0, c_0, I_0, L_0, V_0\rangle$;
- $\Sigma = \bigcup_{i=1}^{n} L_i$,$n = |P|$ is the set of all the links in all the pages **P**;
- $T \subseteq S \times \Sigma \times S$ is a set of all triples ($\langle u_i, c_i, I_i, L_i, V_i\rangle$, l_{i+1}, $\langle u_{i+1}, c_{i+1}, I_{i+1}, L_{i+1}, V_{i+1}\rangle$) such that $l_{i+1} = u_{i+1}, l_{i+1} \in L_i$.

The authors then present a theorem that states that each trace of a session automaton is also a trace of a web app automaton (cfr. Proposition 1 and Theorem 1 of the paper).

After this, a generalization to Kripke structures is made. The definition of a Kripke structure of a web application and of a browsing session are given (cfr. Definition 7 and Definition 8 of the paper). Then, a theorem that states that the browsing session Kripke structure M_{RRS} is a "reduced abstraction" of a web app Kripke structure M_{WA} (cfr. Proposition 2 in the paper). This means that if a property is violated in the browsing session Kripke structure, then it is also violated in the web application Kripke structure, for infinite counterexamples. For finite counterexamples, only *safety* properties keep this claim.

6.2 Implementation and empirical evaluation

The authors built a tool that can record a browsing session, build an internal representation of the session, evaluate a set of properties against the internal representation, and visualize the communicating automata. The set of properties can be split into *general* properties—applicable to every web app in existence—also defined as non-functional, and *specific* properties also defined as functional.

The exploration was performed on a number of websites chosen by the authors. Part of the websites were explored manually (by a human), and part by a crawler. The crawler performed a *complete* exploration: all the pages of the web app were explored.

Many of the defined properties were violated. The authors note how small and large web applications have a lower number of violations, while medium-sized applications have the highest.

An example of a specification where the authors found a counterexample is the following.

Example 3. $G((montreal \wedge fire \wedge underg) \vee ((montreal \wedge fire) \vee ((montreal \wedge underg) \vee ((underg \wedge fire) \vee montreal \vee underg \vee fire))))$

The counterexample is shown in Fig. 3.

Figure 3: Counterexample found by the tool.

7 Conclusions

It is important to notice how the rapid change of web development impacts the results of this paper. Using multiple frames is not common practice (although still used, e.g. in micro-frontends), and, more prominently, server-rendered HTML or manually written HTML is not the standard way of serving web applications.

Nowadays, web applications are typically *Single Page Applications (SPAs)*. SPAs make heavy use of JavaScript code on the frontend. For instance, routing is not happening via the browser, but by custom JavaScript modules called front-end routers. The result is that the HTML of a web page is changed by JavaScript and not sent by the server, which instead sends data in JSON format. While posing advantages and disadvantages, this has now become the standard practice for non-trivial web applications.

When talking about websites, the situation is different. Requiring less dynamic content by their nature, websites nowadays use either the SPA approach like web applications, the more traditional server-rendered approach, or a mix of the two: server rendering a page at its first load, and then performing *hydration* on it to transform it into an SPA. Additionally, a fourth approach is statically generating websites via Static Site Generators (SSGs) (examples being Jenkins[6], Hugo[7], Astro[1]). SSGs take as input plaintext files in formats like Markdown and, via a compilation step, output a set of HTML files that will form the website. This happens *once*, at development time, and not when a web page is requested.

The approach presented in the paper is suitable to be used in static, statically generated websites and server-rendered websites, but not in Single Page Applications and SPA-like websites.

References

[1] *Astro*. Astro. URL: https://astro.build/.

[2] Roy T. Fielding, Mark Nottingham, and Julian Reschke. *HTTP Semantics*. Request for Comments RFC 9110. Num Pages: 194. Internet Engineering Task Force, June 2022. DOI: 10.17487/RFC9110. URL: https://datatracker.ietf.org/doc/rfc9110.

[3] May Haydar, Alexandre Petrenko, and Houari Sahraoui. "Formal Verification of Web Applications Modeled by Communicating Automata". In: *Formal Techniques for Networked and Distributed Systems – FORTE 2004*. Ed. by David de Frutos-Escrig and Manuel Núñez. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 115–132. ISBN: 978-3-540-30232-2.

[4] May Haydar et al. "A formal approach for run-time verification of web applications using scope-extended LTL". In: *Information and Software Technology* 55.12 (2013), pp. 2191–2208. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2013.07.013. URL: https://www.sciencedirect.com/science/article/pii/S0950584913001596.

[5] May Haydar et al. "Propositional scopes in linear temporal logic". In: *Proceedings of the 5th International Conference on Novelles Technologies de la Repartition (NOTERE 2005)*. 2005. URL: https://www.researchgate.net/profile/Alexandre-Petrenko/publication/251394594_Propositional_Scopes_in_Linear_Temporal_Logic/links/004635296cd6fab256000000/Propositional-Scopes-in-Linear-Temporal-Logic.pdf (visited on 03/03/2025).

[6] *Jenkins*. Jenkins. URL: https://www.jenkins.io/.

[7] *The world's fastest framework for building websites*. URL: https://gohugo.io/.