Proving Properties of Rich Internet Applications

James Smith
Imperial College
London, United Kingdom
jecs@imperial.ac.uk

Abstract

We introduce application layer specifications, which allow us to reason about the state and transactions of rich Internet applications. We define variants of the state/event based logic UCTL* along with two example applications to demonstrate this approach, and then look at a distributed, rich Internet application, proving properties about the information it stores and disseminates. Our approach enables us to justify proofs about abstract properties that are preserved in the face of concurrent, networked inputs by proofs about concrete properties in an Internet setting. We conclude that our approach makes it possible to reason about the programs and protocols that comprise the Internet's application layer with reliability and generality.

1 Introduction

A glance at the existing literature shows that the Internet's transport layer is well understood from a mathematical point of view, with the inductive analysis of the TLS protocol [15] being perhaps the best example. On the other hand, the application layer that sits above it encompases a plethora of languages and frameworks, and understanding seems to be incomplete. At the highest level, the behaviour of Internet programs and protocols has been analysed [5, 13, 11], often for security reasons, but nevertheless there appears to be much work still to do.

These days, Internet applications are made up of both server side and client side parts. These so called "rich" Internet applications are now so commonplace as to be the norm. Any formalism that helps us to understand them mathematically seems a worthy goal, therefore. A complete formal treatment would include a formalisation of JavaScript [10] and much more, however, so here we attempt a loss lofty goal, giving formal specifications and proofs for that part of their behaviour related to communication over HTTP. To give an example, consider a "shoot-em-up" style game implemented in JavaScript and run in a web page. The functionality related to the unfolding of the gameplay would encompass game logic and the rendering of each scene, functionality not directly related to the Internet. Suppose the game includes a high score table, however, with high scores stored on the server. Maintaining this table would encompass issues relating to concurrency and communication over HTTP. It is the formal specifications and proofs for this kind of behaviour that is the subject of this paper.

Our motivation in studying this kind of behaviour is a broad understanding of the programs and protocols that comprise the Internet's application layer. With this in mind, this paper demonstrates that it is possible to formalise certain properties of rich Internet applications using a combination of mathematical models, transition systems, logics and related techniques. The approach is based on the conventional HTTP request-response mechanism whose participants are usually the web server on the one hand and the web browser on the other. Crucially, however, since communication between the client side part of an Internet application with its server side part, usually carried out by way of the XMLHTTPRequest object, uses the same HTTP request-response mechanism, our approach encompasses the behaviour of rich Internet applications, too.

A. Ravara and J. Silva (Eds.): 9th Workshop on
Automated Specification and Verification of Web Systems (WWV'13)
EPTCS 123, 2013, pp. 49–63, doi:10.4204/EPTCS.123.6

Our contributions are therefore threefold. Firstly, we formalise HTTP transactions and Internet application state, and introduce the concept of transition rules which describe the relation between the two. This constitutes our model of the Internet's application layer. Secondly, we define variants of the logic UCTL* [19], whose semantics are defined over the variants of Kripke transition systems [14] that these transition rules generate. We use these logics to express properties of these transition systems. Thirdly, in the case study we present a formal treatment of a distributed, rich Internet application and prove properties about the information it stores and disseminates. Our approach is a general one in the sense that we do not rely on a particular framework or programming paradigm. The properties we prove are ones that could be exhibited by any HTTP-based Internet program or protocol.

2 HTTP transactions and Internet application state

To the uninitiated an Internet application may appear to persist. A web page loads and then remains in the browser in much that same way that a desktop application remains in front of the user. The real situation is somewhat different, however. The presence of the web page may give the impression of permanence but in reality the instance of the server part of the application that produced it is likely to be destroyed the moment the web page is produced. By contrast, the client side part of a rich Internet application does persist, it can communicate with the server side part alongside the browser and it has access to and can amend shared information stored by the browser. To complicate the picture still further, the freedom that the browser admits means that user interactions cannot be guaranteed to occur sequentially in relation to the unfolding of the application state. The user may open new tabs, refresh the page, type URLs directly into the address bar or, most notoriously, use the forward and back buttons. The effects of these actions are complex to discern to say the least.

Rather than attempting to model this cat's cradle, we take a simplified approach. It begins with the working of the HTTP protocol and in particular the request-response pair, which we call an HTTP transaction. Although it is possible for the request-response pair to be broken, the connection may fail or the server may fault, for example, in our approach it will be considered inviolable. Many transactions may happen, building up the complete picture, so we therefore take account of all possible transactions. Although if we make the assumption that no two transactions can occur at exactly the same time then they can be given a temporal ordering, such an ordering is of little practical use since it is extremely difficult if not impossible to establish any causal link between successive transactions. It is best therefore to think of the transactions to begin with as floating around in a kind of "soup", with no connections between them at all.

At this point we make two observations. The first is that HTTP transactions can happen as a result of requests from both the browser and the client side part of the application, and therefore this soup may contain both types. The second is that no account is taken of the cause of each transaction. Each may happen as a result of the user clicking a link; pressing the back button; manually typing an address into the address bar; the client side part of the application making a call via the XMLHTTPRequest object; the list is endless. By eschewing these causes we avoid the trap of failing to properly account for them. Furthermore, we do not have a model of browser architecture or attempt to discern the subtle interplay of user interactions with iframes, different tabs and so on. And if we lose something in taking this line what we gain is a complete model of what remains, namely the behaviour of the application as evinced by its transactions over HTTP.

This soup of transactions on its own is of little use. To begin with it takes no account of the state of the application, however this is defined. And some causal links must be established, some additional structure needs to be added. To do this we include the states of the application in our approach, and associate them with transactions. More precisely, we say that if an application is in a certain state before an HTTP request, its state may change after the HTTP response. We call this process a transition, with that part of the transition visible over HTTP being precisely the transaction.

Our approach therefore consists of modelling all possible application states together with all possible transactions between them. We do not consider the causes of requests, and we assume that any request can happen at any time. This approach has precedents in the literature on the verification of network protocols, for example [16]. In such cases there is no user interface to be modelled, and verification consists of exhaustively "covering all the bases" with regard to all possible transactions and their effects on application state. And this approach also characterises some of the most stringent web security requirements in industry, including [3], where no assumptions are made about the types of requests that can be made in an "anything can happen" analysis. Our approach is a natural extension of these approaches.

3 A formal model and examples

In this section we define a formal model of HTTP transactions and Internet application state as well transition rules, which describe the relation between the two. We then define two example applications based on this model. The first application uses cookies to enforce user flow through a site and demonstrates the persistence of application state in the browser. The second application maintains a server state in order to track visitors to a site.

The HTTP protocol is a request-response protocol that admits communication between a client, typically a web browser or a JavaScript program running in a browser, and a server, typically an instance of a script or program running on a web server. Both requests and responses consist of a number of headers together with a body. The request often contains a reference, called a uniform resource locator or URL for short, to a resource to be retrieved by the server. Scripts and programs on the server can also be invoked in this manner. The headers communicate cookies in both directions, alongside other information.

Cookies consist of name-value pairs of textual information and allow stateful communication between clients and servers. They are initially passed by the server to the client in response to a request. They are then stored in the browser and passed back to the server in subsequent requests. Usually the instance of a script or program that produces any response is destroyed the moment that reponse is produced and therefore cookies allow subsequent instances to recover information specific to a particular client. Typically an identifier unique to the client is stored in a cookie, which allows information specific to that client to be retrieved from a database on the server. This leads to the common misconception that the identifier itself persists there. Aside from this, state not related to any particular client may be held on the server. Also a JavaScript program running in the browser may hold state apart from the browser's cookie store.

We model cookies as atomic entities rather than name-value pairs, taking them from a set of cookies $\mathscr{C}$, with c, c' ranging over $\mathscr{C} = \mathscr{P}(\mathscr{C})$. Since the browser can effectively be instructed to both add and remove cookies from its store, we model them as being signed when passed to the browser, so we have a set of signed cookies $\mathscr{C}^{\pm} = \{+x|x \in \mathscr{C}\} \cup \{-x|x \in \mathscr{C}\}$, with $c^{\pm}$ ranging over $\mathscr{C}^{\pm} = \mathscr{P}(\mathscr{C}^{\pm})$. Global states are tuples containing browser's cookie store, which from now on we refer to as the browser's state, together with the client side and server side states:

$$(c, j, s) \in \mathscr{C} \times \mathscr{J} \times \mathscr{S}$$

Requests consist of a set of cookies $c \in \mathscr{C}$ together with a URL u taken from a set of URLs $\mathscr{U}$. Responses consist of a set of signed cookies $c^{\pm} \in \mathscr{C}^{\pm}$ together with a body b taken from a set of bodies B. We may refine requests and responses further as required, adding additional request headers, for example, or parameterising the response bodies. A transaction consists of a request and response in one tuple, which labels a transition between global states:

$$(c,j,s) \xrightarrow{(c,u,c^{\pm},b)} (c',j',s')$$

Note that the browser's state c is echoed directly in the request, and therefore there is some redundancy in this formalism. We keep this, however, so that the transaction represents that part of the transition visible over HTTP in its entirety. To continue, when the browser receives signed cookies, it amends its state according to the following generic rule:

$$c' = c \cup \{x \in \mathscr{C} \mid +x \in c^{\pm}\} \backslash \{x \in \mathscr{C} \mid -x \in c^{\pm}\}$$

Transitions are governed by transition rules, which, given an initial state and request, determine both the response and final state:

$$(c,j,s) \xrightarrow[c^{\pm},b]{c,u} (c',j',s')$$

Transition rules are typically obtained from static analysis. This process could be automated or might be an informal but nonetheless informed approximation, as in case study given later. In the case of the examples that follow, however, the transition rules are formally stated and effectively define the applications in question.

3.1 The 'agreement' example

When visiting a website, a user must first agree to the use of cookies, then agree to the terms and conditions, and only then are they given access to the welcome page. No assumptions can be made about the order in which the pages are requested, however. In order to enforce the correct flow, two nonce cookies are set when the user agrees to the relevant missives. An error page is shown if the user requests the pages out of order, say, or bookmarks a page in order to come back to it at a later stage when the cookies have been deleted.

The user agrees both to the use of cookies and to the terms and conditions by way of forms presented on the first two pages. In both cases, only if the user ticks the checkbox and then submits the form are they allowed to continue. Upon successful submission of the forms nonce cookies are set, and it is the presence or otherwise of these cookies which determines which pages can be shown. No distinction is made between a page requested via the address bar and form submissions with the checkboxes left unticked, in which case the input is treated as invalid and the relevant page is shown again.

We model the two nonce cookies, plus the URLs that are used to request the three pages. To model valid form submissions, two additional URLs with ticks are added. We model the four page bodies returned, the first two containing the forms, then the welcome and error pages:

$$\mathscr{C} = \mathscr{P}(\{\textcircled{1},\textcircled{2}\}) \quad \mathscr{U} = \{1,2,w,2\checkmark,w\checkmark\} \quad \mathscr{B} = \{I,II,W,E\}$$

The application's behaviour is defined by the following set of transition rules:

$$\{\} \xrightarrow[\{\}\,I]{\{\}\,1\,|\,2} \{\} \qquad \{\textcircled{1}\} \xrightarrow[\{\}\,II]{\{\textcircled{1}\}\,2\,|\,w} \{\textcircled{1}\} \qquad \{\textcircled{1},\textcircled{2}\} \xrightarrow[\{\}\,W]{\{\textcircled{1},\textcircled{2}\}\,w} \{\textcircled{1},\textcircled{2}\}$$

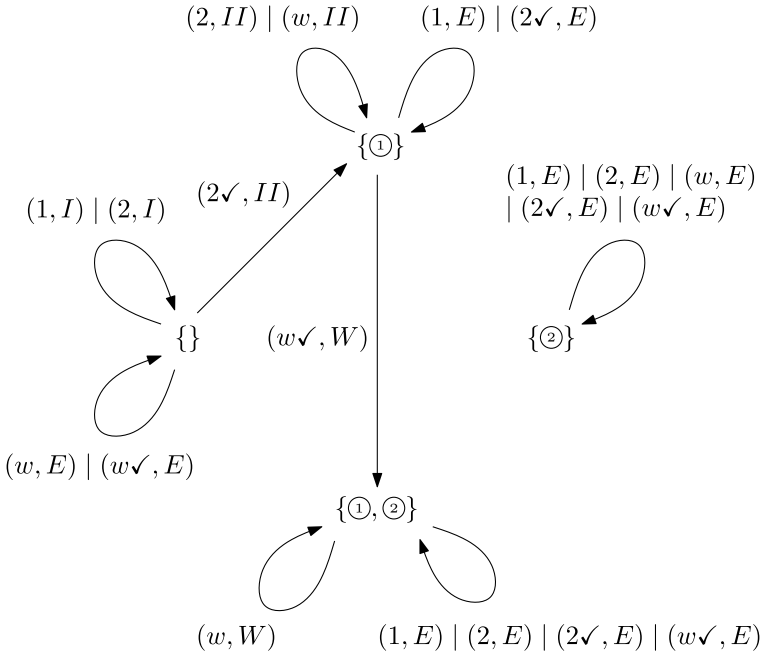
$(2, II) \mid (w, II)$
$(1, E) \mid (2\checkmark, E)$
$\{\textcircled{1}\}$
$(1, E) \mid (2, E) \mid (w, E)$
$\mid (2\checkmark, E) \mid (w\checkmark, E)$
$(1, I) \mid (2, I)$
$(2\checkmark, II)$
$\{\}$
$(w\checkmark, W)$
$\{\textcircled{2}\}$
$(w, E) \mid (w\checkmark, E)$
$\{\textcircled{1}, \textcircled{2}\}$
(w, W)
$(1, E) \mid (2, E) \mid (2\checkmark, E) \mid (w\checkmark, E)$

Figure 3.1: The ‘agreement’ application transition system

$$\{\} \xrightarrow[\{+\textcircled{1}\}\ II]{\{\}\ 2\checkmark} \{\textcircled{1}\} \qquad \{\textcircled{1}\} \xrightarrow[\{+\textcircled{2}\}\ W]{\{\textcircled{1}\}\ w\checkmark} \{\textcircled{1}, \textcircled{2}\}$$

$$\{\ldots\} \xrightarrow[\{\}\ E]{\{\textcircled{1}\} \mid \{\textcircled{2}\} \mid \{\textcircled{1},\textcircled{2}\}\ 1} \{\ldots\} \qquad \{\ldots\} \xrightarrow[\{\}\ E]{\{\textcircled{2}\} \mid \{\textcircled{1},\textcircled{2}\}\ 2} \{\ldots\} \qquad \{\ldots\} \xrightarrow[\{\}\ E]{\{\} \mid \{\textcircled{2}\}\ w} \{\ldots\}$$

$$\{\ldots\} \xrightarrow[\{\}\ E]{\{\textcircled{1}\} \mid \{\textcircled{2}\} \mid \{\textcircled{1},\textcircled{2}\}\ 2\checkmark} \{\ldots\} \qquad \{\ldots\} \xrightarrow[\{\}\ E]{\{\} \mid \{\textcircled{2}\} \mid \{\textcircled{1},\textcircled{2}\}\ w\checkmark} \{\ldots\}$$

The first row represents the occasions when the user requests the correct page or submits a form in the correct order but without ticking the checkbox. The second row represents the occasions when the user correctly submits a form and a nonce cookie is set. The third and fourth rows represent the remaining occasions, on all of which error pages are shown. Note that both the client side and server side states are missing from these rules, since they are not present in this example. Note also that for brevity's sake, the browser states are sometimes replaced by ellipses with the understanding that they can be inferred from the headers. Recall that the browser's state is echoed directly in the request. Finally, note that vertical bars allow several rules to be combined into one. Figure 3.1 shows the resulting transition system, which in this example is no more than a diagrammatic representation of the transition rules. The cookie headers are omitted, since they can be inferred from the initial and final states. Note that we also use the vertical bar rather than set notation, in keeping with the transition rules.

3.2 The ‘visitors’ example

A website maintains a counter on a landing page which is incremented every time a user requests that page for the first time. No cookies can be used and so in order to determine whether a request has originated from a new user, the request IP address is checked against the server state. If the request IP address cannot be found, the counter is incremented and the new IP address is added to the server state.

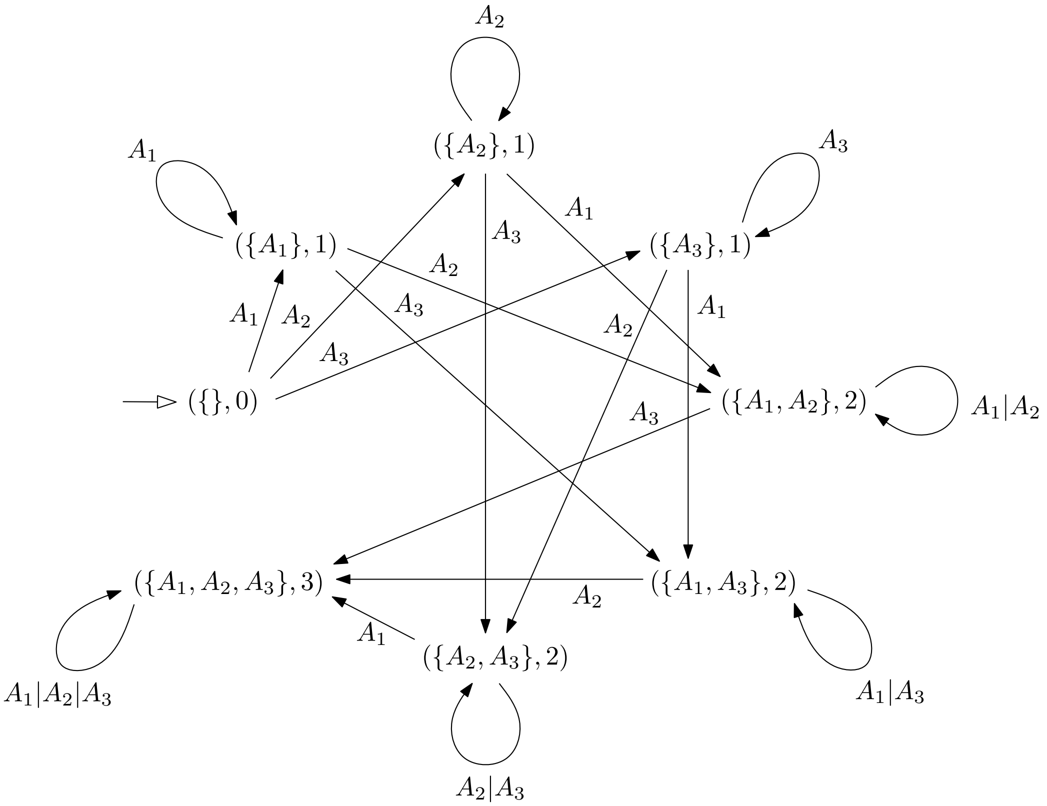

Figure 3.2: The 'visitors' application transition system

We model the lack of cookies, a single URL, a set of IP addresses and a countable set of page bodies. The server state is modelled as a subset of the set of the IP addresses paired with the states of the counter, which are precisely the natural numbers $\mathbb{N}$. In this case the application also has an initial state:

$$\mathscr{C} = \mathscr{P}(\{\}) \quad \mathscr{U} = \{U\} \quad \mathscr{A} = \{A_1, A_2, \cdots, A_N\} \quad \mathscr{B} = \{B(n) \mid n \in \mathbb{N}\}$$

$$(a,n) \in \mathscr{S} \subseteq \mathscr{P}(\mathscr{A}) \times \mathbb{N} \quad (\{\},0) \in \mathscr{S}$$

The application's behaviour is defined by the following two transition rules for $1 \leqslant i \leqslant N$:

$$(a,n) \xrightarrow[B(n)]{U,A_i} (a,n) \quad A_i \in a \qquad\qquad (a,n) \xrightarrow[B(n+1)]{U,A_i} (a \cup \{A_i\}, n+1) \quad A_i \notin a$$

Whereas the IP address is communicated alongside the URL in the request, on the other hand the counter is communicated as part of the page body. The returned page is essentially parameterised by the counter, therefore, reflecting the fact that the counter is embedded in the body of the page by some means. Note that the browser and client side states are missing from these rules, since neither is not present in this example. Note also that since there is no browser state, no cookies are communicated in the request and response parts of the rules. Strictly speaking these two transition rules are not single rules but rule schemas defining two countable sets of rules for each $(a,n) \in \mathscr{P}(\mathscr{A}) \times \mathbb{N}$. Figure 3.2 shows the resulting transition system in the case when $N = 3$. The initial state is highlighted on the left with a white filled arrow. Since there is only one request URL and one returned page, these are omitted. It is also assumed that the correct value of the counter is echoed in the returned page and therefore this information is not included. Note again that we use the vertical bar rather than set notation for the transition labels.

4 A treatment of the examples using temporal logics

The two transition systems defined in the previous section are, with small differences, instances of Kripke transition systems, with information on both the states and transitions. In order to express properties of the applications in question we therefore use a variant of UCTL* [19], a state/event based logic whose semantics are defined over these types of transition systems. In what follows we give the standard definition of Kripke transition systems [14], together with the syntax and semantics of UCTL*, and then define variants of both for our purposes.

Definition 4.1. *A Kripke Transition System is a tuple* $(S, Act, \longrightarrow, AP, \mathscr{L})$ *where:*

- *S is a set of states ranged over by s,s_0 and s_1,*
- *Act is a set of actions, with 2^{Act} ranged over by α,*
- *$\longrightarrow \subseteq S \times 2^{Act} \times S$ is the transition relation with $(s_0, \alpha, s_1) \in \longrightarrow$,*
- *AP is a set of atomic propositions ranged over by p,*
- *$\mathscr{L} : S \times AP \longrightarrow \{true, false\}$ is an interpretation function that associates a value of $true$ or $false$ with each $p \in AP$ for each $s \in S$,*
- *For any two transitions, $(s_0, \alpha_0, s_1), (s_0, \alpha_1, s_1) \in \longrightarrow \Rightarrow \alpha_0 = \alpha_1$.*

Note that since transitions carry sets of actions and not just one, there is no silent action, with the empty set $\{\}$ being considered silent instead. Note also that we limit the number of transitions between any two states in any one direction to at most one.

Paths are sequences of transitions where the final state of one transition equals the initial state of the next, if there is one. They are ranged over by σ, σ' and σ''. Maximal paths are either infinite or their last state has no outgoing transitions. For the set of maximal paths starting at state s we write $\mu path(s)$. For the initial and final states of the first transition of a path σ we write ${}_{\mathrm{S}}\sigma$ and σ_{S}, respectively, and we write σ_{T} for the set of actions of the first transition of a path σ. A suffix σ' of a path σ is a path such that $\sigma = \sigma''\sigma'$ for some possibly zero length path σ''. A proper suffix σ' of a path σ is a path such that $\sigma = \sigma''\sigma'$ for some non-zero length path σ''. We write $\sigma \leqslant \sigma'$ when σ' is a suffix of σ and $\sigma < \sigma'$ when σ' is a proper suffix of σ.

Definition 4.2. *The syntax UCTL* is:*

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi' \mid \exists\pi \qquad \pi ::= \phi \mid \neg\pi \mid \pi \wedge \pi' \mid X\pi \mid X_a\pi \mid \pi\, U\pi'$$

Here ϕ is a state formula and π a path formula.

Definition 4.3. *The semantics of UCTL* is:*

- $s \models p$ *iff* $\mathscr{L}(s, p) = true$
- $s \models \neg\phi$ *iff* $s \not\models \phi$
- $s \models \phi \wedge \phi'$ *iff* $s \models \phi$ *and* $s \models \phi'$
- $s \models \exists\pi$ *iff* $\exists\sigma \in \mu path(s) : \sigma \models \pi$
- $\sigma \models \phi$ *iff* ${}_{\mathrm{S}}\sigma \models \phi$
- $\sigma \models \neg\pi$ *iff* $\sigma \not\models \pi$
- $\sigma \models \pi \wedge \pi'$ *iff* $\sigma \models \pi$ *and* $\sigma' \models \pi'$
- $\sigma \models X\pi$ *iff* $\exists(s, \alpha, s')\sigma'' : \sigma = (s, \alpha, s')\sigma'', \sigma'' \models \pi$
- $\sigma \models X_a\pi$ *iff* $\exists(s, \alpha, s')\sigma'' : a \in \alpha, \sigma = (s, \alpha, s')\sigma'', \sigma'' \models \pi$
- $\sigma \models \pi\, U\pi'$ *iff* $\exists\sigma' \geqslant \sigma : \sigma' \models \pi', \forall\sigma'' : \sigma \leqslant \sigma'' < \sigma' : \sigma'' \models \pi$

We now define a variant of UCTL* for our purposes, replacing atomic propositions with propositions about typed variables.

Definition 4.4. *Let x be a typed variable. The type of x, written τ_x, is the set of its permitted values. A set of typed variables is written X.*

Definition 4.5. *An interpretation of a set of typed variables X is a function I from X to $\bigcup\{\tau_x | x \in X\}$ such that $I(x) \in \tau_x$ for each $x \in X$. The set of all interpretations for a set of typed variables X is written $\mathbb{I}$.*

Definition 4.6. *An (amended) Kripke Transition System a tuple $(S, Act, \longrightarrow, X, \mathscr{L})$ where the set of atomic propositions AP and interpretation function $\mathscr{L}$ replaced with the following:*

- *X is a set of typed variables,*
- *$\mathscr{L} : S \longrightarrow \mathbb{I}$ is an interpretation function that maps every state to an interpretation.*

Action formulae remain the same, however state formulae must be amended in order to accomodate propositions of the form $x = x'$ where x and x' are of the same type, or $x = v$ where $v \in \tau_x$. In the spirit of [6], we also introduce a set K of typed global variables, extending the syntax to include propositions of the form $x \leqslant k$ where $k \in K$, x and k are of the same type, the type is a numerical type that supports inequalities and the value of k is in τ_x.

Definition 4.7. *The (amended) syntax of UCTL* state formulae ϕ is:*

$$\phi ::= true \mid x = v \mid x = x' \mid x \leqslant k \mid \neg\phi \mid \phi \wedge \phi' \mid \exists\pi$$

Definition 4.8. *The (amended) semantics of UCTL* state formulae ϕ is:*

- *$s \models x = v$ iff $\mathscr{L}(s)(x) = v$*
- *$s \models x = x'$ iff $\mathscr{L}(s)(x) = \mathscr{L}(s)(x')$*
- *$s \models x \leqslant k$ iff $\mathscr{L}(s)(x) \leqslant k$*

The relation $s \models p$ is dropped with the other relations remaining.

If x has a numerical type, we also permit expressions of the form $x = v + 1$ with the obvious semantics. Unlike [6], which introduces typed constants, we might close formulae containing typed global variables under existential or universal quantification. If they are not closed in such a way, we consider them constant. We also quantify over sets of constants.

Finally, we derive the "eventually" F and "always" G operators [8] in the usual way and add a T operator, the latter being a convenient shorthand for combining path formulae satisfied immediately and "nexttime" on a path:

$$F\pi' = true_{true}U_{true}\pi' \qquad G\pi = \neg F \neg \pi \qquad \pi T_{\chi}\pi' = \pi \wedge X_{\chi}\pi'$$

4.1 The 'agreement' example

Here $X = \{c\}$ and $\tau_c = \{\{\}, \{\textcircled{1}\}, \{\textcircled{2}\}, \{\textcircled{1}, \textcircled{2}\}\}$. For the sake of readability, formulae $x = v$ are shortened to just v. For example, $c = \{\textcircled{1}\}$ is written $\{\textcircled{1}\}$. The ideal user journey is the following path:

$$\{\} \xrightarrow{(1, I)} \{\} \xrightarrow{(2\checkmark, II)} \{\textcircled{1}\} \xrightarrow{(w\checkmark, W)} \{\textcircled{1}, \textcircled{2}\}$$

It is easy to produce a path with the same requests but the wrong responses, however:

$$\{\textcircled{1}, \textcircled{2}\} \xrightarrow{(1, E)} \{\textcircled{1}, \textcircled{2}\} \xrightarrow{(2\checkmark, E)} \{\textcircled{1}, \textcircled{2}\} \xrightarrow{(w\checkmark, E)} \{\textcircled{1}, \textcircled{2}\}$$

Lemma 4.1. *The following two formulae can be satisfied:*

$$\exists(\{\}T_{(1,I)}\{\}T_{(2\checkmark,II)}\{\textcircled{\scriptsize 1}\}T_{(w\checkmark,W)}\{\textcircled{\scriptsize 1},\textcircled{\scriptsize 2}\}) \qquad \exists(X_{(1,E)}X_{(2\checkmark,E)}X_{(w\checkmark,E)}true)$$

Proof. These follow immediately from inspecting figure 3.1, which includes the above paths. □

4.2 The 'visitors' example

Here $X = \{a,n\}$, $\tau_a = \mathscr{P}(\mathscr{A})$, $\tau_n = \mathbb{N}$ and $K = \{N_1,N_2,N\}$ where $N_1,N_2,N \in \mathbb{N}$ and $N_1,N_2 \leqslant N$. We continue to use an ordered pair (a,n) for the state rather than a set.

Lemma 4.2. *The only state with $n = 0$ and $a = \{\}$ is the initial state.*

Proof. Only the second rule can be applied to the initial state, and there is no rule for subsequently decreasing n or reducing a. □

Lemma 4.3. *Apart from the initial state, all reachable states are of the form $(a \cup \{A_i\}, n+1)$, where $A_i \notin a$, and are reached by way of a transition from state (a,n).*

Proof. The first transition rule leaves states unchanged, therefore if a state is not the initial state it must be reached by the application of the second transition rule to some state, say (a,n), and therefore must be of the form $(a \cup \{A_i\}, n+1)$. □

Lemma 4.4. *For any reachable state (a,n), $|a| = n$.*

Proof. By induction on n. For $n = 0$ we know from lemma 4.2 that the only state with $n = 0$ is the initial state. Assume now that for all states (a,k) we have $|a| = k$. Consider some other, reachable state with $n = k+1$ which by lemma 4.3 we can write as $(a \cup \{A_i\}, k+1)$, reachable from the state (a,k). By the induction hypothesis $|a| = k$ and therefore $|a \cup \{A_i\}| = k+1$. □

Lemma 4.5. *For any reachable state (a,n), $n \leqslant N$. Formally, $\forall G(n \leqslant N)$ is valid.*

Proof. We simply observe that $n = |a| \leqslant |\mathscr{A}| = N$. □

We omit the proof that there is a path that leads to a state (a,n) for any $a \in \mathscr{P}(\mathscr{A})$ but note that this fact, in tandem with lemma 4.4, defines the set $\mathscr{S}$ of all reachable states.

Lemma 4.6. *The value of the counter returned to the user is no greater than N.*

Proof. For the first rule, the response is $B(n)$ with $n \leqslant N$. For the second rule, the response is $B(n+1)$ but $n = |a|$ by lemma 4.4, $a \subset \mathscr{A}$ and since $|\mathscr{A}| = N$, $n < N$ and therefore $n+1 \leqslant N$. □

Lemma 4.7. *No user can increment the counter twice. Formally, the following is not satisfiable:*

$$\exists A_i, N_1, N_2 : \exists(((n = N_1)T_{A_i}(n = N_1 + 1)) \wedge F((n = N_2)T_{A_i}(n = N_2 + 1)))$$

Proof. Suppose that the formula is satisfied. The path in question must be of the following form:

$$(a_1,N_1) \xrightarrow{A_i} (a_1 \cup \{A_i\}, N_1+1) \;\cdots\; (a_2,N_2) \xrightarrow{A_i} (a_2 \cup \{A_i\}, N_2+1)$$

We first note that $a_1 \cup \{A_i\} \subseteq a_2$, since all transitions are governed by two transition rules, one of which leaves the set a of any state (a,n) alone and the other of which adds an element. Then we note that in order for the second rule to be applied at the end of the path we must have $A_i \notin a_2$. However, $a_2 \supseteq a_1 \cup \{A_i\}$ and $A_i \in a_1 \cup \{A_i\}$ implies $A_i \in a_2$, a contradiction. □

5 The QuICDoc case study

QuICDoc [4] is a small, distributed, rich Internet application that uses the Concur algorithm [18] to merge concurrent changes to a shared document. A user simply types into a textarea and a client-side part of the application communicates the changes, what are called "diffs", to the server side part, which disseminates them to all the other client side parts after amending them in such as way as to ensure that they can be merged consistently. In [18] it is proved that this is possible but no account is taken of the effects of implementing the algorithm in an Internet setting. We bridge that gap in this section.

To begin with we rule out concurrency effects at all levels bar the application level. QuICDoc is executed on Node.js [1], which uses an event-driven, non-blocking I/0 model, therefore we rule out concurrency effects at the system level without further ado. At the network level, once initialised the client side part of QuICDoc makes two repeated calls to the server side part. These are synchronous calls, opening a connection and providing a callback method. We mitigate against the chance of one callback interrupting another by introducing a shared gUpdate variable that stops one method making a call if the other method's callback is pending. The relevant parts of the client side code are shown immediately below:

```
var gUpdate;
var fGetDoc = function () {
    gUpdate = false;
    fGetDiffs();
}
var fGetDiffs = function () {
    if ( gUpdate ) { setTimeout( "fGetDiffs()", 1000 ); return; }
    gUpdate = true;
}
var fPutDiffs = function () {
    if ( gUpdate ) { return; }
    gUpdate = true;
}
```

Note that in order to avoid race conditions, the fPutDiffs() method exits if the fGetDiffs() callback is pending, meaning that the diffs generated by the user input are discarded. The next user input will generate fresh diffs with no information being lost, however. On the other hand the fGetDiffs() method is rescheduled if the fPutDiffs() callback is pending. Use of a shared variable to enforce asynchronicity in the face of timers, the event model and the like may seem naïve, however we maintain that it has the desired effect in practice. Moving on, at what we call the application level, concurrency effects are mitigated by the Concur algorithm.

In modelling the implementation of this algorithm we consider just two clients, defining the global state as a tuple $(j_1, j_2, s) \in \mathscr{J}_1 \times \mathscr{J}_2 \times \mathscr{S}$ encompassing both client side states alongside the server side state. In order to relate the formal model to the implementation we define states as named tuples with values associated with the relevant heap variables, thus $j_i(\mathsf{gUid}, \mathsf{gWorkingDoc}, \mathsf{gTempDoc})$ for $i = 1,2$ together with $s(\mathsf{gLastUid}, \mathsf{gDocument}, \mathsf{gDiffss})$. The transition rules are broken down into client side and server side rules, and are obtained from the source code [4] by manual static analysis:

$$j_i(\varepsilon, \varepsilon, \varepsilon) \xrightarrow[uid,doc]{\mathsf{GET\_DOC}} j_i(uid, doc, \varepsilon)$$

$$j_i(uid, doc, temp) \xrightarrow[diffs]{\mathsf{GET\_DIFFS}, uid} j_i(uid, \mathsf{applyDiffs}(doc, diffs), temp)$$

$$j_i(uid, doc, temp) \xrightarrow{\mathsf{PUT\_DIFFS}, uid, \mathsf{makeDiffs}(doc, temp)} j_i(uid, temp, temp)$$

$$s(luid, doc, diffss) \xrightarrow[luid+1, doc]{\mathsf{GET\_DOC}} s(luid+1, doc, \mathsf{resetDiffs}(diffss, luid+1))$$

$$s(luid, doc, diffss) \xrightarrow[\mathsf{getDiffs}(diffss, uid)]{\mathsf{GET\_DIFFS}, uid} s(luid, doc, \mathsf{resetDiffs}(diffss, uid))$$

$$s(luid, doc, diffss) \xrightarrow{\mathsf{PUT\_DIFFS}, uid, diffs} s(luid, \mathsf{amendDoc}(doc, uid, diffs), \mathsf{amendDiffss}(diffss, uid, diffs))$$

Here ε represents an undefined value and a side condition for all the rules is that $uid \neq \varepsilon$. There is also an initial global state, which is again broken down into client side and server side parts, thus $j_i(\varepsilon, \varepsilon, \varepsilon)$ for $i = 1,2$ together with $s(0, \text{“”}, [[], []])$. Here [...] represents an array.

We note in passing that breaking down the transition rules in this way provides a means of partially specifying the behaviour of each part of the application, and hence taken together they can be considered a specification of the application as a whole as evinced by its transactions over HTTP. For this reason we call these sets of rules application layer specifications.

To make use of the rules we recombine them. We combine the GET_DOC rules informally first, by way of an example, generating sets of substitutions that we apply to the final states. In order to do this we note that the client side part produces a request and consumes a response whilst the server side part consumes a request and produces a response. Here the request consists of just a constant, with no other elements. The response on the other hand consists of non-constant elements. Since the client side part consumes the response, the *uid* and *doc* variables of the client side part are replaced with the $luid + 1$ term and *doc* variable of the server side part. In order to distinguish client side elements from server side elements, subscripts are added. Once the substitutions have been made, all elements labelling the transition are discarded with the exception of the constant representing the command. The resulting global rule is the following:

$$(j_i(\varepsilon, \varepsilon, \varepsilon), s(luid, doc, diffss)) \xrightarrow{\mathsf{GET\_DOC}} (j_i((luid+1)_s, doc_s, \varepsilon), s(luid+1, doc, diffss))$$

We use this rule to justify an assumption in the proof of the correctness of the Concur algorithm. Specifically, lemma 6.1 in [18] makes the assumption that the document is passed from the server to the client intact. The relevant transitions are shown in figure 5.1.

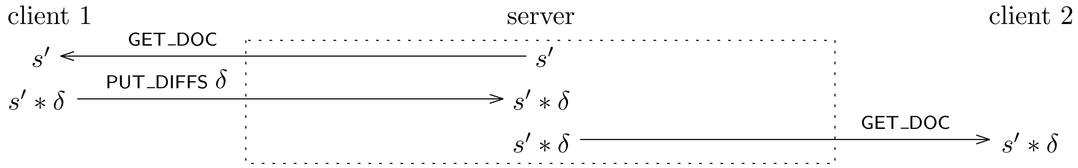

Figure 5.1: The GET_DOC and related transitions

Lemma 5.1. *A* GET_DOC *transition results in the* gWorkingDoc *client side variable attaining the value of the* gDocument *server side variable.*

Proof. See the aforementioned global rule. The second element on the client side tuple is mapped to the gWorkingDoc client side variable while doc_s holds the value of the gDocument server side variable. □

We next formally define a matching function for the rules in a similar vein to [9], collecting the elements of the requests and responses into tuples, with elements being either variables, constants or terms. A substitution is returned only if a match can be found.

Definition 5.1. *The partial matching function for pairing transition rules is defined by structural induction by means of the following partial functions:*

$$\begin{aligned}
\mathsf{match}((e_1,e_2...),(e'_1,e'_2...)) &= \{\mathsf{match}(e_1,e'_1)\} \cup \mathsf{match}((e_2...),(e'_2...))\\
\mathsf{match}(v,v') &= v/v'\\
\mathsf{match}(v,c') &= v/c'\\
\mathsf{match}(v,t') &= v/t'\\
\mathsf{match}(c,c) &= \emptyset\\
\mathsf{match}((),()) &= \{\}
\end{aligned}$$

Here e_1, e'_1*, etc are arbitary elements,* v, v' *variables,* c, c' *constants and* t' *terms.*

To match the requests, we make the server side tuple the first argument of the matching function since it is this tuple that contains the variables that must be replaced in the substitutions that follow. To match the responses, we make the client side tuple the first argument. By way of an example we show the matching of the requests and responses of the GET_DIFFS rules:

$$\begin{aligned}
&\mathsf{match}((\mathsf{GET\_DIFFS}, uid_s),(\mathsf{GET\_DIFFS}, uid_i))\\
&= \{\emptyset\} \cup \mathsf{match}((uid_s),(uid_i))\\
&= \{\emptyset\} \cup \{\mathsf{match}(uid_s, uid_i)\}\\
&= \{\emptyset\} \cup \{uid_s/uid_i\} \cup \mathsf{match}((),())\\
&= \{\emptyset, uid_s/uid_i\}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{match}((diffs_i),(\mathsf{getDiffs}(diffss_s, uid_s)_s))\\
&= \{\mathsf{match}(diffs_i, \mathsf{getDiffs}(diffss_s, uid_s)_s)\} \cup \mathsf{match}((),())\\
&= \{diffs_i/\mathsf{getDiffs}(diffss_s, uid_s)_s\}
\end{aligned}$$

Here the rules are paired as expected by matching the constants, and we apply the substitutions to obtain the final states of the global rule:

$$\begin{aligned}
&j_i(uid, \mathsf{applyDiffs}(doc, diffs), temp).\{diffs_i/\mathsf{getDiffs}(diffss_s, uid_s)_s\}.\{\emptyset, uid_s/uid_i\}\\
&= j_i(uid, \mathsf{applyDiffs}(doc, \mathsf{getDiffs}(diffss_s, uid_s)_s), temp).\{\emptyset, uid_s/uid_i\}\\
&= j_i(uid, \mathsf{applyDiffs}(doc, \mathsf{getDiffs}(diffss_s, uid_i)_s), temp)
\end{aligned}$$

$$\begin{aligned}
&s(luid, doc, \mathsf{resetDiffs}(diffss, uid)).\{\emptyset, uid_s/uid_i\}.\{diffs_i/\mathsf{getDiffs}(diffss_s, uid_s)_s\}\\
&= s(luid, doc, \mathsf{resetDiffs}(diffss, uid_i)).\{diffs_i/\mathsf{getDiffs}(diffss_s, uid_s)_s\}\\
&= s(luid, doc, \mathsf{resetDiffs}(diffss, uid_i))
\end{aligned}$$

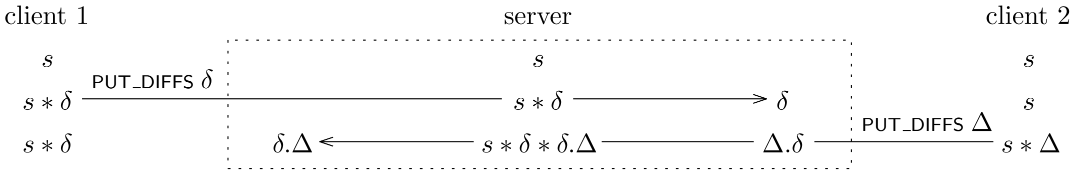

Figure 5.2: The PUT_DIFFS transitions

Note that both substitutions must be applied to the final states, since terms labelling the transitions may themselves require subsitutions. For example, the $\{\emptyset, uid_s/uid_i\}$ substitution can be applied to the getDiffs$(diffss_s, uid_s)_s$ term to yield getDiffs$(diffss_s, uid_i)_s$, which can then be substituted into the final state of the client side rule, yielding the same result. We choose to apply the substitutions in reverse order so that they can both be applied directly to the final state, however.

Lemma 5.2. *A* GET_DIFFS *transition results in the client in question applying its pending diffs to its copy of the document, held in the* gWorkingDoc *client side variable. The client in question's pending diffs are then reset.*

Proof. See the aforementioned final states above. □

To conclude we consider the global rule for the PUT_DIFFS command. In this case we apply the requisite substitutions but keep the client side and server side parts separate due to space considerations. We treat the client side part first:

$$j_i(uid, doc, temp) \xrightarrow{\mathsf{PUT\_DIFFS}} j_i(uid, temp, temp)$$

The implementation maps the gTempDoc client side variable directly to the textarea. When its value changes, diffs are generated from the difference between this variable and the gWorkingDoc variable. When these diffs are put on the server, the gWorkingDoc variable is updated with the value of the gTempDoc variable. The above rule simply states this fact. Moving on, we give the server side part of the global rule:

$$s(luid, doc, diffss) \xrightarrow{\mathsf{PUT\_DIFFS}} s(luid, \mathsf{amendDoc}(doc, uid_i, diffs_i), \mathsf{amendDiffss}(diffss, uid_i, diffs_i))$$

Here we have abbreviated makeDiffs$(doc_i, temp_i)_i$ as $diffs_i$. We use this rule to justify another assumption in the proof of the Concur algorithm. Specifically, lemma 6.2 in [18] makes the assumption that if clients put diffs on the server, these are amended appropriately and applied to its copy of the document. The relevant transitions are shown in figure 5.2.

Lemma 5.3. *A* PUT_DIFFS *transition results in the* gDocument *server side variable being updated with the return value of the* appendDoc(...) *server side method, whose arguments are the identifier and diffs passed by the client in question.*

Proof. See the aforementioned global rule, where the second element of the server side tuple is mapped to the gDocument server side variable while uid_i and $diffs_i$ hold the values of the gUid client side variable and return value of makeDiffs(...) client side method, respectively. □

6 Related work

The approach taken in [5] defines a formal model of web security and identifies, amongst others, vulnerabilities in WebAuth, a web-based authentication system based on Kerebos. The concept of non-linear time is adopted, with the authors not being "...concerned with the actual temporal order between unrelated actions", an approach strongly espoused here.

In [12], significant emphasis is placed on user interactions. The opinion that "...any verfication tool for the web that does not account for user operations is not only incomplete, but potentially misleading" could be construed as being somewhat the opposite to our own. In [7], emphasis is placed on the structure of the web site in terms of its interelated links, with transitions being seen as taking place between web pages. In effect a temporal order is imposed, with users choosing only from a set of actions presribed by each web page, again in marked contrast to the approach taken in [5] and our own.

By eschewing a model of user interactions, our approach owes more to the analysis of web services [16] than applications. And although our approach is not a model checking one, it has close parallels with the exhaustive treatments that typify model checking. Lastly, we note that our approach bears some resemblance to the inductive analysis of network protocols [15], which effectively build an operational semantics of protocols given a set of rules and then prove that certain safety properties are preserved.

7 Conclusions and future work

In this paper we have developed a formal model of HTTP transactions and Internet application state. We have defined two formal, example applications based on this model and proved properties of these applications. We then extended our approach to the treatment of a case study, namely a rich, distributed Internet application, and proved properties of this application, too. Because our model is founded only on HTTP transactions and Internet application state, and not on any particular framework or programming paradigm, we claim that it can be used to prove properties of any Internet program or protocol.

The case study chosen was relatively simple, with both server side and client side parts written in JavaScript. Our long term goal is more ambitious, however. In future work we plan to use application layer specifications to formalise properties of larger HTTP-based programs and protocols with, specifically, the OAuth 2.0 protocol [2] in mind.

The choice of the logic UCTL* was motivated by the need for formulae with nested next time operators but, rather than defining the $\phi T_\chi \phi'$ operator as shorthand for $\phi \wedge X_\chi \phi'$, we could define its semantics directly and otherwise do away with the need for a fully branching logic. Furthermore, as demonstrated in the case study, it is possible to express properties of these systems without using a logic. For this reason, developing further variants of UTCL* or any other logic is currently not the focus of future work.

Finally, we make some comments on the use of terms in the transition rules in the case study. The treatment was somewhat informal, however a formal treatment can be found for example in [17], which includes transition systems generated by rules, sets of which are gratifyingly called transition system specifcations. Moreover, semantics that encompases both the denotational meaning of terms and their operational meaning over transition systems are developed therein. We conclude therefore by looking forward to a formal treatment of the ideas presented here based on these approaches.

7.1 Acknowledgements

The author thanks the anonymous reviewers for their many helpful comments and corrections.

References

[1] *node.js*. Available at http://nodejs.org/.

[2] *OAuth2.0*. Available at http://oauth.net/2/.

[3] *PCI SSC Data Security Standards Overview*. Available at https://www.pcisecuritystandards.org/security_standards/.

[4] *QuICDoc*. Available at http://www.doc.ic.ac.uk/~jes204/quicdoc.zip.

[5] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell & Dawn Song (2010): *Towards a Formal Foundation of Web Security*. In: *CSF*, pp. 290–304. Available at http://doi.ieeecomputersociety.org/10.1109/CSF.2010.27.

[6] Jürgen Bohn, Werner Damm, Orna Grumberg, Hardi Hungar & Karen Laster (1998): *First-Order-CTL Model Checking*. In: *FSTTCS*. Available at http://dx.doi.org/10.1007/b71635.

[7] Alin Deutsch, Liying Sui & Victor Vianu (2007): *Specification and verification of data-driven Web applications*. *J. Comput. Syst. Sci.* 73(3), pp. 442–474. Available at http://dx.doi.org/10.1016/j.jcss.2006.10.006.

[8] E. Allen Emerson & Joseph Y. Halpern (1983): *"Sometimes" and "not never" revisited*. POPL '83, ACM, New York, NY, USA, pp. 127–140. Available at http://doi.acm.org/10.1145/567067.567081.

[9] Alessandro Fantechi, Stefania Gnesi, Alessandro Lapadula, Franco Mazzanti, Rosario Pugliese & Francesco Tiezzi (2012): *A logical verification methodology for service-oriented computing*. *ACM Trans. Softw. Eng. Methodol.* Available at http://doi.acm.org/10.1145/2211616.2211619.

[10] Philippa Gardner, Sergio Maffeis & Gareth David Smith (2012): *Towards a program logic for JavaScript*. In: *POPL*, pp. 31–44. Available at http://doi.acm.org/10.1145/2103656.2103663.

[11] Zef Hemel, Danny M. Groenewegen, Lennart C. L. Kats & Eelco Visser (2011): *Static consistency checking of web applications with WebDSL*. *J. Symb. Comput.* 46(2), pp. 150–182. Available at http://dx.doi.org/10.1016/j.jsc.2010.08.006.

[12] Daniel R. Licata & Shriram Krishnamurthi (2004): *Verifying Interactive Web Programs*. In: *ASE*, pp. 164–173. Available at http://doi.ieeecomputersociety.org/10.1109/ASE.2004.10054.

[13] Michael C. Martin, V. Benjamin Livshits & Monica S. Lam (2005): *Finding application errors and security flaws using PQL: a program query language*. In: *OOPSLA*, pp. 365–383. Available at http://doi.acm.org/10.1145/1094811.1094840.

[14] Markus Müller-Olm, David A. Schmidt & Bernhard Steffen (1999): *Model-Checking: A Tutorial Introduction*. In: *SAS*, pp. 330–354. Available at http://dx.doi.org/10.1007/3-540-48294-6_22.

[15] Lawrence C. Paulson (1999): *Inductive Analysis of the Internet Protocol TLS*. *ACM Trans. Inf. Syst. Secur.* 2(3), pp. 332–351. Available at http://doi.acm.org/10.1145/322510.322530.

[16] Anders P. Ravn, Jirí Srba & Saleem Vighio (2010): *A Formal Analysis of the Web Services Atomic Transaction Protocol with UPPAAL*. In: *ISoLA (1)*, pp. 579–593. Available at http://dx.doi.org/10.1007/978-3-642-16558-0_47.

[17] Jan Rutten & Daniele Turi (1994): *Initial Algebra and Final Coalgebra Semantics for Concurrency*.

[18] James Smith (2013): *Concur - An Algorithm for Merging Concurrent Changes without Conflicts*. Available at http://arxiv.org/abs/1303.7462.

[19] James Smith (2013): *State-event based versus purely Action or State based Logics*. Available at http://arxiv.org/abs/1303.7459.