Modeling Web Applications Infrastructure with ASMs

Vincenzo Gervasi, Egon Börger, Antonio Cisternino

Dipartimento di Informatica, Università di Pisa
Largo B. Pontecorvo 3
56123 Pisa, ITALY

Abstract

We describe via Abstract State Machines the major ingredients of contemporary web applications: a web browser running JavaScript programs and a web server dispatching requests to one of several modules, each one representing a class of established web application frameworks.

The web browser model comes in four levels, namely transport, stream, context and browser level, and is focussed on the interaction with possibly multiple servers (which requires a concurrent computation model) and on script execution (which requires a dynamic assignment of agents to programs). The server model is focussed on the Request-Reply pattern, and specifies a delegation strategy where the handling of a request is entrusted to a module. We show how several major frameworks for web applications can be described as progressive refinements of a number of basic modules. Three modules are further detailed: static file transfer, CGI and generic scripting modules.

Keywords: Web applications, Abstract State Machines, JavaScript

1. Introduction

In software engineering the term 'application' traditionally refers to a specific program or process users can invoke on a computer. The emergence of distributed systems and in particular of web applications has significantly changed this meaning of the term. Here functionality is provided by a set of independent cooperating modules with a distributed state. In web applications, all these modules are offering a unified interface to their user — namely, as a web page — to the point that the user may have no way to distinguish whether a single application or a set of distributed web applications is being used.

There is still no precise general definition or model of what a web application is. What is there is a variety of (often vague and partly incompatible) standards, web service description languages at different levels of abstraction

Email addresses: `gervasi@di.unipi.it` (Vincenzo Gervasi), `boerger@di.unipi.it` (Egon Börger), `cisterni@di.unipi.it` (Antonio Cisternino)

(like BPEL, BPMN, workflow patterns, see [13] for a critical evaluation of the latter two) and difficult to compare techniques, architectures and frameworks offered for implementations of web applications, ranging from CGI (Common Gateway Interface [31]) scripts to PHP (Personal Home Page) and ASP (Application Server Page) applications and to frameworks such as ASP.NET [26] and Java Server Faces (JSF [1]). All of them seem to share that a web application consists of a dynamically changing network of systems that send and receive data through the HTTP protocol to and from other components and provide services of all kinds which are subject to continuous change (as services may become temporarily or permanently unavailable), to dynamic interference with other services (competing for resources, suffering from overload, etc.) and to all sorts of failures and attacks.

1.1. Overall goal

The challenge we see is to discover and formulate the pattern underlying such client-server architectures for (programming and executing concurrent distributed) web applications. We want to make their common structural aspects explicit by defining precise high-level (read: code, platform and framework independent) models for the main components of current web application systems such that the major currently existing implementations can be described as refinements of the abstract models. The goal of such a rational reconstruction is to make a rigorous mathematical analysis of web applications possible, including to precisely state and analyze the similarities and differences among existing frameworks, e.g. the similarities between PHP and ASP and the differences between PHP/ASP and JSP/ASP.NET. This has three beneficial consequences: (a) it helps web application analysts to better understand different technologies before integrating them to make them cooperate; (b) it builds a foundation for content-based certifiability of properties one would like to guarantee for web applications; (c) it supports teachers and book authors to provide an accurate organic bird's perspective of a significant area of current computer technology.

For the present state of the art, given the lack of rigorous abstract models of (at least the core components of) web application frameworks, it is still a theoretical challenge to analyze, evaluate and classify web application systems along the lines of fundamental behavioral model properties which can be accurately stated and verified and be instantiated and checked for implementations.

1.2. Approach

The modeling concepts one needs to work on the challenge become clear if we consider the above mentioned feature all web applications have in common, namely to be an application whose interface is presented to the user via a web browser, whose state is split between one or more clients and one or more servers and where the only interaction between client and server is through the HTTP protocol. This implies that an attempt to abstractly model web application frameworks must define at least the following two major client-server architecture components with their subcomponents and the communication network supporting their interaction:

- the browser with all its subcomponents: launcher, netreader, (html, script, image) parsers, script interpreter, renderer, etc.
- the server with its modules providing runtimes of various programming languages (e.g. PHP, Python [2], ASP, ASP.NET, JSF),
- the asynchronous network which supports the interaction (in particular the communication) between the components.

This calls for a modeling framework with the following features:

- A notion of *agents* which execute each their (possibly dynamically changing) program concurrently, possibly at different sites.
- A notion of *abstract state* covering design and analysis at different levels of abstraction (to cope with heterogeneous data structures of the involved components) and the distributed character of the state of a web application.
- A sufficiently general *refinement method* to controllably link (using validation and/or verification) the different levels of abstraction, specifically to formulate different existing systems as instances of one general model.
- A flexible mechanism to express forms of *non-determinism* which can be restricted by a variety of constraints, e.g. by different degrees of transmission reliability ranging from completely unreliable (over the Internet) to safe and secure (like for components running on one isolated single machine).
- A flexible *environment adaptation mechanism* to uniformly describe web application executions modulo their dependence on run-time contexts.
- A smooth *support for traceable model change* and refinement changes due to changing requirements in the underlying (often de facto) standards.

Based on previous experiences, we find that Abstract State Machines offer most of the modeling concepts needed for our goal, and in addition are supported by a long list of success stories (see [15] for partial list) and by a sizeable amount of tools that could allow simulating and verifying our models. ASMs are thus our framework of choice for the present work.

1.3. Outline of the paper

In this paper[1], we first present a model of a web browser (Section 2), focusing on the two critical aspects of interaction with one or more servers, and of execution of scripts. Both of them require a sophisticated handling of concurrent execution, which is provided in our model by the notion of agents and by the dynamic assignments of programs to agents, and a sophisticated notion of state, sufficient to represent the Document Object Model of a web page as well as both source code and runtime state for the execution of JavaScript scripts, which is provided by the ASM notion of abstract state.

[1]The present paper is an extension and re-elaboration of [21] and [10].

For our web browser, we use a layered approach, with functionality organized across different levels: transport, stream, context and browser. The functionality described, while far from being a complete model of all aspects of a web browser, suffices to model a complete round of the Request-Reply pattern [22, 8] that characterizes browser/server interactions (see Fig. 1).

Then (Section 3) we present a model of a web server, focusing on the server-side view of the same Request-Reply pattern. We describe in particular how requests are dispatched to appropriate modules, and present – as representatives of their respective classes – three modules: static file transfer, CGI (refined to FastCGI), and a generic scripting module (that can be refined to ASP, PHP or JSP scripting, or to the more complex JSF or ASP.NET approaches).

Some conclusions and reflections on future work in Section 5 complete the paper.

2. Modeling a web browser

Space constraints as well as discourse economy prevent us from presenting a full formal specification of HTML 5 [32]. Such an effort would probably be excessive for our purposes: we are more interested in modeling only those aspects of a browser that render it a suitable thin client for web applications. As such, we will often skip the more contrived details, ignore the issues with graphics and layout entirely, and at times describe a specific implementation where the specification would allow several possibilities[2].

Our model consists of several layers. At the basis, we have a *transport-level* layer, where we describe (rather abstractly) the TCP/IP communication according to the HTTP protocol which relates a web server and a web browser[3].

On top of the transport layer, we have a *stream-level* layer, with individual agents in charge of receiving and interpreting information coming from the network. These agents are instantiated dynamically, and roughly correspond to the multiple threads that are often found in real browsers.

Above the stream level, a *context-level* layer defines the behavior of a *browsing context*. A browsing context typically corresponds to a single *Document* (which in turn has a *DOM* or Document Object Model) and regulates the user interaction with the same Document. Most typically, each window, tab, or frame in a web browser is a different browsing context.

Finally, on top of all this we have a *browser-level* layer, where we specify (in part) the behavior of a web browser, seen as an application of the host operating system. At this level we describe initialization of new browsing contexts and interaction with the host operating system.

[2]In particular, several aspects of HTML 5 are unnecessarily complicated by the need to ensure that the quirks of several different implementations in widespread use are deemed conformant.

[3]The transport layer can also be used to model those ECMAScript library functions that access the network, e.g. methods of the `XMLHttpRequest` object.

2.1. Notation

Providing a *complete* specification of all the different technologies involved, from the basics of point-to-point networking to the various protocols, languages, and frameworks employed by contemporary realistic web applications would be a herculean task, and moreover would hide in an unnecessary amount of details the interesting points that we want to highlight. Hence, in the following we will make somewhat liberal use of descriptions in natural language for clerical operations, and of text in $\boxed{\textsf{this style}}$ to indicate non-trivial operations that, however, have found no place in our effort, as being out of scope for the present work. One could think of such fragments as of undefined macros of which we even omit the name and signature, relying on the text to describe their purpose.

We will also use at times *meta-variables*, denoted in $\textsf{this style}$, to indicate a family of proper syntactic elements whose identifier is built by replacing the meta-variable with a value from a given set. So, for example, a family of predicates $\textit{has}\textsf{Attrib} : \textit{Element} \rightarrow \textit{Boolean}$ would stand for the whole set of predicates *hasName*, *hasId*, *hasStyle*, *hasSrc* etc. Other notation is as popularized by standard ASM practice, e.g. as used in [15].

2.2. Transport layer

The transport layer models the sending (and receiving) of raw data between agents residing on different hosts via HTTP over TCP/IP. Here we define the abstract state and macros needed to model a Request/Response exchange via HTTP; the interpretation of the data exchanged is then left to the stream layer in Section 2.3.

2.2.1. Channels and buffers

At the basis of the transport layer we have the concept of *Channel*s. A Channel consists of a pair of queues called *Buffer*s, each Buffer serving as a send-queue for the sending machine, and as a receive-queue for the receiving machine. There is an underlying expectation that what is written in the send-queue on one end of a channel, will appear, in order, in the receive-queue on the other end of the channel, and vice versa. However, this is not specified in our model, and in fact which data is read from a channel is, formally, totally non-deterministic. This allows us to reason on fringe cases, including communication errors, dropped connections, man-in-the-middle attacks, transparent and filtering proxies along the route, browser plug-ins to remove advertisement, antivirus and anti-scam software, parental filters, etc.

It is interesting to notice that the existence of all those potential intermediaries in practice makes TCP/IP's guarantee of "a data packet will arrive either intact and in order, or not arrive at all" [17] inapplicable in our case. In allowing for non-determinism in data transfer, we explicitly model our renounce to that guarantee.

Buffers transfer data as sequences of octets (bytes), but to simplify our models, we will assume that our background contains functions to turn these sequences of octets into the corresponding abstract types. So, for example, we will assume the ability to recognize a whole HTML element such as `<FONT size=1>` without going into the details of how the character sequence is transformed into an HTML element. Similarly for other data types (images, scripts, etc.).

On Buffers, we assume (as part of our background) the following operations:

- The macro TCPSEND(*host*, *data*, *buffer*) will initiate a network transfer of the given *data* to the *host* (which includes address and port), preparing to receive a reply, if any, through the *buffer*. It models the act of creating a socket, binding it to an address and writing a data packet to the socket, and eventually reading the reply in a given memory buffer.
- x*Available* : *Buffer* $\to$ *Boolean* a family of predicates that return true if a full data element of type x is available for reading from the head of a buffer, or false otherwise.
- *head*X : *Buffer* $\to$ X a family of functions that return the data element of type X that is available for reading at the head of a buffer, or **undef** if no data is available.
- The command **dequeue** *e* **from** *buffer* is used to remove data element *e* from the head of the buffer (thus updating the buffer).
- *isFinished* : *Buffer* $\to$ *Boolean* a predicate that is true if the transfer associated to a buffer is finished and no more data have to be expected. This situation corresponds in actual implementation to a `close` operation on a TCP socket.

Notice that our transport-level background is fit to describe full TCP or UDP exchange, whereas the macros and functions defined above are sufficient to describe typical HTTP interactions (from the browser's perspective).

2.2.2. HTTP Request/Response

The HTTP protocol specifies that Requests should be sent to compatible servers in a specific format. First is a compulsory *request line* including a *method* (one of GET, POST, PUT, HEAD, and a handful of others), a *resource* (most typically, a pathname with optional query parameters), and protocol versioning information. The request line is followed by a (possibly empty) sequence of *headers* each of which is a pair (*key*, *value*), and by an optional *body*, containing arbitrary *data* to be processed by the server. Headers and body are separated by an empty line.

In the following we associate a unique identifier k to each request/response pair; we will see later how k can also serve to associate a request/response pair to higher-level operations and data. Moreover, since network transfers happen asynchronously, we use a *callback pattern*, where the response to a given request will be processed at some future time by a machine *proc* which is provided with the request.

The lexical details of how a Request is encoded need not concern us here; the actual sending of a request is modeled through the following macro, where *host* represent the (abstract) network identity of the machine that will receive the request, *head* includes the request line and headers, and *data* is as defined above:

```
$\textsc{Send}(host, head, data, proc, k) =$
  let $buffer =$ new $Buffer$, $a =$ new $Agent$ in
    $ag(k) := a$
    $buf(k) := buffer$
    $\mathrm{TCP}\textsc{Send}(host, head \cdot EMPTYLINE \cdot data, buffer)$
    $mode(k) := ExpectStatus$
    $program(a) := \textsc{Receive}(proc, k)$
```

The macro above creates a new buffer to hold the server response, constructs an HTTP Request by joining head and body, and more importantly creates a new agent whose task is to (eventually) process the response through the callback *proc* (once completed, the agent will terminate):

```
$\textsc{Receive}(proc, k) =$
  if $mode(k) = ExpectStatus$ then
    if $lineAvailable(buf(k))$ then
      let $l = headLine(buf(k))$ in
        dequeue $l$ from $buf(k)$
        $mode(k) := ExpectHeader$
        $status(k) := l$
  if $mode(k) = ExpectHeader$ then
    if $lineAvailable(buf(k))$ then
      let $l = headLine(buf(k))$ in
        dequeue $l$ from $buf(k)$
        if $isEmptyLine(l)$ then $mode(k) := ExpectData$
        if $isSetCookie(l)$ then $\forall cookie \in l$, $\textsc{StoreCookie}(cookie, rurl(k))$
        else $\boxed{\text{manage other headers, e.g. for cache control}}$
  if $mode(k) = ExpectData$ then
    $proc(k)$
```

The program for processing the data portion of the response is provided by the caller of the macro (and hence, eventually, by the initiator of the transfer). All elements are bound together through the unique key k, that serves as the unique identifier for this particular HTTP interaction. Notice how Receive stores any cookie sent by the server in a global (not described here) storage, whence they will be retrieved by the *cookiesFor*() function used in the next macro.

A full HTTP transfer is initiated by invoking the Transfer macro below:

```
$\textsc{Transfer}(method, url, data, proc, k) =$
  $rmethod(k) := method$
  $rurl(k) := url$
  $rdata(k) := data$
  if $protocol(url) = \texttt{http}$ then
    let $cookies = cookiesFor(url)$,
    $hheader = makeHeader(method, url, cookies)$,
    $hdata = makeData(data)$,
    $host = addressFor(url)$ **in**
      $\textsc{Send}(host, hheader, hdata, proc, k)$
  **else**
    $\boxed{\text{other forms of transfer, e.g. file, ftp, etc.}}$
```

The macro first saves the parameters that characterize the request into state location indexed by the unique key k (for possible later reference, e.g. in error messages), then obtains the set of stored cookies that match the given URL; finally it builds the HTTP header by combining the method, the URL, and the cookies via the *makeHeader* function, and analogously builds the body of the request via the *makeData* function (which in real implementations performs, among other processing, the base-64 encoding of the binary data provided with the request). The destination host is identified by parsing the provided URL via the *addressFor* function, then the full HTTP request is sent to the network as described above.

Notice that our TRANSFER is a simplified version of the fetching algorithm described in full in [32, §2.7].

We hide here the details of how cookies are stored and retrieved by the browser (e.g., in a file on the user's home directory) into abstract functions and macros, yet it might be noticed that we are assuming a locking mechanism for the cookie storage, provided by the underlying file system or operating system, since multiple transfers can be occurring at the same time. In practice, most browsers in widespread use would rely on file system locks to ensure that multiple threads concurrently trying to write and retrieve cookies from a common storage would not interfere with each other in unexpected ways.

2.3. Stream layer

The transport layer described how HTTP requests are sent out, and how responses are streamed into a Buffer. Here we describe how these streams are interpreted upon reception by showing a number of *stream processor* submachines. These machines receive incoming data in a buffer, in a streaming fashion (that is, the data is made available piecemeal, as soon as it is obtained from the network), and they incrementally process it in a variety of ways.

We split each stream processor in two sub-layers: the *processor* proper discriminates between the various *return codes* returned in the response, and – depending on whether the request was successful, or an error was returned, or other special actions need to be taken – executes the appropriate rule. In case of a successful request, the actual processing of the data returned is delegated to a specialized *parser*.

2.3.1. HTML streams

The most important stream processor is the HTML one, whose main task is to parse an HTML document and build the corresponding DOM (Document Object Model).

HTML processor. The HTML processor dispatches the handling of successful transfers to an HTML parser (described in the next section), whereas error codes are handled by an abstract macro that we will not further detail (typically, a synthetic "error page" is presented to the user; this would be simply modeled by replacing the current Document with a prepared one), and redirections are processed by restarting the transfer with a new URL (which is provided as part of the response itself).

```
HTMLProc($k$) =
  if $isSuccessCode(status(k))$ then
    HTMLParser($k$)
  elseif $isErrorCode(status(k))$ then
    HandleHTMLError($k$)
  elseif $isRedirectCode(status(k))$ then
    RestartTransfer($k$)
  else
    $\boxed{\text{handling of other return codes}}$
```

HTML parser. In the following, we will assume the existence of a *DOM Tree*, whose elements are *Node*s. An exact specification of the contents of this tree, and of how the various nodes are build, is outside the scope of this document; the interested reader can however refer to [32, §1.8] and [32, §2.1.3] for a quick introduction. Here, we assume that navigation functions (dynamic functions such as *parent*(), *firstChild*(), *nextSibling*(); derived functions such as *root*(), *lastChild*(), etc.) are always available and describe the intended structure of the tree. We also assume that there is a *current tree* and a *current node* while the tree is being built; the abstract macros AddText, AddChild etc. modify the node data and navigation functions of the current tree as expected (these macros are detailed later).

Finally, we assume a range of functions over nodes to access their attributes and embedded content (e.g., the text contained in a CTEXT node); their usage in the following will be clear from context.

The machine below highlights three aspects of the HTML parsing process (which are among the most relevant ones for web applications): building the DOM tree, loading further resources, and executing scripts. These aspects will be illustrated in the following subsections. We will instead glide over other issues such as handling of malformed content, converting different character encodings, and applying style sheets, since these do not normally[4] affect the execution of well-behaved web applications.

We define the following macro for parsing HTML contents; its operations are explained immediately after the definition:

[4]Notice that techniques such as using a style sheet to hide a certain UI component, thus preventing the normal user from issuing certain UI commands to the application, are not to be considered among the best practices. In fact, user agents (such as web browsers) can ignore or allow the user to override such style specifications, regaining control of the hidden elements, to unforeseen effects.

```
HTMLParser($k$) =
  if $\neg paused(k)$ then
    if $textAvailable(buf(k))$ then
      let $t = headText(buf(k))$ in
        dequeue $t$ from $buf(k)$
        AddText($t, curNode(k)$)
    if $tagAvailable(buf(k))$ then
      let $e = headTag(buf(k))$ in
        dequeue $e$ from $buf(k)$
        if $isOpeningTag(e)$ then
          let $n = newNodeFor(e)$ in
            AddChild($n, curNode(k)$)
            if $\neg isClosingTag(e)$ then $curNode(k) := n$
            match $e$
              case <SCRIPT src=url> :
                Transfer($GET, url, \langle\rangle$, ScriptProc, $n$)
              case <IMG src=url> :
                Transfer($GET, url, \langle\rangle$, ImageProc, $n$)
              case <LINK rel=rel src=url> :
                if "stylesheet" $\in rel$ then
                  Transfer($GET, url, \langle\rangle$, StylesheetProc, $n$)
              case [similar nodes that require a background transfer]
      if $isClosingTag(e)$ then
        if $\neg isOpeningTag(e)$ then $curNode(k) := parent(curNode(k))$
        match $e$
          case </SCRIPT> :
            if $isAsync(curNode(k))$ then StartAsync($curNode(k), k$)
            elseif $isDeferred(curNode(k))$ then AddDeferred($curNode(k), k$)
            else RunImmediate($curNode(k), k$)
          case [similar nodes which require post-processing]
  if $isFinished(buf(k))$ then
    if $hasDeferred(k)$ then
      RunDeferred($k$)
    else
      FinalizeLoading($k$)
      $program(self) :=$ undef
```

Building the DOM tree. In building the DOM tree, the HTMLParser assumes that, at every instant, there is a current node *curNode*(k) (where k is the unique instance token of the parser) which is the parent of the content that is currently being parsed. For example, in the following stream of HTML code, we have marked below each element the corresponding *curNode* at the time of parsing the element:

$$\texttt{<DIV>}\underbrace{\texttt{<FONT size=1>}}_{\texttt{<DIV>}}\underbrace{\texttt{Sample <B>}}_{\texttt{<FONT size=1>}}\underbrace{\texttt{text</B>}}_{\texttt{<B>}}\underbrace{\texttt{ here</FONT>}}_{\texttt{<FONT size=1>}}\underbrace{\texttt{</DIV>}}_{\texttt{<DIV>}}$$

For a new transfer, $curNode(k)$ is initialized at the root element of an empty DOM (this is performed by the browser when, for example, opening a new tab: see macro PageLoad in Section 2.4.2).

The HTML parser uses a number of predicates and functions which are informally defined in the following.

- AddText$(text, node)$ appends a section of *text* to the contents of a *node*.

  ```
  AddText$(text, node)$ =
    **if** $ctext(node)$ =**undef then**
      $ctext(node) := text$
    **else**
      $ctext(node) := ctext(node) \cdot text$
  ```

  This is most commonly the case for the content text of web pages. Notice that the text will be appended, in order, in chunks as it arrives from the network. Also, we assume that *headText* takes care of converting named entities (such as `&eacute;`) into the corresponding symbol (`é`) and of other encoding conventions, including tokenization as documented in [32, §8.2.4].
- $isOpeningTag : Element \to Boolean$ and $isClosingTag : Element \to Boolean$ are two predicates that indicate if a given tag is an opening tag (e.g., `<FONT>`) or a closing tag (e.g., `</FONT>`); notice that both could be true of the same tag (e.g., `<IMG ... />` or empty elements such as `<BR>`), whereas at least one of the two must be true for any given tag.
- $newNodeFor : Element \to DOMNode$ builds a fresh node, setting appropriate dynamic functions based on the supplied element, so that later it will be possible to retrieve the tag name, the value of its attributes, etc.
- AddChild$(child, parent)$ add the *child* node to the DOM tree, as the last child of *parent*.

  ```
  AddChild$(c, p)$ =
    **if** $firstChild(p)$ =**undef then** $firstChild(p) := c$
    **else let** $last = lastChild(p)$ **in**
      $nextSibling(last) := c$
    $parent(c) := p$
  ```

- The **match** construct we use is intended as a short-hand for compare & bind sequences. For example,

  ```
  **match** $e$
    **case** <SCRIPT src=*url*> : ...
  ```

  is a shorthand for

  ```
  **if** $tagName(e) = \texttt{SCRIPT} \wedge hasAttribute(e, \texttt{src})$ **then**
    **let** $url = valueOfAttribute(e, \texttt{src})$ **in** ...
  ```

The process we described in HTMLParser suffices for our purposes, but the reader should keep in mind that the full specification for building the DOM tree in [32, §8.2.5] includes a large number of other special cases (which, however, are not peculiar to the execution of web applications, and hence are out of scope for the present work).

Loading of external resources. While most HTML elements include references to external resources, only a few of the latter are automatically loaded at the same time as the page itself is. This is in particular the case of images, scripts, and style sheets, and also of less-used resources such as audio, video, embedded objects, etc.

These cases are handled by the **match** construct in HTMLParser. When a tag that requires background loading of further resources is encountered, the Transfer macro is invoked, retrieving the given URL and processing the retrieved contents through the appropriate parser.

It is important to stress that these transfers happen *concurrently*, without pausing or interrupting the transfer of the main HTML page that is being performed. In fact, after the HTTP request to retrieve them has been sent out, the processing of the results is delegated to a new agent, different from the one that is processing the page. These agents receive as their context (through the unique transfer key) the DOM node that caused the transfer, n. Based on our model, n is not yet added to the DOM tree when the Transfer macro is called (since its execution happens in the same step as the execution of AddChild), but it will be properly installed by the time the response for the transfer is parsed, since that will happen in subsequent steps[5].

Script execution. The last element in the HTML parser concerns the execution of scripts encountered in the page, either as embedded scripts, or referenced through an external URL.

Traditionally, script execution in HTML pages was strictly *serialized*, and neither the JavaScript language, nor any interpreter in common use, supported any form of multi-threading. Moreover, execution was *blocking*: since a script could generate parts of the document on-the-fly, which were to be textually inserted immediately after the script itself, and before any other contexts, or even cause a redirection – thus halting the loading of the page entirely –, it was not possible to overlap execution and page loading.

The current HTML standards however, allows three different modes of script execution: *synchronous* or *immediate*, *deferred* and *asynchronous*. We describe each mode in turn in the following.

Immediate execution. The first mode of execution (and the default one, unless the `async` or `deferred` attributes of the `<SCRIPT>` tag are given) is the synchronous or immediate execution.

```
RunImmediate($node, k$) =
  $paused(k) := true$
  match $type(node)$
    case text/javascript :
      ECMAScriptInterpret($contents(node), node,$ RunCompleted, $k$)
```

[5]In particular, it cannot happen in the current step since the agent having the parser as its program will not exist yet.

```
        case $\boxed{\text{specific versions and other languages handled similarly}}$
```

There is a need to pause the HTML parser while executing a script in immediate mode. In fact, due mostly to historical legacy from the initial implementations of JavaScript, it is possible to write from a script parts of the document to be parsed, i.e. generate dynamically the page itself (including, possibly, further `<SCRIPT>` tags that would then be executed in turn). While this technique offers significant flexibility, at the same time it clearly impedes continuing the parsing of the page till the execution of the script is complete. This is obtained in our model by pausing the parser via the $\mathit{paused}(k)$ function.

Along the same lines, there is a need to restart the parser once the execution is complete. This is obtained by passing to the interpreter a *callback* macro and a token parameter k (this is the same technique that we used in Receive). In this case, the callback will be

```
RunCompleted($k$) =
  $\mathit{buf}(k) := \mathit{documentWriteBuffer}(k) \cdot \mathit{buf}(k)$
  $\mathit{paused}(k) := \mathit{false}$
```

The typical method for generating HTML contents to be injected into the page from a script is through the `document.write()` method (see [32, §3.5.3]). While the specification more accurately describes the processing to be performed in this case, we will be satisfied by postulating that all output generated by `document.write()` and `document.writeln()` during the executing of a given script is collected, in order, in $\mathit{documentWriteBuffer}(k)$ and prepended to $\mathit{buf}(k)$ at the end of the execution of the script, but prior to resuming parsing the HTML source.

It is worth remarking that the ECMAScriptInterpret might have to wait till the script has finished loading prior to actually starting the execution, in case its source text is obtained by a Transfer (this will be better illustrated in 2.3.2).

Deferred execution. Deferred execution consists in postponing the execution of a script until the page is fully loaded. This is accomplished by storing in a set of pending scripts the information needed for later execution:

```
AddDeferred($\mathit{node}, k$) =
  enqueue $\mathit{node}$ to $\mathit{deferred}(k)$
```

The predicate $\mathit{hasDeferred}()$ tells whether a certain document has pending scripts:

$$\mathit{hasDeferred}(k) \;=\; \mathit{deferred}(k) \neq \emptyset$$

In that case, the scripts are executed, sequentially and in-order, at the end of page loading and parsing:

```
RunDeferred($k$) =
  let $\mathit{node} = \mathit{head}(\mathit{deferred}(k))$ in
    dequeue $\mathit{node}$ from $\mathit{deferred}(k)$
    RunImmediate($\mathit{node}, k$)
```

Given our previous definitions, this is sufficient to pause the parser (and, with it, the initiation of further executions from the deferred set), and properly serialize the execution of all pending scripts until the *deferred*(k) queue is empty.

Asynchronous execution. For asynchronous scripts, we need not pause the HTML parser; script execution and further parsing of the HTML document can proceed concurrently, subject to proper mutual exclusion when accessing shared data (mostly, the DOM tree itself). In our model, the locking protocol is abstracted into the AddText and AddChild macros used by the parser.

As a result, starting an asynchronous script execution is remarkably similar to the immediate execution, with the proviso that the parser is not paused. Notice that, since ECMAScriptInterpret provides its own agent to execute the interpreter, no agent creation is needed here.

```
StartAsync($node, k$) =
  **match** $type(node)$
    **case** text/javascript :
      ECMAScriptInterpret($contents(node), node$, **skip**, $k$)
    **case** $\boxed{\text{specific versions and other languages handled similarly}}$
```

Final processing. When the entire DOM tree has been created and all deferred scripts have finished execution, the parser *fires* a number of events, namely: `DOMContentLoaded`, `load`, `pageshow`[6]. This processing is abstracted in the FinalizeLoading macro that we do not detail here[7].

2.3.2. Script streams

Script streams are used whenever the browser needs to load the source code for a script from a remote URL, which typically happens when a `<SCRIPT src=`*url*`>` element is processed while parsing HTML data.

Script processor. Processing scripts is similar to other forms of stream processing, in that the data is accumulated in case of a successful transfer, whereas errors (e.g., attempts to load a script from a non-existing URL) will simply result in an empty content. This is different from the behavior of the HTML processor, which would notify the user in case of a failure in loading a page.

```
ScriptProc($k$) =
  **if** $isSuccessCode(status(k))$ **then**
    ScriptParser($k$)
  **elseif** $isErrorCode(status(k))$ **then**
```

[6]For malformed documents, that we do not consider here, further processing is performed to close all unclosed tags, firing corresponding `popstate` events.

[7]These events would be enqueued in the main event queue of the browsing context (Section 2.4), and retrieved and processed in the EventLoop macro (Section 2.4.2). Full details are in [32, §8.2.6].

```
    $programText(k) := “”$
  **elseif** $isRedirectCode(status(k))$ **then**
    RESTARTTRANSFER$(k)$
  **else**
    $\boxed{\text{handling of other return codes}}$
```

Script parser. The script parser is invoked to process the source text of a script, while it is being received from the network following an occurrence of a `<SCRIPT src=`*url*`>` tag in the page.

Although the precise rules for the encoding of script source text are subtly different than those for general text, as specified in [32, §4.3.1.2], we will simplify the matter here[8] and use the same functions we already use for general text, as follows:

```
ScriptParser$(k) =$
  **if** $textAvailable(buf(k))$ **then**
    **let** $t = headText(buf(k))$ **in**
      **dequeue** $t$ **from** $buf(k)$
      $programText(k) := programText(k) \cdot t$
  **if** $isFinished(buf(k))$ **then**
    $complete(k) := true$
    $program(self) :=$**undef**
```

Notice that in our model the execution of a script can *start* while the script is still loading, but will certainly not *finish*, and may not progress at all, until *complete*() signals that loading has finished and the full text of the program is available. This means that in the case of asynchronous execution, the agent created by ECMASCRIPTINTERPRET will idly execute **skip** steps as long as $complete(k)$ is false, and only upon $complete(k)$ becoming true, will it execute the actual interpretation machinery. The specification allows for varied behavior in this respect (e.g., an implementation could start building the parse tree incrementally while the loading is still in progress, or postpone any processing to after the full program has been received). Our model provides the information needed to implement the different variants, but hides the actual choice inside the ECMASCRIPTINTERPRET macro.

2.3.3. Image streams

Image streams are used when receiving images from the server, as part of a web page, or via instantiation of the corresponding classes in the JavaScript library.

[8]Notice that we already applied the same principle in loading in-line script source as text for `<SCRIPT>` nodes without the `src` attribute.

Image processor. The behavior in case of a successful transfer is analogous to that of previous stream processors (i.e., the Image parser is run). Erroneous cases are handled differently, typically by substituting a pre-defined error image (e.g., a large red X or an icon depicting a broken link) for the missing image.

```
ImageProc($k$) =
  if $isSuccessCode(status(k))$ then
    ImageParser($k$)
  elseif $isErrorCode(status(k))$ then
    $imgData(k) := errorImgData$
  elseif $isRedirectCode(status(k))$ then
    RestartTransfer($k$)
  else
    $\boxed{\text{handling of other return codes}}$
```

Image parser. The parsing of image data is often done incrementally, in order to properly implement so-called *progressive* image formats (i.e., when data are arranged in such a way that it is possible to construct a low-quality version of an image early in the loading stage, and then refine that to a better resolution or color depth as more data arrives) or to show load progress (i.e., by updating the rendered image on a scan-line basis as soon as data is available). We abstract from all the details of various image formats, and from how the browser distinguishes them based on their MIME types. The only aspect that we want to highlight is the progressive nature of the loading, since it expresses the fact that the graphical user interface of the web application may be not fully loaded when the application code starts executing.

The general process of loading an image is thus described as follows:

```
ImageParser($k$) =
  if $dataAvailable(buf(k))$ then
    let $d = headData(buf(k))$ in
      dequeue $d$ from $buf(k)$
      $imgData(k) := imgData(k) \cdot data$
  if $isFinished(buf(k))$ then
    $program(self)$ :=undef
  UpdateImage($k$)
```

Here, we assume that UpdateImage(k) will perform any needed update to the internal data structures holding the actual image, based on $imgData(k)$. Notice that the macro is invoked at each step even if no new data has been received; this allows the implementation to perform any timed decoding or animation (e.g., a hourglass or spinning circle or progress bar) to indicate loading progress.

2.4. Context layer

A *browsing context* is an environment in which documents are shown to the user, and where interaction with the user occurs. In web browsers, browsing contexts are usually associated with windows or tabs, but certain deprecated HTML structures (namely, frames) also introduce separate browsing contexts.

In our model, a browsing context is characterized primarily by five elements:

- a *document* (i.e. a DOM as described in Section 2.3.1), which is the currently active document presented to the user;
- a *session history*, which is a navigable stack of documents the user has visited in this browsing context;
- a *window*, which is a designated operating system-dependent area where the Document is presented and where any user interaction takes place;
- a *renderer*, which is a component that produces a user-visible graphical rendering of the current Document (Section 2.4.1) in the corresponding window;
- an *event loop*, which is a component that receives and processes in an ordered way the various operating system-supplied events (such as user interaction or timer expiration) that serve as local input to the browser (Section 2.4.2).

We keep the *window* abstract, as its behavior can be conveniently hidden by keeping the actual rendering abstract as well and by assuming that user interaction with the window is handled by the operating system. Thus we deal with events that have been already pre-processed by a window manager. Due to space limits, we also omit the rather straightforward modeling of the *session history*.

When starting a newly created Browsing Context k, $DOM(k)$ is initialized by a pre-defined implementation-dependent initial document *initialDOM*; it is usually referred to through the URL `about:blank` and may represent an empty page or a "welcome page" of some sort. Two agents are equipped with programs to execute the Renderer and the EventLoop for k.

```
StartBC(k) =
  let a = new Agent, b = new Agent in
    program(a) := Renderer(k)
    program(b) := EventLoop(k)
    DOM(k) := initialDOM
    agents(k) := {a, b}
```

To stop a running browsing context, the dual steps have to be taken:

```
StopBC(k) =
  forall agent in agents(k) do
    program(agent) := undef
  DOM(k) := undef
  agents(k) := {}
```

It is also understood that stopping a browsing context will also close the associated window/tab provided by the OS windowing system, if any.

The Renderer and EventLoop macros used in StartBC are specified below.

2.4.1. Renderer

The Renderer produces the user interface of the current *DOM* in the (implicit) corresponding window. It is kept abstract by specifying only that it works when it is (a) supposed to perform (at system dependent *RenderingTime*) and (b) allowed to perform because no other agent has a lock on the *DOM* (e.g., not blocked because the HTMLParser is adding new nodes to the DOM during the stream-level loading of an HTML page).

```
Renderer($k$) =
  if $renderingTime(k)$ and $\neg locked(DOM(k))$ then
    GenerateUi($DOM(k), k$)
```

2.4.2. Event Loop

We assume that *events* are communicated by the host environment (i.e., the specific operating system and UI toolkit of the client machine where the browser is executed) to the browser by means of an *event queue.* These UI events are merged and put in sequential order with other events that are generated in the course of the computation, e.g. DOM manipulation events (fired whenever an operation on the DOM, caused by user actions or by JavaScript operations, leads to the execution of a JavaScript handler or similar processing) or History traversal events (fired whenever a user operates on the Back and Forward buttons offered by most browsers to navigate through the page stack).

Here we detail the basic mechanism used in (the simplest form of) web applications to prepare a Request to be sent to the server (with the understanding that when a Response is received, it will replace the current page in the same browsing context). HTML *forms* are used to collect related data items, usually entered by the user, and to package them in a single Request. Figure 1 shows when the different macros we defined are invoked; lifelines represent agents executing a rule. Remember that ASM agents can change their program dynamically (e.g., when Receive becomes HTMLProc) and that operations by an agent in the same activation, albeit shown in sequence, happen in parallel.

An HTML form is introduced by a `<FORM>` element in the page. All the input elements[9] that appear in the subtree of the DOM rooted at the `<FORM>` are said to belong to that form. Among the various input elements, there is normally a designated one (whose UI representation is often an appropriately labeled button) tasked with the function of *submitting* a form. This involves collecting all the data elements in the form, encoding them in an appropriate format, and sending them to a destination server through various means. This

[9]These include elements such as `<INPUT>`, `<SELECT>`, `<OPTION>` etc.

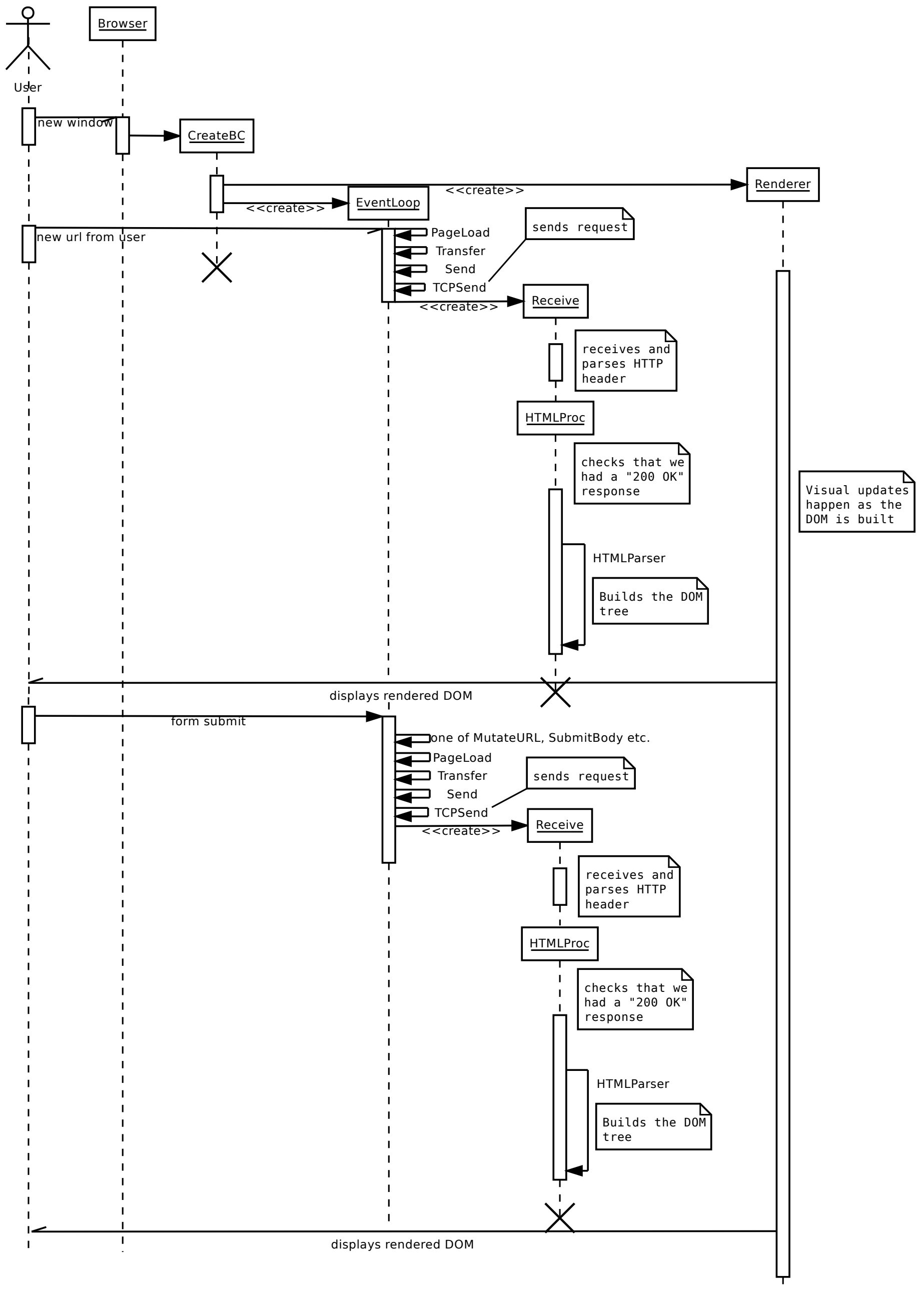

Figure 1: A diagram depicting the behavior of our browser model for a user who opens a new window in a browser, manually loads the first page of a web application, interacts locally with a form, and then sends the data back to the server, receiving a new or updated page in response.

may include sending the data by email or initiating an FTP transfer, although these possibilities are seldom, if ever, used in contemporary web applications.

It is also of interest to note that submission of a form may be initiated from a script, by invoking the `submit()` method of the form object, and hence happen independently from user behavior. In the following, we will not concern ourselves with the details of how a submit operation has been initiated, but only with the emergence of the submit event in the event queue, whatever its origin may be.

We model the existence of a separate event queue for each browsing context, which is processed by a dedicated agent created in the StartBC macro above. When an event is extracted from the event queue that indicates that the user has provided a new URL to load (e.g., by typing it in a browser's address bar, or by selecting an entry from a bookmarks list, etc.), the browsing context is navigated to the provided URL by starting an asynchronous transfer (in the normal case, the HTTP Request will be sent to the host mentioned in the URL, and later processing of the Response will replace the DOM displayed in the page).

When an event is extracted from the event queue that indicates a form submission, the form and related parameters are extracted from the event, appropriate encoding of the data is performed based on the action and method attributes as specified in the `<FORM>` node, and finally either the data is sent out (e.g., in the case of a `mailto:` action) or the browsing context is populated with the results returned from a web server identified by the form's action. In normal usage, that will be the same web server hosting the web application that originally sent out the page with the form, thus completing the loop between server and client and realizing the well-known page-navigation paradigm of web applications[10].

Similar to Renderer, our EventLoop receives a parameter, k, which identifies the particular instance. The macros MutateUrl, SubmitBody and PageLoad used in EventLoop are defined below; other macros used here are left abstract, as discussed later.

```
EventLoop($k$) =
  if $eventAvailable(eventQueue(k))$ then
    let $e = headEvent(eventQueue(k))$ in
      dequeue $e$ from $eventQueue(k)$
      if $isNewUrlFromUser(e)$ then
        PageLoad($GET, url(e), \langle\rangle, k$)
      elseif $isFormSubmit(e)$ then
        let $f = formElement(e), data = encodeFormData(f)$,
            $a = action(f), m = method(f), u = resolveUrl(f, a)$ in
```

[10]Notice that we are not considering here AJAX applications, where a Request is sent out directly from JavaScript code, and the results are returned as raw data to the same script, instead of being used to replace the contents of the page. The general processing for this case is, however, similar to the one we describe here.

```
        **match** $(schema(u), m)$ :
          **case** $(\texttt{http}, \texttt{GET})$ : $\textsc{MutateUrl}(u, data, k)$
          **case** $(\texttt{http}, \texttt{POST})$ : $\textsc{SubmitBody}(u, data, k)$
          **case** $(\texttt{ftp}, \texttt{GET})$ : $\textsc{GetAction}(u, data, k)$
          **case** $(\texttt{ftp}, \texttt{POST})$ : $\textsc{GetAction}(u, data, k)$
          **case** $(\texttt{javascript}, \texttt{GET})$ : $\textsc{GetAction}(u, data, k)$
          **case** $(\texttt{javascript}, \texttt{POST})$ : $\textsc{GetAction}(u, data, k)$
          **case** $(\texttt{data}, \texttt{GET})$ : $\textsc{GetAction}(u, data, k)$
          **case** $(\texttt{data}, \texttt{POST})$ : $\textsc{PostAction}(u, data, k)$
          **case** $(\texttt{mailto}, \texttt{GET})$ : $\textsc{MailHead}(a, data)$
          **case** $(\texttt{mailto}, \texttt{POST})$ : $\textsc{MailBody}(a, data)$
    **else**
      $\boxed{\text{handle other events}}$
```

We do not further specify here the mail-related variants MailHead and MailBody (although it is interesting to remark that they do not need further access to the browsing context k, contrary to most other methods, since no reply is expected from them – and thus their applicability in web applications is close to nil). We also glide over the possibility of using a `https` schema, which however implies the same processing as `http`, with the only additional step of properly encrypting the communication. Given the purposes of this paper we omit a definition of GetAction and PostAction, since they involve URL schemas (namely: `ftp`, `javascript` and `data`) that have not been addressed in our transport layer model. Thus, below we only refine MutateUrl and SubmitBody together with PageLoad.

The macro MutateUrl consists in synthesizing a new URL from the action and the form data (which are encoded as query parameters in the URL) and in causing the browsing context to navigate to the new URL:

```
$\textsc{MutateUrl}(u, data, k) =$
  **let** $u' = u \cdot \text{“?”} \cdot data$ **in**
    $\textsc{PageLoad}(GET, u', \langle\rangle, k)$
```

The macro SubmitBody differs only in the way the data is encoded in the request, namely not as part of the URL, as above, but as body of the request:

```
$\textsc{SubmitBody}(u, data, k) = \textsc{PageLoad}(POST, u, data, k)$
```

The macro PageLoad starts an asynchronous Transfer and (re-)initializes the browsing context and the HTMLProcessor which will handle the Response:

```
$\textsc{PageLoad}(m, u, data, k) =$
  $\textsc{Transfer}(m, u, data, \text{HTML}\textsc{Proc}, k)$
  $htmlParserMode(k) := Parsing$
  **let** $d =$**new** $Dom$ **in**
    $DOM(k) := d$
    $curNode(k) := root(d)$
```

Notice that while for the sake of brevity we have modeled navigation to the response provided by the server as a direct TRANSFER here, in reality it would require a few additional steps, including storing the previous document and associated data in the session history and releasing resources used in the original page (e.g., freeing memory for images, or stopping plug-ins that were running). While resource management can be conveniently abstracted, handling of history navigation (i.e., the Back, Forward and Reload commands available in most browsers) is a critical component in proving robustness, safety and correctness properties of web applications, and needs to be considered in any verification effort.

2.5. Browser layer

The browser layer is concerned with initiating the overall execution of the browser, and of interacting with the user for application-wide operations (such as opening a new window).

It is interesting to remark that, differently from the previous layers, the browser layer is not standardized in any way. Browser developers are free to design their applications the way they see fit, as long as each browsing context inside the application behaves in the expected way.

In practice, the most common design of a browser allows one or more independent browsing contexts (visually rendered as tabs inside a single window and/or as different windows). We will thus abstractly model this common behavior as follows:

```
BROWSER =
  if userIntention = startupBrowser then
    let newContext = new Context in
      contexts := {newContext}
      STARTBC(newContext)
  if userIntention = addContext then
    let newContext = new Context in
      add newContext to contexts
      STARTBC(newContext)
  if userIntention = closeContext(context) then
    remove context from contexts
    STOPBC(context)
  if userIntention = shutdownBrowser then
    forall context in contexts do
      remove context from contexts
      STOPBC(context)
```

In the macro above, a monitored location *userIntention* conveys, in abstract form, user commands to start or shut down the browser, and to open or close browsing contexts (i.e., windows or tabs). This location can be thought of, a bit less abstractly, as the foremost event in an event queue provided by the window manager / GUI toolkit / operating system that the browser utilizes for its user interface.

More complex commands (e.g., saving or restoring the current session) can also be modeled in the same way (e.g., by saving *contexts* in a serialized form in permanent storage).

It may be noticed that BROWSER, alone of all our macros, does not expect any parameter. Indeed, the act of launching a browser does not require further specifications (and in fact, is usually performed by a mouse click on an icon, or analogous means).

3. Modeling a web server

The next part of our modeling effort focuses on defining a high-level model of a web server (Sect. 3.2) with typical refinements for the underlying handler modules, namely for file transfer (Sect. 3.3), CGI (Sect. 3.4) and scripting modules (Sect. 3.5).

3.1. Server-side transport layer

On the server, we need to model what is called a *server socket* in TCP/IP terminology. A server socket is essentially an unique contact point for a server; multiple clients can connect to a server socket (identified by host and port number, as given by the *host* parameter of our macro TCPSEND from Section 2.2.2), and for each connection a new channel between server and client is created.

In our model we will slightly abstract from these details (which, however, could be modeled in the same spirit as the SEND mechanism defined in Section 2.2.2), and assume instead that a dedicated channel (from Section 2.2) exists between each client and our server, but that the input (from the point of view of the server) buffers of all such channels are merged into a single buffer called *requestQueue*. Each *req*uest read from the *requestQueue* will then carry sufficient background information to allow the various modules in Sections 3.3, 3.4, 3.5 to retrieve, via *requestOutput*(*req*), the output buffer associated to the original *req*uest sender, and thus send them the corresponding response.

3.2. Functional Request-Reply web server view

In the high-level view the server appears as a dispatcher which to handle a request finds and triggers the code (a 'module') whose execution will provide a response to the request[11]. Thus a high-level web server model can be formulated as an ASM WEBSERVER which in a reactive manner, upon any *req*uest in its *requestQueue*, will delegate the request to a new agent (read: a thread we call *request handler*) to handle the EXECution of the request—if the *req*uest passes

[11]The ASM model for the Virtual Provider (VP) defined in [7] has a similar structure: it receives requests, forwards them to appropriate providers and collects the replies from the providers to return them to the original requestor.

the *Security* check and the *requestedModule* is *Available* and can be loaded by the server.

We succinctly describe checking various kinds of *Property* (here access security, module availability and loadability) by functions (here *checkSecurity*, *findModule*, *loadModule*) whose values are

- either three-digit-values v in an interval $[n00, n99]$, for some $n \in [0, 9]$ as defined for each *Property* of interest in [4, Sect.4.1] to indicate that the *Property* holds or fails to hold (in the latter case of $PropertyFailure(v)$ the value v also indicates the reason for the failure), or
- some different value, like a found requested module, which implicitly also indicates that the checked *Property* holds, e.g. that the requested module is available or could be successfully loaded.

Since in case $PropertyFailure(v)$ is true, the function value v is assumed to indicate the reason for the failure, the value appears in the *failureReport* that the WEBSERVER will SEND to the client. The function *failureReport* abstracts from the details of formatting the response message out of the parameters.

The *requestedModule* depends on the server *env*ironment, the *resourceName* that appears as part of the *req*uest and the $header(req)$. For a loaded *module* STARTHANDLER creates a new thread and puts it into its *init*ial state from where the thread will start its program, which is obtained from a function *exec* on the basis of the *module* itself, and parametrized by the *module*[12], the *req*uest and the *env*ironment.

A loaded *module* is of one of finitely many kinds. We will devote the rest of this section to specify the different machines that implement behavior for file transfer, CGI that is then refined to FastCGI, generic scripting that is then refined for ASP/PHP/JSP and JSF/ASP.NET.

To reflect the functional client/server request/reply view STARTHANDLER appears as atomic action of the WEBSERVER which goes together with removing the *req*uest from the *requestQueue*. This atomicity reflects the fact that once a request has been handled, the server is ready to handle the next one.

```
WebServer =
  if $requestAvailable(requestQueue)$ then
    let $req = headRequest(requestQueue)$ in
      dequeue $req$ from $requestQueue$
      let $env = environment(server, req), s = checkSecurity(req, env)$ in
        if $SecurityFailure(s)$ then
          Send($failureReport(req, s)$)
        else
          let $requestedModule =$
            $findModule(env, resourceName(req), header(req))$ in
```

[12]Although most of our specifications do not appear to use the *module*, for practical purposes it is useful to provide to the executable module a reference to itself, e.g. to side-load further resources that might be stored in the same directory as the executable.

```
          if $ResourceAvailabilityFailure(requestedModule)$ then
             $\textsc{Send}(failureReport(req, requestedModule))$
          else
             let $module = loadModule(requestedModule, env)$ in
                if $ModuleLoadabilityFailure(module)$ then
                   $\textsc{Send}(failureReport(req, module))$
                else
                   $\textsc{StartHandler}(module, req, env)$
    where
       $SecurityFailure(s)$ iff $s = 403$,
       $ResourceAvailabilityFailure(m)$ iff $m = 503$,
       $ModuleLoadabilityFailure(module)$ iff $module = 500$,
       $\textsc{StartHandler}(module, req, env) =$
          let $a =$ new $Agent$ in
             $program(a) := \mathrm{EXEC}_{module}$
             $params(a) := (module, req, env)$
             $mode(a) := init$
```

Each EXEC_i machine has the structure

```
$\textsc{Exec}_i =$
   let $module, req, env = params(self)$ in
      $\textsc{Module}_i(module, req, env)$
```

where $\textsc{Module}_i$ is refined as follows.

3.3. Refinement for file transfer

To start with a simple case we illustrate how the machine $\textsc{Module}_i$ can be refined to a machine $\textsc{Module}_{FileTransfer}$ which handles file transfer *modules*, the earliest form of server module. Such a *module* simply buffers the requested *file* in an output buffer if the *file* is present at the location determined by the path from the *root*(*env*) to the *resourceName*(*request*). We use a machine TransferDataFromTo which abstracts from the details of the (not at all atomic, but durative) transfer action of the requested file data to the output.

We leave it open what the scheduler does with the request handler when the latter is DEACTIVATEd once the file transfer *isFinished*, i.e. when it has been detected (here via TransferDataFromTo) that no more data are to be expected for the transfer (this can be refined, for example, to model thread pooling).

```
$\textsc{Module}_{FileTransfer}(module, req, env) =$
   let $file = makePath(root(env), resourceName(req))$ in
      if $mode(\mathbf{self}) = init$ then
         if $UndefinedFile(file)$ then
            $\textsc{Send}(failureReport(req, ErrorCode(UndefinedFile)))$
            $\textsc{Deactivate}(\mathbf{self})$
```

```
        else
          $\textsc{Send}(\mathit{successReport}(\mathit{req}, \mathit{OkResponseCode}))$
          $\mathit{mode}(\mathbf{self}) := \mathit{transferData}$
      if $\mathit{mode}(\mathbf{self}) = \mathit{transferData}$ then
        $\textsc{TransferDataFromTo}(\mathit{file}, \mathit{requestOutput}(\mathit{req}))$
      if $\mathit{isFinished}(\mathit{file})$ then
        $\textsc{Deactivate}(\mathbf{self})$
  where
    $\mathit{ErrorCode}(\mathit{UndefinedFile}) = 404$,
    $\mathit{OkResponseCode} = 200$,
    $\textsc{Deactivate}(\mathbf{self}) = (\mathit{mode}(\mathbf{self}) := \mathit{final})$
```

3.4. Refinement for Common Gateway Interface

A Common Gateway Interface (CGI) [31] *module* allows the request handler to pass requests from a client web browser to an (agent which executes an) external application and to return application output to the web browser. There are two main forms of CGI modules, the historically first one (called CGI) and an optimized one called FastCGI [16]. They differ in the way they introduce agents for external process execution: CGI creates one agent for each request, whereas FastCGI creates one agent and re-uses it for subsequent requests to the same application (though with different parameters).

3.4.1. CGI module

A CGI *module* sends an error message if the *executable* for the requested process is not defined at the indicated location. Otherwise the requested process execution (by an independent newly created agent a, not by the request handler)[13] is triggered for the appropriate *requestVariables* (also called environment variables containing the request data), like Auth(entication)-Type, Query-String, Path-Info, RemoteAddr (of the requesting browser) and Remote-Host (of the browser's machine), etc. (see [31, Sect. 5]) and a positive response is sent to the requesting client. Once the new agent a has been $\textsc{Connect}$ed, the request handler

- transfers data from *requestInput* (coming from the client browser) to the *module*'s *stdin* stream (as input for the execution of the process by a), and
- transfers data from the *module*'s *stdout* stream (generated by a running the *executable*) to *requestOutput* (from where it will be sent to the client browser)

It is usually assumed that the executable $\mathit{program}(a)$ agent a gets equipped with eventually disconnects a (from the request handler **self**) so that the predicate $\mathit{Connected}(a, \mathbf{self})$ becomes false. Then $\textsc{Module}_{CGI}$ terminates wherefore

[13]Therefore each request triggers a fresh instance of the associated external application program to be executed. This is a possible source for exceeding the workload capacity of the machine where the server runs.

the request handler is DEACTIVATEd. Nevertheless the agent a even after having been disconnected may continue the execution of the associated *executable* and may not terminate at all, but such a further execution would be unrelated to the computation of the request handler and from the WEBSERVER's point of view yields a garbage process. Even more, no guarantee is given that $program(a)$ does disconnect a. In these cases the operating system has to close the connection and/or to kill the process by descheduling its executing agent (e.g. via a timeout). The CGI standard [31] leaves this issue open, but it has to be investigated if one wants to provide some behavioral guarantees for the execution of CGI modules.

```
MODULE$_{CGI}(module, req, env)$ =
  let $exe = makePath(root(env), resourceName(req), env)$ in
    if $mode(\mathbf{self}) = init$ then
      if $UndefinedProcess(exe)$ then
        SEND$(failureReport(req, ErrorCode(UndefinedProcess)))$
        DEACTIVATE$(\mathbf{self})$
      else
        let $a$ = new $Agent$ in
          $program(a) := exe(processEnv(env, requestVariables(req)))$
          CONNECT$(a, \mathbf{self})$
          SEND$(req, OkResponseCode)$
          $mode(\mathbf{self}) := transferData$
    if $mode(\mathbf{self}) = transferData$ then
      if $DataAvailable(stdout)$ then
        TRANSFERDATAFROMTO$(stdout, requestOutput(req))$
      if $verb(req) = \texttt{POST}$ and $DataAvailable(requestInput(req))$ then
        TRANSFERDATAFROMTO$(requestInput(req), stdin)$
    if $isDisconnected(a)$ then
      DEACTIVATE$(\mathbf{self})$
  where
    $ErrorCode(UndefinedProcess) = 404$,
    $OkResponseCode = 200$,
    $isDisconnected(a) = \mathbf{not}\ Connected(a, \mathbf{self})$
```

We remark that the server *env*ironment is needed as argument to compute the path information in *makePath*. This is particularly important for the optimized FastCGI version we describe now.

3.4.2. FastCGI module

Concerning the execution of external processes a FastCGI module has the same function as a CGI module. There are two behavioral differences:

- A FastCGI module creates a new agent for the execution of a process only upon the first invocation of the latter by the request handler. An agent a which has been created to process an *exe*cutable is kept alive once this processing *isFinished* so that the agent can become active again for the next invocation of that *exe*cutable — with the new values for the *requestVariables*. To $\textsc{Connect}(a, \mathbf{self})$ now means to link its (local variables for) input (resp. output) locations, denoted below by $in(a)$ (resp. $out(a)$), to corresponding locations of the request handler **self** executing the *module* from where (resp. to which) the data transfer from *requestInput* (resp. to *requestOutput*) is operated. In particular $in(a)$ is used to pass the parameters $requestVariables(req)$ of the process to initialize the *exe*cutable.

- It is assumed that the program $program(a)$ agent a gets equipped with eventually sets a location *EndOfRequest* for the current *req* to false, namely by updating this location during the TransferDataFromCgi action. This makes the request handler terminate.

Thus the CGI structure is refined to the FastCGI module structure as follows:

```
$\textsc{Module}_{FastCGI}(module, req, env) =$
  let $exe = makePath(root(env), resourceName(req), env)$ in
    if $mode(\mathbf{self}) = init$ then
      if $UndefinedProcess(exe)$ then
        $\textsc{Send}(failureReport(req, ErrorCode(UndefinedProcess)))$
        $\textsc{Deactivate}(\mathbf{self})$
      else
        if $\nexists a \in Agent$ with $program(a) = exe(processEnv(env))$ then
          let $a =$ new $Agent$ in
            $program(a) := exe(processEnv(env))$
        else
          skip
        $mode(\mathbf{self}) := connect$
    if $mode(\mathbf{self}) = connect$ then
      let $a = \iota x.(\, x \in Agent \wedge program(a) = exe(processEnv(env))\,)$ in
        $\textsc{Connect}(a, \mathbf{self})$
        $\textsc{Initialize}(a)$
        $mode(\mathbf{self}) := transferData$
    if $mode(\mathbf{self}) = transferData$ then
      let $reqin = requestInput(req),\ reqout = requestOutput(req)$ in
        if $DataAvailable(out(a))$ then
          $\textsc{TransferDataFromCgi}(out(a), reqout, EndOfRequest(request))$
        if $verb(request) = \texttt{POST}$ and $DataAvailable(reqin)$ then
          $\textsc{TransferDataToCgi}(reqin, in(a))$
    if $EndOfRequest(req)$ then
      $\textsc{Deactivate}(\mathbf{self})$
where
  $ErrorCode(UndefinedProcess) = 404,$
  $\textsc{Initialize}(a) =$
    $\textsc{PassParams}(requestVariables(req), in(a))$
    $EndOfRequest(req) := false$
```

Notice that TRANSFERDATATOCGI implies an encapsulation of the content to be transmitted into messages which carry either data or control information; inversely TRANSFERDATAFROMCGI implies a decoding of this encapsulation.

3.5. Refinement for scripting

Scripting modules like ASP, PHP, JSP all provide dynamic web page facilities by allowing the server to run (directly through its request handler) dynamically provided code. We define here a scheme which makes the common structure of such scripting modules explicit.

As for CGI modules, first the server searches the file containing the page to be interpreted, at the place indicated by the *resourceName* of the *request*, starting at the *root* of the server *env*ironment. If the file is defined, the code is executed not by an independent agent as for CGI modules, but directly by the request handler which uses as program the SCRIPTINTERPRETER. For the state management across different server invocations by a series of requests from the same client the uniquely determined *sessionID* (associated to the *request* under the given *env*ironment) and the corresponding session and application (if any) have to be computed. The computation of session (resp. application) comprises that a new session (resp. application) is created in case none is defined yet in the server *env*ironment for the *sessionID* (resp. *applicationName*) of the *request*.[14] Furthermore the syntax conversion of the *script* file from quotation to full script code (denoted here by a machine QUOTETOSCRIPT which is refined below for ASP, PHP and JSP) has to be performed and the corresponding host objects have to be created to be passed as parameters to the SCRIPTINTERPRETER call.

The functions involved to COMPUTESESSION and to COMPUTEAPPLICATION, which allow the server to track state information between different requests of a same client, depend on the *module*, namely *sessionID*, *makeSession* (and therefore *session*), *applicationName*, *makeApplication* (and therefore *application*). Similarly for the functions involved to COMPUTEINTERPRETEROBJECTS. We express this using the **amb** notation as defined in [14].

```
$\text{MODULE}_{script}(module, req, env) =$
  amb $module$ in
    let $script = makePath(root(env), resourceName(req))$,
        $id = sessionID(req, env)$,
        $applName = applicationName(resourceName(req))$ in
      if $mode(\textbf{self}) = init$ then
        if $script = ErrorCode(UndefinedScript)$ then
          $\text{SEND}(failureReport(req, ErrorCode(UndefinedScript)))$
          $\text{DEACTIVATE}(\textbf{self})$
        else
          $\text{COMPUTESESSION}(id, req, env)$
```

[14] Typical refinements of the *sessionID* function also contain specific security policies we necessarily have to abstract from in this high-level description.

```
          $\textsc{ComputeApplication}(applName, req, env)$
          $scriptCode(request) \leftarrow \textsc{QuoteToScript}(script, env)$[15]
          $mode(\mathbf{self}) := compInterprObjs$
      if $mode(\mathbf{self}) = compInterprObjs$ then
        $\textsc{ComputeInterpreterObjects}(req, id, applName)$
        $program(\mathbf{self}) :=$
          $\textsc{ScriptInterpreter}(scriptCode(req), InterpreterObjects))$
where
  $ErrorCode(UndefinedScript) = 404,$
  $\textsc{ComputeSession}(id, req, env) =$
    if $session(id) =$ undef then
      $session(id) := makeSession(req, env, id),$
  $\textsc{ComputeApplication}(applName, req, env) =$
    if $application(applName) =$ undef then
      $application(applName) := makeApplication(req, env, applName),$
  $\textsc{ComputeInterpreterObjects}(req, id, applName) =$
    $reqObj(req) := makeRequestHostObj(req)$
    $responseObj(req) := makeResponseHostObj(req)$
    $sessionObj(req) := makeSessionHostObj(session(id))$
    $applObj(req) := makeApplicationHostObj(application(applName))$
    $serverObj(req) := makeServerHostObj(req, env),$
  $InterpreterObjects =$
    $[reqObj(req), responseObj(req), sessionObj(req), applObj(req),$
      $serverObj(req)]$
```

3.5.1. ASP/PHP/JSP modules

ASP, PHP and JSP modules are instances of the scripting module scheme described above. In fact their $\textsc{Module}_i$ is defined as for the scripting scheme but each with a specific way to produce dynamic web pages, in particular with a specific computation of $\textsc{QuoteToScript}$, as we are going to describe below.

Also the following auxiliary functions and the called $\textsc{ScriptInterpreter}$ are specific (as indicated by an index ASP, PHP, JSP) though not furthermore detailed here:

- The $make \ldots HostObj$ functions are specialized to $make \ldots HostObj_{index}$ functions for each $index \in \{ASP, PHP, JSP\}$.
- $\textsc{ScriptInterpreter}$ becomes $\textsc{ScriptInterpreter}_{index}$ for any $index$ out of ASP, PHP, JSP.

See [19] for explanations how to construct an ASM model of the JavaScript interpreter as described in [3].

A PHP module acts as a filter: it takes input from a file or stream containing text or special PHP instructions and via their $\textsc{ScriptInterpreter}_{PHP}$ interpretation outputs another data stream for display.

[15]The definition of ASMs with return value supporting the notation $l \leftarrow M(x)$ is taken from [15, Def. 4.1.7.].

ASP modules choose the appropriate interpreter for the computed *scriptCode* (so-called *active scripting*). Examples of the type of script code are JavaScript, Visual Basic and Perl.

Thus for ASP the definition of $\textsc{ScriptInterpreter}_{ASP}$ has the following form:

```
$\textsc{ScriptInterpreter}_{ASP}(scriptCode, InterprObjs) =$
  $\textbf{let}\ scriptType = type(scriptCode)\ \textbf{in}$
    $\textsc{ScriptInterpreter}_{scriptType}(scriptCode, InterprObjs)$
```

The value of *scriptCode*(*request*) is defined as the **result** computed by a machine QuoteToScript for a *script* argument. For the original version of PHP, to mention one early example, this machine simply computed a syntax transformation *transform*(*script*). Later versions introduced some optimization. At the first invocation of QuoteToScript(*script*)—i.e. when the syntactical transformation of (the code text recorded at) *script* has not yet been *compiled*—or upon later invocations for a *script* (with code text) changed since the last compilation of *transform*(*script*), due to some code text replacement stored at *script* that is out of the control of the web server, the target bytecode is *compiled* and *timeStamp*ed, using a *compile*r which can be specified using the techniques explained for Java2JVM compilation in [30]. At later invocations of the same *script* the already available *compiled*(*transform*(*script*)) bytecode is taken as *scriptCode* instead of recompiling again. Since the value of the code text located at *script* is not controlled by the web server, the function *timeStamp*(*script*) appears in this model as a monitored function.

```
$scriptCode(request) \leftarrow \textsc{QuoteToScript}(script, env)$
$\textbf{where}$
  $\textsc{QuoteToScript}(script) =$
    $\textbf{let}\ s = transform(script)\ \textbf{in}$
      $\textbf{if}\ compiled(s) = \textbf{undef or}$
        $timeStamp(lastCompiled(script)) \leq timeStamp(script)\ \textbf{then}$
        $compiled(s) := compile(s)$
        $\textbf{result} := compile(s)$
        $timeStamp(lastCompiled(script)) := now$
        $type(compile(s)) := typeOf(script, env)$
      $\textbf{else result} := compiled(s)$
```

For ASP and PHP the QuoteToScript machine describes an optional optimization[16] that cannot be observed from outside. For ASP the machine has the additional update for the *type* of the computed **result** (namely the *scriptCode*) that uses a syntax function *typeOf* which typically yields a directive, e.g.

$$< \%@Language = \text{“}JScript''\% >$$

[16]It is an ASM refinement of the non-optimized original PHP version.

or a default value.

The type of the *scriptCode* depends on the *script* and on the *env*ironment; for example the *env*ironment typically defines a default type for the case that nothing else is specified.

For JSP no syntax translation is required (formally the *transform* function is the identity function) because *scriptCode* is a class file (Servlet which comes with a certain number of fixed interfaces like `doPost()`, `doGet()`, etc.) so that the operations are performed by a JVM. This permits to embed predefined actions (implemented by Java code which can also be included from some predefined file via appropriate JSP directives) into static content. Here the machine QuoteToScript is mandatory because different invocations of the same *scriptCode* can communicate with each other via the values of static class variables.

3.5.2. JSF/ASP.NET modules

It seems that a detailed high-level description of Module_i for the *module*s as offered by the Java Server Faces (JSF [1]) and Active Server Pages (ASP.NET [26]) frameworks can be obtained as a refinement of the ASM defined above for the execution of scripting modules. As mentioned above PHP, ASP and JSP use a character based approach in which the script outputs characters (either explicitly through the Response object or implicitly by using the special notation converted by QuoteToScript). The JSF and ASP.NET frameworks use their virtual-machine based environment (respectively, JVM and CLR) to provide more flexible ways for the ScriptInterpreter to write on the response stream (e.g. in ASP.NET based on the Windows environment) and to define a server-side event and state management model that relieves the programmer from having to explicitly deal with the state of a web page made up by several components. The programming model offered by these environments provides a sort of DOM tree where each node upon being visited is asked for the data to be sent as part of the response so that the programmer has the impression of manipulating objects rather than generating text of a web page. For example, a request handled by the ASP.NET module triggers a complex lifecycle[17] which allows the programmer to manipulate a tree of components each of which has its own state, in part stored inside the web page (in the form of a hidden field) and in part put by the application into the session state. We are currently working on modeling these features as refinements of the ASM model for scripting module execution.

4. Discussion

4.1. Threats to validity

A major concern for each modeling effort is the correctness issue: does the abstract (typically formal) description truly capture the (required intended)

[17] See http://msdn.microsoft.com/en-us/library/ms178472.aspx.

behavior of the described system? If the system to be modeled is given by a mathematically precise description (e.g. code with a well-defined semantics) one way to show the correctness of the model is to accurately state the correctness condition and then prove it (in the usual mathematical sense or by using mechanically supported theorem provers) to hold for the system description and the abstract model. See Sections 4.2 and 5.2 for some concrete correctness conditions of interest for web applications. If the system to be modeled has no such accurate description to be taken as a definition of its behavior — this is the case of the major current web application systems analyzed in this paper — the correctness problem is what in [12] is called ground model problem. Two things can be done to solve this problem: one is a model inspection process, similar to code inspection, where the model elements are compared with the informal descriptions of their intended behavior in the given system (description); the other is to run the model on well defined characteristic test suites and to compare the outcome of these executions with what the system does (if it is executable, as is the case here for web applications). The inspection of our models can be performed — and we invite the reader to check this — since we have tried to produce our ASMs conceptually as close as possible to their informal descriptions in the documentations we refer to. As to an experimental test-based validation what has to be done as future work is to refine our models using provably correct ASM refinements [11] to (e.g. CoreAsm [20]) executable ones one can validate.

4.2. Related work and comparison

Ours is not the first attempt at building a formal model of the infrastructure underpinning web applications. Most previous proposals, though, have focused especially on security properties, rather than on the expository value of such models. Moreover, these models often concentrate on specifying events at certain interfaces, rather than full behavior, and are thus partially different in spirit from our endeavor. Here we will compare our approach to some recent contribution in the field.

A couple of recent examples of this line of approach are [24] and [6]. The former specifies in Coq [9] the kernel of a contemporary web browser (inspired by the Chrome architecture), in terms of the interactions of unverified but sandboxed components (e.g.: renderer, network module, script interpreter). By using theorem proving techniques, and the assumption that the unverified components can interact exclusively with the kernel, the authors can verify that a number of interesting security properties hold (e.g.: that different browser contexts do not interfere with each other, that cookies are only sent back to their corresponding domain, etc.). It is interesting to observe that these properties are similar to those that we can verify by direct inspection of our models — although we have not used sandboxes for our machines (which, however, could be added by judicious use of the **amb** construct).

While [24] only present a model of the browser, their approach based on *shim verification* could be extended to the server, by considering the module dispatching machinery as a kernel, and each server module as an unverified, sandboxed component. Indeed, that would be very similar to the approach we presented in Section 3. It should be noted that while the theorem proving approach provides a stronger guarantee than the mere inspection (assuming both the model and the properties are correctly specified), in our work we have modeled to a certain level of abstraction many components that in [24] are considered pure black-boxes.

The latter work, namely [6], is closer to our own in scope, in that it considers both browser and server, and moreover they aim to describe the full life cycle of realistic web applications, including browser history navigation (which we have modeled, but not detailed in this paper due to space considerations), client-side scripting, and various server modules (e.g.: PHP, ASP, Java Servlets). Their approach abstracts all the features of both browser and server except for a small set of keypoints in the respective computations, the most important of which are conditionals on simple expressions, getting and setting a *sessionID* and *cookies*, and reading or writing data in a *database* (the latter is aimed at typical three-tier architectures). A web application is then modeled as a series of steps, moving from page to page, where each transition entails a number of these operations. The transitions are then expressed in a variant of LTL, and model checking techniques are used to prove properties of specific applications.

Their view of the problem is thus complementary to our own: whereas in this paper we have provided a model of the *infrastructure* of web applications (namely: browser, server, communication), and abstracted the specific applications by assuming instead the availability of interpreters, in [6] the authors abstract away the infrastructure, and focus instead on a specific application (of which a model in their scripting language must be produced). Again, the correctness guarantees that can be obtained by model checking are stronger than those that we can obtain by inspection; yet their technique can prove the correctness of the *model* of a web application, whereas proving that the model faithfully reflect the actual application is left as an open problem.

A similar model checking approach is presented in [23], where a web application is modeled as a pair of automata (one for page navigation, another for internal state transitions), and properties about reachability of pages etc. are proved (the general spirit being close to the earlier [18] and [25]). Again, these models totally abstract from the infrastructure, focusing instead on the internal logic of each specific web application. It should also be noted that the older works all describe a single-browser, single-server, purely sequential scenario (all simplifying assumptions that make model checking feasible), whereas we have focused specifically on modeling the concurrency aspects in multi-browser, multi-server scenarios.

A model of a special-purpose web server, equipped with a fixed set of Java servlets, is presented in [27]. As for the browser model in [24] that we discussed above, the server is specified in Coq, and a few application-specific properties are formally proved by using the Coq interactive theorem prover. Compared to our approach, the work reported in [27] focuses on a single specific web application, rather than on general web server architecture.

5. Conclusions

The goal of our work was to provide abstract models of the infrastructure running modern web applications. We have shown models of the relevant parts of a web browser and a web server, and in doing so we have also highlighted their internal architecture, both in terms of interacting layers (as in our browser model) and in terms of refinement (as in our server model).

5.1. Contribution

The models we have developed are sufficiently detailed for expository purposes, thus fulfilling one of our goals (point (c) in Section 1.1).

As for proving properties (point (b) in Section 1.1), we have indeed provided the foundations for doing so, but further work is needed before being able to reach a meaningful form of verification. The reader should not think that this is simply a matter of adding further detail. On the contrary, unnecessary detail can make any proof exceedingly (and unnecessarily) hard, whereas the lack of sufficient detail can make it impossible to prove the desired property. The hard task is thus to find the *right* level of abstraction. For an illustrative example we can refer to [30] where in terms of rigorous models for Java, the JVM and a compiler Java2JVM the mere mathematically precise formulation of the compiler correctness property stated in Theorem 14.1.1. (p.177-178) needs 10 pages, the entire section 14.1.[18] A formulation in terms of some logic language understood by a theorem prover (e.g. in the language of KIV which has been used for various mechanical verifications of properties of ASMs [28, 29] or in Event-B [5]) is still harder and will be considerably longer, as characteristic for formalizations.

Finally, point (a) from Section 1.1, about our desire of helping other analysts understand, compare and assess the various technologies involved in web applications, cannot be judged based on the models alone. Whether our goal has been met, will be revealed in time by the usage fellow analysts will make of our models. We are however confident that the mere structure we gave our models will be found to be helpful in isolating component technologies, and that we have at least simplified the work of understanding those parts that we have explored in greater detail in this work.

5.2. Future work

The next logical step, beside investing in the further refinement of our models, with a view of making them more complete and precise for those parts we have so far handled informally, concerns proving properties of applications build on top of the infrastructure we have described.

We list here some properties of web applications we suggest to precisely formulate and prove or disprove in terms of abstract web application models.

[18]In comparison the proof occupies 24 pages, the rest of chapter 14.

A first group consists of correctness properties for the crucial session and state management:

- Session management refers to the ability of an application to maintain the status of the interaction with a particular browser. A typical property is that session state is not corrupted by user actions like hitting the *Back*/*Forward* buttons or navigating away from the page and then coming back.
- State management is about the virtual state of the application, which is usually distributed among multiple components on both client and server side, with parts of the state 'embedded' into the local state of several programs, and often also replicated entirely or partially. Typical desirable properties are that at significant time instants replicated parts of the state
  - are consistent, that is they are allowed to be out-of-sync at times and consistence is considered up to appropriate abstraction functions,
  - are equivalent between the client-side and the server-side of the state,
  - can be reconstructed, e.g. when the client can change and its state must be persisted to another client (for example from desktop to mobile).

A second group concerns robustness (e.g. upon loss of a session or client and server state going out-of-sync), security (e.g. confidential information is not compromised) and liveness (e.g. the web application will eventually respond to valid user actions). While security of web applications has been explored in depth, and indeed constitutes a fully developed research area of itself, robustness and liveness have received much less attention, and we believe the foundation we laid with our model can contribute to their analysis.

A third group consists of what we consider to be the most challenging properties which are also of greatest interest to the users, namely application correctness properties. These properties are about the dependence of the intended application-focussed behavior of web applications on the programming and execution infrastructure—on the used browser, web server, net infrastructure (e.g. firewall, router, DNS), connection, plug-ins, etc. Such components are based on their own (not necessarily compatible) standards and therefore may influence the desired application behavior in unexpected ways. This makes their rigorous high-level description mandatory for a precise analysis. An outstanding class of such application-group-specific properties is about application integration where common services are offered on an application-independent basis (e.g. authentication or electronic payment services). We see such investigations as a first step towards defining objective content-based criteria for the reliability of web application software and for building reliable web applications, read: web applications whose properties of interest can be certifiably guaranteed—by theorem proving or model checking or testing or combinations of these activities—to hold under precisely formulated boundary conditions.

References

[1] Java Server Faces. http://www.jcp.org/en/jsr/detail?id=314.

[2] Python. http://www.python.org/.

[3] ECMAScript language specification. Standard ECMA-262, Edition 5.1, June 2011. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[4] HTTP1.1 part 2 message semantics. www.ietf.org, cosulted February 2012.

[5] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge, 2010.

[6] M. Alpuente, D. Ballis, and D. Romero. A rewriting logic approach to the formal specification and verification of web applications. *Science of Computer Programming*, 2013. (in press).

[7] M. Altenhofen, E. Börger, A. Friesen, and J. Lemcke. A high-level specification for virtual providers. *IJBPIM*, 1(4):267–278, 2006.

[8] A. Barros and E. Börger. A compositional framework for service interaction patterns and communication flows. In K.-K. Lau and R. Banach, editors, *Proc. ICFEM 2005*, volume 3785 of *LNCS*, pages 5–35. Springer, 2005.

[9] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[10] E. Boerger, A. Cisternino, and V. Gervasi. Contribution to a rigorous analysis of web applications frameworks. In *Proceedings of ABZ 2012*, LNCS. Springer-Verlag, June 2012.

[11] E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.

[12] E. Börger. Construction and analysis of ground models and their refinements as a foundation for validating computer based systems. *Formal Aspects of Computing*, 19:225–241, 2007.

[13] E. Börger. Approaches to modeling business processes. A critical analysis of BPMN, workflow patterns and YAWL. *JSSM*, pages 1–14, 2011. DOI: 10.1007/s10270-011-0214-z.

[14] E. Börger, A. Cisternino, and V. Gervasi. Ambient Abstract State Machines with applications. *JCSS*, 78(3):939–959, 2012.

[15] E. Börger and R. Stärk. *Abstract state machines: a method for high-level system design and analysis*. Springer, 2003.

[16] M. R. Brown. Fast CGI specification. http://www.fastcgi.com/, April 1996.

[17] V. G. Cerf, Y. Dalal, and C. Sunshine. RFC675: Specification of Internet Transmission Control Program, Dec. 1974.

[18] J. Chen and X. Zhao. Formal models for web navigations with session control and browser cache. In *Formal Methods and Software Engineering: Proceedings of the 6th International Conference on Formal Engineering Methods*, volume 3308 of *LNCS*, pages 46–60. Springer-Verlag, Nov. 2004.

[19] C. Dittamo, V. Gervasi, E. Börger, and A. Cisternino. A formal specification of the semantics of ECMAScript. In *VSTTE-10*, Edinburgh, 2010. Poster session.

[20] R. Farahbod, V. Gervasi, and U. Glaesser. CoreASM: An extensible ASM execution engine. *Fundamenta Informaticae*, 77:71–103, Mar./Apr. 2007.

[21] V. Gervasi. An ASM model of concurrency in a web browser. In *Proceedings of ABZ 2012*, number 7316 in LNCS, pages 79–93. Springer-Verlag, June 2012.

[22] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing, 2003.

[23] K. Homma, S. Izumi, K. Takahashi, and A. Togashi. Modeling, verification and testing of web applications using model checker. *IEICE Transactions on Information and Systems*, E94.D(5):989–999, 2011.

[24] D. Jang, Z. Tatlock, and S. Lerner. Establishing browser security guarantees through formal shim verification. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 1–16, Berkeley, CA, USA, 2012. USENIX Association.

[25] S. Krishnamurthi, R. Findler, P. Graunke, and M. Felleisen. Modeling web interactions and errors. In D. Goldin, S. Smolka, and P. Wegner, editors, *Interactive Computation: The New Paradigm*, pages 255–276. Springer-Verlag, 2006.

[26] Microsoft. ASP.NET. http://www.asp.net.

[27] P. Neron and Q.-H. Nguyen. A formal security model of a smart card web server. In E. Prouff, editor, *Smart Card Research and Advanced Applications*, volume 7079 of *LNCS*, pages 34–49. Springer Berlin Heidelberg, 2011.

[28] G. Schellhorn and W. Ahrendt. The WAM case study: Verifying compiler correctness for Prolog with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume III, pages 165–194. Kluwer Academic Publishers, 1998.

[29] G. Schellhorn, H. Grandy, D. Haneberg, and W. Reif. The Mondex Challenge: Machine Checked Proofs for an Electronic Purse. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006*, volume 4085 of *LNCS*, pages 16–31. Springer, 2006.

[30] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.

[31] W3C. CGI: Common Gateway Interface. http://www.w3.org/CGI/.

[32] World Wide Web Consortium. HTML 5: A vocabulary and associated APIs for HTML and XHTML. http://www.w3.org/TR/html5. W3C Working Draft 19 October 2010.

Contents