Contribution to a Rigorous Analysis of Web Application Frameworks

Egon Börger and Antonio Cisternino and Vincenzo Gervasi

Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
`gervasi,cisterni,boerger@di.unipi.it`

Abstract. We suggest an approach for accurate modeling and analysis of web application frameworks.

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 indipendent cooperating modules with a distributed state, in web applications all offering a unified interface to their user—to the point that the user may have no way to distinguish whether a single application or a set of distributed web applications is used. Also recent non-web systems, like mobile apps, follow the same paradigm allowing the state of an application to be persistent and distributed, no longer tied to the traditional notion of operating system process and memory.

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 (like BPEL, BPMN, workflow patterns, see [9] 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 [23]) scripts to PHP (Personal Home Page) and ASP (Application Server Page) applications and to frameworks such as ASP.NET [19] 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 through the HTTP protocol data 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.

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 birds' 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.

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 a client and a server 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.

1.1 Concrete Goals and Results So Far

As a first step towards the goal outlined above we started to model the client-server architecture of a browser interacting with a web server. In [17] the transport and stream levels of an abstract web browser model are defined. To this we add here models for the main components of the context level layer (Sect. 2) which together with the web server model defined in Sect. 3 allow one to describe one complete round of the Request-Reply pattern [18,8] that characterizes browser/server interactions (see Fig. 1).[1] In Sect. 3.1 a high-level functional Request-Reply web server view is defined which is then detailed (by refinement steps) for the two main approaches to module execution:

- the CGI-approach where the server delegates the execution of an external process to another agent (Sect. 3.3),
- the script-approach where the server itself executes script code (Sect. 3.4).

We explain how one can view existing implementations as instantiations of these models.

We use the ASM (Abstract State Machines) method [12] as modeling framework because it offers all the features listed above which are needed for our endeavor[2] and because various ASM models in the literature contribute specifically to the work undertaken here. For example both the browser and the server model use a third group of basic components, namely ScriptInterpreters for various Script languages, which can be specified by an ASM model adopting the method used in [22] to define an interpreter for Java (and reused in [11,15,16] to rigorously define the semantics of C# and the CLR). These models provide a significant part of the infrastructure web applications typically use. For example applets which run inside a browser, or the Tomcat application server [3], are written in Java. Furthermore, the method developed for modeling Java/JVM can be reused to define a model for the JavaScript interpreter (see [14] for some details) corresponding to the ECMAScript standard ECMA-262 [4], a standard that serves as glue to link various technologies together.

In Sect. 4 we list some verification goals we suggest to pursue on the basis of (appropriately completed) precise abstract models of web application framework components, i.e. to rigorously formulate and check (verify or falsify) properties of interest for the models and/or their implementations.

[1] In the Request-Reply pattern of two-way conversations the requestor (one application) sends a request to the provider (another application) and the provider returns a reply to the requestor.

[2] See [10] for the recent definition of a simple flexible *ambient ASM* concept.

The models we define and their properties we discuss come without any completeness claim and are intended to suggest an approach we consider to be promising for future FM research in a core area of computer technology.

2 Modeling Browser Components

Our browser models focus on those parts of the browser behaviour that are most relevant for the deployement and execution of web applications. The models are developed at four layers. The main components of the *transport layer* (expressing the TCP/IP communication via HTTP) and the *stream layer* (describing how information coming from the network is received and interpreted) are defined in [17]. In this section we add models for characteristic components of the *context layer*, which deals with the user interaction with the document represented by the Document Object Model (DOM). Without loss of generality we omit in this paper the *browser layer* where the behaviour of a web browser seen as an application of the host operating system is described. In practice, most web applications are entirely contained in a single browsing context; in fact an important issue in the development of web standards is how to ensure for security reasons that multiple browsing contexts in the same browser are sufficiently isolated from each other (a security property that we leave to future work).

2.1 Browsing Context

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 [17]), 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 2.2);
- 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 2.3).

We keep the *window* abstract, as its behaviour can be conveniently hidden by keeping the actual rendering abstract 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. We also omit the rather straightforward modeling of the *session history*.

When STARTing a newly created *B*rowsing *C*ontext 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
    $program(a) := \textsc{Renderer}(k)$
    $program(b) := \textsc{EventLoop}(k)$
    $DOM(k) := initialDOM$
```

The RENDERER and EVENTLOOP macros are specified below.

2.2 Renderer

The RENDERER produces the user interface of the current *DOM* in the (implicit) given 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., while 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
    $\textsc{GenerateUi}(DOM(k), k)$
```

2.3 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 macros defined below and in [17] 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.

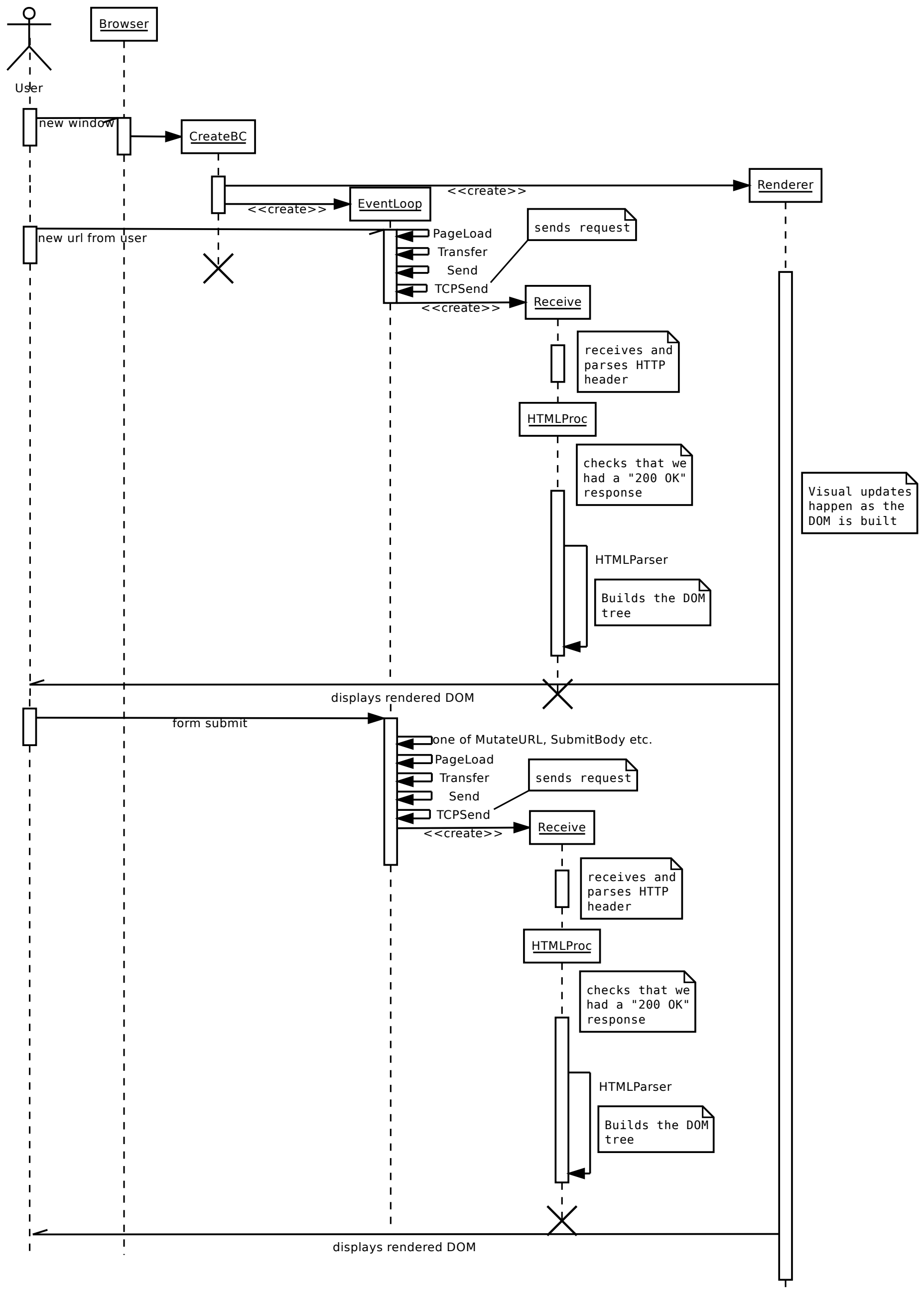

Fig. 1. A diagram depicting the behaviour of our browser model for a user who opens a new window in a browser, manually loads the first page of a web application, interacts locally with a form, and then sends the data back to the server, receiving a new or updated page in response.

An HTML form is introduced by a `<FORM>` element in the page. All the input elements[3] 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 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 indipendently from user behaviour. 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.

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[4].

As for Renderer, the event loop receives a parameter, k, which identifies the particular instance. The macro PageLoad is defined below.

```
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
```

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

[4] 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.

```
  let $f = formElement(e), data = encodeFormData(f),$
    $a = action(f), m = method(f), u = resolveUrl(f, a)$ in
    match $(schema(u), m)$ :
      case (http, GET) : $\textsc{MutateUrl}(u, data, k)$
      case (http, POST) : $\textsc{SubmitBody}(u, data, k)$
      case (ftp, GET) : $\textsc{GetAction}(u, data, k)$
      case (ftp, POST) : $\textsc{GetAction}(u, data, k)$
      case (javascript, GET) : $\textsc{GetAction}(u, data, k)$
      case (javascript, POST) : $\textsc{GetAction}(u, data, k)$
      case (data, GET) : $\textsc{GetAction}(u, data, k)$
      case (data, POST) : $\textsc{PostAction}(u, data, k)$
      case (mailto, GET) : $\textsc{MailHead}(a, data)$
      case (mailto, 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, 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 the transport layer model in [17]. 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 $u' = u \cdot \texttt{?} \cdot data$ 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—which is defined in [17]—and (re-)initializes the browsing context and the HTMLProcessor; the latter is also defined in [17] and will handle the Response:

```
$\textsc{PageLoad}(m, u, data, k) =$
  $\textsc{Transfer}(m, u, data, \text{HTMLProc}, k)$
  $htmlParserMode(k) := Parsing$
  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, releasing resources used in the original page (e.g., freeing images or stopping plug-ins that were running), etc. 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 will be addressed in future work.

3 A High-Level WEBSERVER Model

We define here a companion model to the browser model: a high-level model WEBSERVER (Sect. 3.1) with typical refinements for the underlying handler modules, namely for file transfer (Sect. 3.2), CGI (Sect. 3.3) and scripting modules (Sect. 3.4).

To concentrate on the core issues we abstract in this section from the transmission protocol phase during which the connection between client and server is established and rely upon an abstract SEND mechanism; the missing elements to incorporate this phase can be defined as shown in detail for the browser component models in [17].

3.1 Functional Request-Reply Web Server View

In the high-level view the server appears as dispatcher which to handle a request finds and triggers the code (a 'module') the execution of which will provide a response to the request.[5] Thus a high-level web server model can be formulated as an ASM WEBSERVER which in a reactive manner, upon any *request* in its *requestQueue*, will delegate to a new agent (read: a thread we call *request handler*) to handle the EXECution of the request—if the *request* passes the *Security* check and the *requestedModule* is *Available* in 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 [5, 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

[5] 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.

- 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* 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 *request* and the $header(request)$. 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, namely to Execute the *module*. A loaded *module* is of one of finitely many kinds. For the fundamental CGI and scripting module types we will detail in Sect. 3.3,3.4 what it means to Execute such a module.

To reflect the functional client/server request/reply view StartHandler appears as atomic action of the WebServer which goes together with deleting the *request* from the *requestQueue*. At the transmission protocol level the latter action becomes closing the connection. The atomicity reflects the fact that once a request has been handled, the server is ready to handle the next request.[6]

```
WebServer =
let $request = head(requestQueue)$
if $request \neq$ undef then  // react if there is some $request$
   let $env = env(server, request)$
   let $s = checkSecurity(request, env)$
   if $SecurityFailure(s)$
   then  Send$(failureReport(request, s))$
   else
      let $requestedModule =$
         $findModule(env, resourceName(request), header(request))$
      if $ResourceAvailabilityFailure(requestedModule)$ then
         Send$(failureReport(request, requestedModule))$
      else
         let $module = loadModule(requestedModule, env)$
         if $ModuleLoadabilityFailure(module)$
         then  Send$(failureReport(request, module))$
         else  StartHandler$(module, request, env)$
   Close$(request)$
where
   $SecurityFailure(s)$ iff $s = 403$
   $ResourceAvailabilityFailure(m)$ iff $m = 503$
   $ModuleLoadabilityFailure(module)$ iff $module = 500$
   StartHandler$(module, request, env) =$
      let $a$ = new $(Agent)$ // launch a request handler thread
```

[6] The ASM model supports this view due to the reactive character of ASMs.

```
    $program(a) := \textsc{Exec}(module)(request, env)$
    $mode(a) := init$
  $\textsc{Close}(request) = \textsc{Delete}(request, requestQueue)$
```

3.2 Refinement for File Transfer EXECution

To start with a simple case we illustrate how the machine EXEC(*module*) can be detailed to a machine EXECFILETRANSFER(*module*) which handles file transfer *module*s, 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. The function *requestOutput*(*request*) abstractly represents the appropriate socket through which the response data are sent from the server to the requesting browser.[7]

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.

```
$\textsc{ExecFileTransfer}(module)(request, env) =$
**let** $file = makePath(root(env), resourceName(request))$
**if** $mode(\textbf{self}) = init$ **then**
  **if** $UndefinedFile(file)$ **then**
    $\textsc{Send}(failureReport(request, ErrorCode(UndefinedFile)))$
    $\textsc{Deactivate}(\textbf{self})$ // request handler termination
  **else**
    $\textsc{Send}(successReport(request, OkResponseCode))$
    $mode(\textbf{self}) := transferData$ // Start to transfer the $file$
  **if** $mode(\textbf{self}) = transferData$ **then**
    $\textsc{TransferDataFromTo}(file, requestOutput(request))$
  **if** $isFinished(file)$ **then** $\textsc{Deactivate}(\textbf{self})$
  **where**
    $ErrorCode(UndefinedFile) = 404$
    $OkResponseCode = 200$
    $\textsc{Deactivate}(\textbf{self}) = (mode(\textbf{self}) := final)$
```

3.3 Refinement for Common Gateway Module EXECution

A Common Gateway Interface (CGI) [23] *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

[7] Again this can be made precise as shown in detail for the browser model in [17].

an optimized one called FastCGI [13]. 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).

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)[8] 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 [23, Sect.5]) and a positive response is sent to the requesting client. Once the new agent a has been CONNECTed the request handler

- accepts any further *requestInput* stream (read: data stream coming from the browser) as input for the execution of the process by a, namely via the *stdin* stream of the *module*, and
- transmits any output which (via a's processing the *executable*) becomes available on the *module*'s *stdout* stream to the *requestOutput* stream (from where it will be sent to the requesting browser)—as long as there are data on the *requestInput* resp. on the *stdout* stream.

Thus to CONNECT a to (the agent **self** executing) the CGI *module* a channel is established between the $inputStream(a)$ and the *module*'s *stdin* stream resp. between the $outputStream(a)$ and the *module*'s *stdout* stream[9].

It is usually assumed that the executable $program(a)$ agent a gets equipped with eventually disconnects a (from the request handler **self**) so that the predicate $Connected(a, \mathbf{self})$ becomes false. Then EXEC($module$) terminates wherefor 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 [23] leaves this issue open, but is has to be investigated if one wants to provide some behavioral guarantees for the execution of CGI modules.

[8] 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.

[9] In ASM terms $inputStream(a)$ is a monitored and $outputStream(a)$ an output location for the *executable*, whereas for the *module stdin* is an output location (whereby the request handler **self** passes input to a for the processing of the *executable*) and *stdout* a monitored location (whereby the request handler **self** receives from a output produced through processing the *executable*.)

```
$\textsc{Exec}(module)(request, env) =$
$\textbf{let}\ executable = makePath(root(env), resourceName(request), env)$
$\textbf{if}\ mode(\textbf{self}) = init\ \textbf{then}$
  $\textbf{if}\ UndefinedProcess(executable)\ \textbf{then}$
    $\textsc{Send}(failureReport(request, ErrorCode(UndefinedProcess)))$
    $\textsc{Deactivate}(\textbf{self})$
  $\textbf{else}$
    $\textbf{let}\ a = \textbf{new}\ (Agent)$ // launch a new process instance
    $program(a) := executable(processEnv(env, requestVariables(request)))$
    $\textsc{Connect}(a, \textbf{self})$
    $\textsc{Send}(request, OkResponseCode)$
    $mode(\textbf{self}) := transferData$
$\textbf{if}\ mode(\textbf{self}) = transferData\ \textbf{then}$
  $\textbf{if}\ DataAvailable(stdout)$
    $\textsc{TransferDataFromTo}(stdout, requestOutput(request))$
  $\textbf{if}\ verb(request) = POST\ \textbf{and}\ DataAvailable(requestInput(request))$
    $\textbf{then}\ \textsc{TransferDataFromTo}(requestInput(request), stdin)$
$\textbf{if}\ isDisconnected(a)\ \textbf{then}\ \textsc{Deactivate}(\textbf{self})$
$\textbf{where}$
  $ErrorCode(UndefinedProcess) = 404$
  $OkResponseCode = 200$
  $isDisconnected(a) = \textbf{not}\ Connected(a, \textbf{self})$
```

Remark. 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.

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 *executable* is kept alive once this processing *isFinished* so that the agent can become active again for the next invocation of that *executable*—with the new values for the *requestVariables*. To $\textsc{Connect}(a, \textbf{self})$ now means to link its (local variables for) input resp. output locations, denoted below by $in(a)$, $out(a)$, 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 $requestVariables(request)$ of the process to initialize the *executable*.
- It is assumed that the program $program(a)$ agent a gets equipped with eventually sets a location *EndOfRequest* for the current *request* 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:

```
$\textsc{Exec}(\mathit{module})(\mathit{request}, \mathit{env}) =$
$\textbf{let } \mathit{executable} = \mathit{makePath}(\mathit{root}(\mathit{env}), \mathit{resourceName}(\mathit{request}), \mathit{env})$
$\textbf{if } \mathit{mode}(\textbf{self}) = \mathit{init} \textbf{ then}$
  $\textbf{if } \mathit{UndefinedProcess}(\mathit{executable}) \textbf{ then}$
    $\textsc{Send}(\mathit{failureReport}(\mathit{request}, \mathit{ErrorCode}(\mathit{UndefinedProcess})))$
    $\textsc{Deactivate}(\textbf{self})$
  $\textbf{else}$
    $\textbf{if thereisno } a \in \mathit{Agent} \textbf{ with}$
        $\mathit{program}(a) = \mathit{executable}(\mathit{processEnv}(\mathit{env}))$
      $\textbf{then}$
        $\textbf{let } a = \textbf{new } (\mathit{Agent})$
        $\mathit{program}(a) := \mathit{executable}(\mathit{processEnv}(\mathit{env}))$
    $\mathit{mode}(\textbf{self}) := \mathit{connect}$
$\textbf{if } \mathit{mode}(\textbf{self}) = \mathit{connect} \textbf{ then}$
  $\textbf{let } a = \iota x(x \in \mathit{Agent} \textbf{ and}$
    $\mathit{program}(a) = \mathit{executable}(\mathit{processEnv}(\mathit{env})))$
  $\textsc{Connect}(a, \textbf{self})$
  $\textsc{Initialize}(\mathit{program}(a))$
  $\mathit{mode}(\textbf{self}) := \mathit{transferData}$
$\textbf{if } \mathit{mode}(\textbf{self}) = \mathit{transferData} \textbf{ then}$
  $\textbf{let } \mathit{reqin} = \mathit{requestInput}(\mathit{request}),\ \mathit{reqout} = \mathit{requestOutput}(\mathit{request})$
  $\textbf{if } \mathit{DataAvailable}(\mathit{out}(a))$
    $\textsc{TransferDataFromCgi}(\mathit{out}(a), \mathit{reqout}, \mathit{EndOfRequest}(\mathit{request}))$
  $\textbf{if } \mathit{verb}(\mathit{request}) = \mathit{POST} \textbf{ and } \mathit{DataAvailable}(\mathit{reqin}) \textbf{ then}$
    $\textsc{TransferDataToCgi}(\mathit{reqin}, \mathit{in}(a))$
$\textbf{if } \mathit{EndOfRequest}(\mathit{request}) \textbf{ then } \textsc{Deactivate}(\textbf{self})$
$\textbf{where}$
  $\mathit{ErrorCode}(\mathit{UndefinedProcess}) = 404$
  $\textsc{Initialize}(\mathit{program}(a)) =$
    $\textsc{PassParams}(\mathit{requestVariables}(\mathit{request}), \mathit{in}(a))$
    $\mathit{EndOfRequest}(\mathit{request}) := \mathit{false}$
```

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

3.4 Refinement for Scripting Module Execution

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 file for the to be executed code is searched 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 accross 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 and 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*.[10] 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 [10].

```
$\textsc{Exec}(module)(request, env) =$
$\textbf{let}\ script = makePath(root(env), resourceName(request))$
$\textbf{amb}\ module\ \textbf{in}$  // NB: use of $module$ sensitive functions
  $\textbf{if}\ mode(\textbf{self}) = init\ \textbf{then}$
    $\textbf{if}\ script = ErrorCode(UndefinedScript)\ \textbf{then}$
      $\textsc{Send}(failureReport(request, ErrorCode(UndefinedScript)))$
      $\textsc{Deactivate}(\textbf{self})$
    $\textbf{else}$
      $\textbf{let}\ id = sessionID(request, env)$
        $\textsc{ComputeSession}(id, request, env)$
      $\textbf{let}\ applName = applicationName(resourceName(request))$
        $\textsc{ComputeApplication}(applName, request, env)$
      $scriptCode(request) \leftarrow \textsc{QuoteToScript}(script, env)$[11]
      $mode(\textbf{self}) := compInterprObjs$
  $\textbf{if}\ mode(\textbf{self}) = compInterprObjs\ \textbf{then}$
    $\textsc{ComputeInterpreterObjects}(request, id, applName)$
    $program(\textbf{self}) :=$
      $\textsc{ScriptInterpreter}(scriptCode(request), InterpreterObjects))$
  $\textbf{where}$
    $ErrorCode(UndefinedScript) = 404$
    $\textsc{ComputeSession}(id, request, env) =$
```

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

[11] The definition of ASMs with return value supporting the notation $l \leftarrow M(x)$ is taken from [12, Def.4.1.7.].

```
  if $session(id) = \textbf{undef}$ then
    $session(id) := makeSession(request, env, id)$
$\textsc{ComputeApplication}(applName, request, env) =$
  if $application(applName) = \textbf{undef}$ then
    $application(applName) := makeApplication(request, env, applName)$
$\textsc{ComputeInterpreterObjects}(request, id, applName) =$
  $reqObj(request) := makeRequestHostObj(request)$
  $responseObj(request) := makeResponseHostObj(request)$
  $sessionObj(request) := makeSessionHostObj(session(id))$
  $applObj(request) := makeApplicationHostObj(application(applName))$
  $serverObj(request) := makeServerHostObj(request, env)$
$InterpreterObjects =$
  $[reqObj(request), responseObj(request),$
    $sessionObj(request), applObj(request), serverObj(request)]$
```

ASP/PHP/JSP Module ASP, PHP and JSP modules are instances of the scripting module scheme described above. In fact their $\textsc{Exec}(module)$ is defined as for the scripting scheme but each with a specific way to produce dynamic webpages, 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 [14] for explanations how to construct an ASM model of the JavaScript interpreter as described in [4].

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.

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:

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

The value of $scriptCode(request)$ is defined as the **result** computed by a machine $\textsc{QuoteToScript}$ 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 werver, the target bytecode is *compiled* and *timeStamp*ed, using a *compile*r which can be specified using the techniques explained for Java2JVM compilation in [22]. 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.

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

For ASP and PHP the QUOTETOSCRIPT machine describes an optional optimization[12] 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.

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

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

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

scriptCode can communicate with each other via the values of static class variables.

JSF/ASP.NET Modules It seems that a detailed high-level description of EXEC(*module*) for the *module*s as offered by the Java Server Faces (JSF [1]) and Active Server Pages (ASP.NET [19]) 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 (JVM resp. 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[13] 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 The Challenge of Accurate Analysis

Once sufficiently rich rigorous abstract web application models have been defined they can be used to accurately define properties of interest one would like to prove or falsify for the models via proofs or counterexamples which are preserved by correct refinements for existing implementations. This is by no means an easy task. For an illustrative example we can refer to [22] 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.[14] 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 [20,21] or in Event-B [6]) is still harder and will be considerably longer, as characteristic for formalizations.

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

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

[14] 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 and liveness.

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.

Acknowledgement. This paper is published in the two Proceedings volumes of the joint iFM2012 and ABZ2012 Conference held in Pisa (Springer LNCS 7321 and 7316).

References

1. Java Server Faces. http://www.jcp.org/en/jsr/detail?id=314.
2. Python. http://www.python.org/.
3. Tomcat. http://tomcat.apache.org/.

4. ECMAScript language specification. Standard ECMA-262, Edition 5.1, June 2011. http://www.ecma-international.org/publications/standards/Ecma-262.htm.
5. HTTP1.1 part 2 message semantics. www.ietf.org, cosulted February 2012.
6. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge, 2010.
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. 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.
10. E. Börger, A. Cisternino, and V. Gervasi. Ambient Abstract State Machines with applications. *JCSS*, 78(3):939–959, 2012.
11. E. Börger, G. Fruja, V. Gervasi, and R. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 336(2–3):235–284, 2005.
12. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
13. M. R. Brown. Fast CGI specification. http://www.fastcgi.com/, April 1996.
14. C. Dittamo, V. Gervasi, E. Börger, and A. Cisternino. A formal specification of the semantics of ECMAScript. In *VSTTE-10*, Edinburgh, 2010. Poster session.
15. N. G. Fruja. Towards proving type safety of .NET CIL. *SCP*, 72(3):176–219, 2008.
16. N. G. Fruja and E. Börger. Modeling the .NET CLR Exception Handling Mechanism for a Mathematical Analysis. *Journal of Object Technology*, 5(3):5–34, 2006.
17. V. Gervasi. An ASM model of concurrency in a web browser. In *Proceedings ABZ2012*, LNCS. Springer, 2012.
18. G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing, 2003.
19. Microsoft. ASP.NET. http://www.asp.net.
20. 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. 1998.
21. 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.
22. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.
23. W3C. CGI: Common Gateway Interface. http://www.w3.org/CGI/.