Appears in: *Proceedings of the Eighth IEEE International Symposium on Web Site Evolution (WSE 2006)*, Philadelphia, PA, pp. 103-110, September 23-24, 2006.

Modeling Request Routing in Web Applications

Minmin Han and Christine Hofmeister
Lehigh University
mih9@lehigh.edu, hofmeister@cse.lehigh.edu

Abstract

For web applications, determining how requests from a web page are routed through server components can be time-consuming and error-prone due to the complex set of rules and mechanisms used in a platform such as J2EE. We define request routing to be the possible sequences of server-side components that handle requests. Many maintenance tasks require the developer to understand the request routing, so this complexity increases maintenance costs. However, viewing this problem at the architecture level provides some insight. The request routing in these web applications is an example of a pipeline architectural pattern: each request is processed by a sequence of components that form a pipeline. Communication between pipeline stages is event-based, which increases flexibility but obscures the pipeline structure because communication is indirect.

Our approach for improving the maintainability of J2EE web applications is to provide a model that exposes this architectural information. We use Z to formally specify request routing models and analysis operations that can be performed on them, then provide tools to extract request routing information from an application's source code, create the request routing model, and analyze it automatically. We have applied this approach to a number of existing applications up to 34K LOC, showing improvement via typical maintenance scenarios. Since this particular combination of patterns is not unique to web applications, a model such as our request routing model could provide similar benefits for these systems.

1. Introduction

For web applications, determining how requests from a web page are routed through server components can be a time-consuming and error-prone task due to the complex set of rules and mechanisms used in a platform such as J2EE.

However, viewing this problem at the architecture level provides some insight. The basic structure of this request handling follows a pipeline (pipes and filters) architectural pattern: each request is processed by a sequence of components that form a pipeline [4][25].

Communication between pipeline stages (between server components) is event-based. A request has an associated URI value, and this URI is the "event id": each component begins executing when it receives a request whose URI is one for which it is "registered," and finishes by sending out the request with another URI value. The platform uses a lookup table to determine which URI values each component is registered to receive.

The purpose of using event-based communication is to gain flexibility. Rather than having one component send the request directly to another, it announces the request as an "event." To change the pipeline structure we simply change the events a component receives by updating the lookup table.

This combination of event-based communication and a pipeline architectural pattern is by no means unique to web applications. One example is in an embedded real-time patient monitoring system (HealthyVision) [13], where sensor data is acquired then processed by a series of components in order to generate alarms and display the data as waveforms. Here the event-based communication is supported by a custom platform, which passes the data between stages using "Global Data Objects."

In all of these event-based pipelines it is difficult to see the pipeline structure because communication between pipeline stages is indirect. In J2EE. A request originates from a web page (a JSP, Java Server Page), and the request has an associated URI value. The lookup table is part of the application's deployment descriptor, a file named web.xml. This portion of web.xml uses the URI to route the request to the appropriate component, which will be a servlet component (or a filter component, which is similar to a servlet).

The servlet may do some processing of the request, perhaps using another Java class. Then the servlet will set the target URI and forward the request. Again web.xml will map the URI to a servlet, and the platform will invoke the servlet. This is repeated until the servlet forwards the

Appears in: *Proceedings of the Eighth IEEE International Symposium on Web Site Evolution (WSE 2006)*, Philadelphia, PA, pp. 103-110, September 23-24, 2006.

request to a JSP, which causes the JSP to be displayed in the user's browser.

We define the request route be the sequence of components that handle a request. This sequence starts with the web page from which the request is generated and ends with the result web page that is displayed in the user's browser. Between these two components there may be several components implicitly invoked by the platform. Any Java class explicitly invoked by another when handling a request is not considered part of the request route. Because each component determines the next in the sequence, and can make different choices depending on data in the request, the particular route taken by a request may not be known until runtime.

Then we define an application's request routing to be the union of all possible request routes for requests in that application. Although the route taken by a particular request cannot necessarily be determined until runtime, the possible routes of all types of requests must be known statistically or at least by deployment time.

Common maintenance tasks for web applications require that the developer understand the request routing. For example, a change in a web page means that the code processing requests from that page may need to be changed. The flexibility brought by event-based pipeline architecture pattern results in complexity when determining the request routing. Our goal is to improve the maintainability of web applications by improving the developer's ability to understand and modify the request routing.

Our first step was to improve the implementation of the event-based pipeline pattern by applying conventions for separating the request routing code from the rest of the application. This work is described in [11] . The implementation conventions address the problem of the many variations on how request routing can be encoded. Since the solution employs separation of concerns, the request routing code is located in just a few modules rather than being dispersed among a number of JSPs, servlets, filters, and web.xml.

However, we found that implementation-level improvements alone were not adequate. Although request routing code is separated it is still located in several modules, and the problems caused by event-based communication remain. In order to trace the request routing, the developer must still understand the different component types, how the platform interprets and uses the deployment descriptor (web.xml), and a number of J2EE navigation-related functions.

This paper presents our solution to the remaining problems: use a formal model to represent an application's request routing, and provide tool support for the model. The model makes the request routing easier to understand, reduces the chance of errors due to coding mistakes or misunderstanding of the J2EE platform, and enables us to provide operations for analyzing properties of the request routing.

Furthermore, when certain implementation conventions are used the automatic extraction of the request routing model becomes straightforward. It is now common to use a web framework such as Struts [19], WAF [23], etc. on top of J2EE. Because these frameworks impose implementation conventions on parts of the request routing code, we are able to automatically extract request routing models from applications that use these frameworks.

In Section 2 we introduce an example and describe a maintenance scenario for it. Section 3 presents a formal specification for request routing models, and analysis operations for these models. Section 4 describes tools to extract the model from source code and analyze it. In Section 5 we discuss our evaluation of this solution, and in Section 6 present related work. Section 7 concludes the paper.

2. Example

To illustrate the complexity of request routing in web applications, we describe a very small subset of the request routing of the Petstore application from Sun's J2EE tutorial [22]. This application was written as an exemplar application, so it reflects good practice.

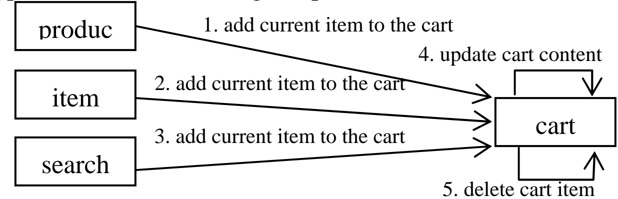


Figure 1 Navigation Map for part of Petstore

Figure 1 shows the navigation map for this portion of Petstore, which is part of the checkout process. A navigation map describes the possible sequences of pages accessible to a user. A user can add a product to their shopping cart by clicking the "Add To Cart" button on the "product", "item" or "search" web page. This brings them to the "cart" web page. From here a user can click on the "Update" or "Remove Item" button to update or delete a cart item.

Such a navigation map, if provided, gives the developer some information about request routing, namely the beginning and end of the request routes. But a developer must currently examine the source code to determine the request routing. Figure 2 shows the parts of the code that implement request routing for this part of Petstore, given as pseudo code.

There are five request routes in this example. Although they start from different web pages (product, item, search and cart), the later parts of the routes are the same.

All of them start from a form or a link in the web page (n1) with URI of cart.do. The deployment descriptor

(web.xml) indicates that this URI is first routed to EncodingFilter (n2). This class does not change the URI so it forwards the request with the same URI: cart.do (n6). Then web.xml indicates the request is routed to SignOnFilter (n3) and similarly it does not change the URI and forwards the request with same URI (n6). According to web.xml (n4) the request is next forwarded to MainServlet. MainServlet invokes cartAction to process requests with URI cart.do (n20-n21). cartAction checks the action attribute in the request and processes the request accordingly (n8-n19). Finally, the request is forwarded to TemplateServlet to generate the cart response page (n5, n23).

```
    product.jsp/item.jsp/search.jsp/cart.jsp
n1 set action attribute and send request with
URI "cart.do"

                    web.xml
n2 all URIs go to EncodingFilter
n3 all URIs go to SignOnFilter
n4 URIs ending with .do go to MainServlet
n5 URIs ending with .screen go to
                              TemplateServlet

          EncodingFilter/SignOnFilter
n6   forward the request

                   CartAction
n7 public final class CartAction extends …{
n8  public Event perform(…) throws … {
n9    String action;
n10   action=(String)req.getParameter("action");
n11   if (action.equals("purchase")) {
n12     … // process purchase
n13   }
n14   else if (action.equals("remove")) {
n15     … // process remove
n16   }
n17   else if (action.equals("update")) {
n18     … // process update
n19   }
}}

                  MainServlet
n20 for requests where URI is cart.do
n21   invoke CartAction perform()
n22 forward request with URI "cart.screen"

                TemplateServlet
n23 for all requests ends with .screen,
        generate the corresponding composite
        web page
```

Figure 2 Request Routing Code (Pseudo Code)

To illustrate why a developer needs to understand an application's request routing, we present a simple maintenance scenario. A programmer decides to separate cartAction into three classes to handle purchase requests, update requests and remove requests separately. ("Purchase" is the original application's terminology for adding something to the cart, so we keep this.) Then three different URIs will be used to indicate the different purposes: purchase (cartP.do), update (cartU.do) and remove (cartR.do).

This maintenance task has three steps and a number of sub-steps:

1. Identify request routes that can reach CartAction and check each component that precedes CartAction along these routes:
   a. Check MainServlet because it invokes cartAction. MainServlet does not change the incoming requests' URI: cart.do (n20-n22).
   b. Check web.xml and find that requests are handled by SignOnFilter (n3).
   c. Check SignOnFilter: it does not do any processing with requests of URI cart.do.
   d. Check SignOnFilter: it does not change the requests' URI (n6).
   e. Check web.xml and find that requests are handled by EncodingFilter (n2).
   f. Check Encoding Filter: it does not do any processing with requests of URI cart.do.
   g. Check Encoding Filter: it does not change the request's URI (n6).
   h. Check web.xml and find no other components that handle the requests. So the requests must come from web pages.
   i. Check every web page and find product.jsp, item.jsp, search.jsp and cart.jsp generate requests with URI cart.do (n1).
2. Modify the related code:
   a. Split code in cartAction into three separate classes for handling purchase (cartPchs), update (cartUpt) and remove requests (cartRmv).
   b. Change code in MainServlet so three incoming URIs (cartP.do, cartU.do and cartR.do) will be mapped to three classes but the outgoing URI is cart.screen.

```
                  MainServlet
for requests where URI is cartP.do
   invoke cartPchs perform()
forward request with URI "cart.screen"
for requests where URI is cartU.do
   invoke cartUpt perform()
forward request with URI "cart.screen"
for requests where URI is cartR.do
   invoke cartRmv perform()
forward request with URI "cart.screen"
```

   c. Change request URI in product.jsp, item.jsp, search.jsp to cartP.do and change the request URI in cart.jsp to cartU.do and cartR.do.

```
      product.jsp/item.jsp/search.jsp
send request with URI "cartP.do"

                  cart.jsp
send request with URI "cartU.do"
send request with URI "cartR.do"
```

Appears in: *Proceedings of the Eighth IEEE International Symposium on Web Site Evolution (WSE 2006)*, Philadelphia, PA, pp. 103-110, September 23-24, 2006.

3. Verify the changes:
   a. Follow the request routes to locate possible errors.
   b. Compile and test run.

From the steps we can see that actual code modification in step 2 is a modest part of the overall effort. The identifying and verifying steps take time, but are also tedious and error-prone.

Our approach for improving this situation is to model the request routing of an application at a higher level, namely as a pipeline. The model abstracts away the details of the event-based communication used in this pipeline. Then the developer can use the request routing model to reduce the effort of maintenance tasks.

3. Request Routing Models

To formally specify request routing models we use the Z notation [18]. We define first a set of state schemas that together represent the request routing. Operation schemas are used for updating and analyzing the state (most of which is the request routing model).

We used Z since it is a well-established formal language for specifying software systems (see [9] for example). Obviously there are other possible choices of formal specification languages such as Alloy [1]. Currently we are satisfied with the expressive power of Z language and the tool support for writing Z specifications (we used Z-Eves [15]) and executing Z specifications (we used Jaza [14]).

To populate the state schemas for a request routing model, some state information is extracted directly from the source code and some is computed. These state schemas and the operation schemas are described in the remainder of this section. Because of the space limitations, here we simply describe the schemas and operations. The Z specifications of these schemas and operations can be seen in [10].

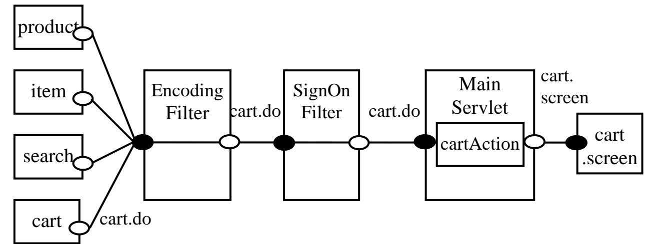
3.1 State Schemas

The data for some of the state schemas is extracted directly from the source code. We define nodes to represent those components from which a request originates and those that are implicitly invoked during request handling. In J2EE web applications a node can be a JSP web page, a composite web page, a servlet class, or a filter class. Web.xml maps URIs to servlet or filter classes but does not handle any request itself, so web.xml is not a node.

Because of the way the mapping in web.xml is specified, nodes in our model must be ordered so that JSPs come first, filters come next and are strictly ordered, servlets follow filters, and composite pages follow servlets. These nodes receive and/or forward requests, each with a target URI. We use ports to describe the different target URIs a node can receive and/or forward with.

We use the following type of request routing diagram to visualize these models (Figure 3).

What happens within a node is that a request enters a node through an entry port, is processed, then is forwarded out of the node through an exit port. We model these pairs of one entry port and one exit port as inNodeConnections to represent how a request is routed inside a node. It is very possible that an entry or exit port appears in several in-node connections. In our section 2 example there are a number of in-node connections, e.g. the pair (entry port with URI "cart.do" attached to SignOnFilter, exit port with URI "cart.do" attached to SignOnFilter).



Symbol	Model	Notes
	Nodes	Nodes are positioned so that request routes proceed from left to right.
URI	Ports (portType = exit)	portURI is noted next to the oval. attachedNode is shown by the position of the port.
URI	Ports (portType = entry)	Same as exit port.
	Function (inside a node)	Function is in the middle of one or more inNodeConns that it associates with. Function return values are noted next to the exit port of the inNodeConns.
	inNode Conns	Connects ports of the same node.
	Between Nodes Conns	Connects ports of different nodes. Since these ports must have the same URI, we show the URI only once.

Figure 3 Visualization Form of the Model

There may be a processing function associated with an in-node connection. The typical usage of a function is: a request comes in through an entry port, then the function is called to process the request. If the return result matches the expected result, the request will be forwarded through the exit port. Since a function usually has multiple possible return values, it is typically associated with multiple in-node connections sharing the same entry port.

If there is no function associated with an in-node connection, it does not necessarily mean that requests traveling through this pair of entry and exit ports are not processed in this node. It could simply mean that the processing is not separated into a function.

To complete the state schemas, we use several derived state schemas. We define betweenNodesConns to represent the connections between nodes. We also define

connectedPorts to be any pair of directly connected or indirectly connected ports, and connectedNodes to be any pair of nodes containing connected ports. Of course, the schemas for derived state are not strictly necessary as part of the model since they can always be computed, but they simplify the definition of subsequent operations and we consider them part of the model.

Section 4.1 describes various ways of extracting the data for these state schemas, however the basic approach is as follows. First we create a node for each JSP in the implementation, and an exit port for each unique URI located in a form or link in the JSP. These nodes have nodeOrder 0.

Next we use web.xml to find the filters and servlets. For each mapping between a URI and a receiving component in web.xml we create a node for the component (if it does not already exist). The node is given a nodeOrder: filters are ordered sequentially starting at 1, and all servlets have nodeOrder one bigger than the maximum filter nodeOrder.

The code for filters and servlets is the source for entry ports, exit ports, in-node connections and functions. A filter/servlet accepts requests with certain URIs, processes the requests and forwards them out. Each incoming URI is an entry port attached to the node for this component. Web.xml also yields information about entry ports, since it says which URIs are mapped to which component. The outgoing URI is an exit port attached to the same node. Also these two ports comprise an in-node connection. If the processing is packaged into a function, this function maps to the function in the model.

When any of the servlet classes forward the request to a web page or build a composite web page, we create a node for each of these response pages, giving it a nodeOrder one bigger than the maximum servlet nodeOrder. These nodes have only one entry port and no exit ports.

Given the request routing model and how it must be extracted, it should be clear why determining the request routing is a time-consuming and error-prone activity. Ports are not present in the code; they are implied by the use of URI values. Exit ports for JSPs are implied by the URI values used in their forms or links. Entry ports for filters, servlets, and composite pages are implied by mappings in web.xml, but their exit ports are implied by in-node connections determined by reading the filter or servlet code. Connections between nodes depend upon matching up exit ports and entry ports with matching URIs, and upon the sequencing rules implicit in web.xml.

3.2 Operations

The following table lists all operations in our request routing model to date. These operations are used to maintain/update states and analyze the model.

Maintaining Extracted State	
AddNode/RemoveNode	Add/Remove a node.
AddPort/RemovePort	Add/Remove a port.
AddInNodeConn/ RemoveInNodeConn	Add/Remove one in-node connection.
AddFunction/ RemoveFunction	Add/Remove a function.
AssociateFunction WithPorts/Release FunctionWithPorts	Associate/Release a function with a pair of connected ports.
Init	Reset all states to empty.
Maintaining Derived State	
UpdateRoutes	Calculate the between-nodes connections.
UpdateConnections	Calculate the connected nodes and ports.
Analyzing the Model	
GenerateNavigationMa p	Calculate the navigation map.
CalculatePreceding	Take a port as a parameter. Return all preceding ports in the request routing. (transitive)
CalculateSucceeding	Take a port as a parameter. Return all succeeding ports in the request routing. (transitive)
FindUnusedNodes	Return the set of nodes not involved in any connections.
FindUnusedPorts	Return the set of ports not involved in any connections.

For example, if we use CalculatePreceding for the maintenance scenario, we use the entry port of MainServlet with URI cart.do as the input port. CalculatePreceding returns exit ports attached to product.jsp, item.jsp, search.jsp and cart.jsp with URI “cart.do”, an entry port attached to EncodingFilter with URI “cart.do”, an exit port attached to EncodingFilter with URI “cart.do”, an entry port attached to SignOnFilter with URI “cart.do”,. an exit port attached to SignOnFilter with URI “cart.do”. This corresponds exactly to the set of ports in the maintenance scenario that we want to find in step one.

One important contribution of our model is that it helps the developer detect implicit dependencies resulting from the use of event-based communication for the request routing. Earlier we identified some of these: for example, there is an implicit dependency between a web page that generates a request and the server components that process it [12]. With a request routing model, the CalculatePreceding and CalculateSucceeding operations can be used to locate components that may have these implicit dependencies.

4. Tool Support

Appears in: *Proceedings of the Eighth IEEE International Symposium on Web Site Evolution (WSE 2006)*, Philadelphia, PA, pp. 103-110, September 23-24, 2006.

In Section 3 we explained how the elements in the request routing model correspond to the source code. Here we address the practical aspects of extracting a request routing model from the source code.

We have developed the tool NavRExtractor to automatically extract the model from the source code. Request routing information is extracted from:

- Web pages: nodes for web pages and the exit ports attached to those nodes.
- Deployment descriptor and server components (servlet and filter classes): nodes for server components, ports attached to those nodes, in node connections and functions.

NavRExtractor extracts request routing information from web pages by searching for strings wrapped by "< >" with keywords "href" or "form". More keywords can be added for the use of tags.

NavRExtractor parses the deployment descriptor file searching for <SERVLET> and <SERVLET-MAPPING> nodes.

The extractor tool cannot extract request routing information from servlets or filters when they do not follow any implementation conventions. But a developer has several options for implementing server side components. Frameworks like Struts [19], WAF [23], JSF [20] are widely used for J2EE web application development. These frameworks impose implementation conventions on request routing and many of them separate parts of the request routing code into XML-based configuration files. We have also investigated using conventions to segregate request routing code into a small number of modules [11].

If the developer does not follow any implementation conventions, the model can be created by manually creating the needed state information. While this is a time-consuming task, it could be done incrementally as maintenance tasks involving the request routing arise. Capturing in a request routing model the results of tracing request routes means that those routes need not be traced again in future maintenance tasks.

Finally, we use a general purpose Z animator Jaza [14] to execute our Z specification. We have NavRExtractor output the model into a text file, then use that file as the input to Jaza. Thus Jaza builds the request routing model and allows the developer to execute operations to view and analyze the model.

5. Evaluation

To evaluate the effectiveness of our approach for modeling request routing, we would ideally like the answer the questions: Does our approach reduce the effort required to understand and/or modify the request routing? By how much?

To date we have focused on three exemplar web applications: Duke's Bank [21] and Petstore from Sun's tutorial [22] and Virtual Shopping Mall from Oracle's tutorial [16]. The three web applications differ in many respects. Duke's Bank is a small on-line banking application (7 KLOC) that uses the front controller design pattern. Virtual Shopping Mall is a medium-sized on-line retail store application (19 KLOC) that also has a front controller, but it uses the Struts framework. Pet Store is another medium-sized on-line store application (34 KLOC) that uses the WAF framework.

We have successfully extracted and modeled the request routing of the three applications described above. We did this for the original applications and for refactored versions that use Struts or our implementation conventions for request routing code. Of course these represent a tiny fraction of existing web applications, but this question will have to be answered by choosing representative applications rather than by looking at all. The three we used were written at different times by two different vendors. More importantly, the three represent different approaches for request routing code: two use web application frameworks (Struts and WAF), and the other uses no framework. Next we plan to try the approach on applications that use other existing frameworks.

As for the approach's applicability to web development platforms other than J2EE, we have not yet gone any further than a cursory examination of a few of these. What we have found is that these other platforms use a similar approach to J2EE with respect to request routing.

To answer the question of how much effort is reduced when a request routing model is used, we employ maintenance scenarios such as the one in Section 2. Without a request routing model there were fourteen steps: nine for identifying the affected code, three for modifying the code, and two for verifying the changes. However, of the nine steps for identifying affected code, seven are for following the request route. By using a request routing model, these seven steps are reduced to one, that of identifying the components along the request route. The table below summarizes the eight steps that are needed for the maintenance scenario when the developer uses a request routing model, compared to the steps needed when there is no such model (fourteen steps).

Note also that the new step 7 is equivalent to old step 3.a, but involves different activities. To accomplish this step without a request routing model, the developer must manually trace through three different request routes. With a new model, although the model must be updated, the analysis can be done automatically.

	Steps needed for the maintenance scenario, when using a request routing model.	Old Steps
1	Identify the components in the request route: Option 1: look at a request routing diagram. Option 2: use NavRAnimator or JAZA to execute CalculatePreceding.	1.a-b, 1.d, 1.e, 1.g-i
2	Check SignOnFilter for code that	1.c

Appears in: *Proceedings of the Eighth IEEE International Symposium on Web Site Evolution (WSE 2006)*, Philadelphia, PA, pp. 103-110, September 23-24, 2006.

	processes requests with URI cart.do.	
3	Check Encodingfilter for code that processes requests with URI cart.do.	1.f
4	Split code in cartAction into three classes.	2.a
5	Modify request routing in MainServlet.	2.b
6	Modify request routing in product.jsp, item.jsp, search.jsp and cart.jsp.	2.c
7	Update the request routing model (manually or using NavRExtractor). Execute analysis operations to check for errors (e.g. FindUnusedNodes, FindUnusedPorts).	3.a
8	Compile and test run.	3.b

We have begun the same kind of analysis with 30 similar maintenance scenarios. This analysis gives us qualitative information about the number of steps that can be eliminated, although it does not reveal differences in the actual effort required.

At this point we have only anecdotal evidence about the effort required to understand request routing with and without a request routing model. For the refactored version of Duke's Bank (refactored to use our implementation conventions [11]), it took 5 hours to understand the request routing. For refactored versions of the other two applications, it took 10 hours each. With a request routing diagram it took only about 1/3 as long. These numbers represent the experiences of two students with some familiarity with J2EE.

6. Related Work

There are two groups of related work. First is research on modeling event-based systems, also called implicit invocation or publisher/subscriber systems. The other is research related to request routing in web applications.

Research on event-based systems mainly focuses on providing implementation conventions and/or frameworks for developing event-based systems. (See for example [2].)

There has also been research on using formal methods to provide rigorous specification and verification for implicit invocation systems. Dingel et. al developed an implicit invocation system verification model based on rely/guarantee reasoning [6]. These researchers also developed second a verification methodology based on linear time temporal logic and compositional reasoning [6]. Garlan et. al. use a state-machine-based model to capture run-time event management and dispatch policy in implicit invocation systems [8]. In other research, Garlan et. al. use Z to specify implicit invocation systems, using their formal model primarily to compare different implicit invocation systems [9].

The main focus of these models is to verify whether the states of cooperating components are synchronized. Examples are: ensuring that senders catch up with the listeners sooner or later, or the cache for temporarily event storage will not overflow. However, these problems are not an issue for web applications, where synchronization is managed by the web server. A developer can assume that each event (each request) will be dispatched to the appropriate listeners within very small amount of time. The framework also ensures that components sharing state are invoked sequentially rather than concurrently.

Our use of a formal model is quite different. Our model superimposes a pipeline pattern and provides a set of analyzing operations. The nature of the event-based pipeline system makes the identification of the event-passing route (request routing in web application domain) difficult. So our model is used to describe and verify properties of such routes instead of verifying correctness properties of the communication.

In the web engineering research area, researchers have developed notations to be used when designing web applications, in order to define how application behavior maps to implementation components. WWM [5] uses UML use case diagrams and UCM scenario diagrams to describe how a request starts from a web form and goes through server objects. Dialog Flow Notation [3] describes dialog control in web applications that uses nested dialogs. Though its focus is not request routing, it describes how requests travel through server components and the actions performed along the path. Both of these notations cover some of the things needed to model request routing, namely the web page nodes, server component nodes and the connections between the nodes. But the notations do not allow us to model URI values or processing functions, nor connecting these to form a request route. The models are descriptive rather than formal, they do not explicitly model the pipeline pattern, and there is no support for automatic analysis of the model or automatic extraction of a model from an existing web application.

In Section 3.2 we noted how the analysis operations for a request routing model can be used to help locate implicit dependencies between a web page and the server components that process the request generated from that web page. This is one of the four types of dependency relationships in web applications defined by Ricca et al. [17]. In addition to defining the four types, they discuss how slicing can be used to locate these dependencies. However, they provide no support for modeling or analyzing the request routing and no tool support to automatically extract this dependency relationship.

7. Conclusion

In this paper we present a formal specification of request routing models for web applications, analysis operations for these models, and tools to support the specification and analysis of request routing models.

A request routing model makes explicit the pipeline architectural pattern used in request routing. The flexibility of the web application is not sacrificed, since the application retains the event-based communication used between pipeline stages. But with a model describing the sequence of server components that process each request, the developer can quickly locate relevant code when performing maintenance tasks, either by examining the model or by executing analysis operations on the model. Additional operations analyze the request routing for correctness (e.g. FindUnusedNodes).

Our request routing models and analysis operations are specified in Z, a well-established formal language that provides sufficient expressive power for describing request routing models and adequate existing tools for writing and animating (executing) specifications [18][15][14].

We have developed the NavRExtractor tool, which extracts the request routing information from source code. NavRExtractor does this for web applications that follow either the implementation conventions we describe in [11], and or those imposed by the use of the Struts framework. The NavRExtractor output is fed into the Jaza animator, which executes our Z specification. Thus Jaza builds the request routing model and allows the developer to execute operations to view and analyze the model.

To date we have used these tools to extract and model the request routing of three exemplar J2EE applications: Duke's Bank [21] and Petstore from Sun's tutorial [22] and Virtual Shopping Mall from Oracle's tutorial [16]. The three web applications differ in size, structure, and in use of underlying framework.

We have validated the utility of request routing models using maintenance scenarios. For the maintenance scenario described in the paper, using a request routing model reduces the number of required maintenance steps from fourteen down to eight. Most other maintenance scenarios show similar reductions, mainly because any maintenance task that involves server-side components requires the developer to trace parts of the request routing.

Our experiences also indicate that a developer can understand the request routing much more quickly when using a request routing model rather than simply by examining the implementation (about 1/3 as long with the model).

8. References

[1] The Alloy Analyzer. http://alloy.mit.edu

[2] Barrett, D. J., Clarke, L. A., Tarr, P. L, and Wise, A. E. A framework for event-based software integration. ACM Transactions on Software Engineering and Methodology, 5(4): 378-421, Oct 1996.

[3] Book, M. and Gruhn, V.: Modeling Web-Based Dialog Flows for Automatic Dialog Control. In *19th IEEE International Conference on Automated Software Engineering*, 2004. 100-109

[4] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons. 1996.

[5] Daewkasi, C., and Rivepiboon, W.: WWM: a Practical Methodology for Web Application Modeling. In *26th International Computer Software and Applications Conference (COMPSAC)*, 2002. 603-608

[6] Dingel, J., Garlan, D., Jha, S., and Notkin, D.: Towards a formal treatment of implicit invocation. Formal Aspects of Computing, 10, (1998). 193-213.

[7] Dingel, J., Garlan, D., Jha, S., and Notkin, D. Reasoning about implicit invocation. In *Proceedings of the 6th ACM SIGSOFT international Symposium on Foundations of Software Engineering* (Lake Buena Vista, Florida, United States, Nov, 1998). SIGSOFT '98/FSE-6. 209-221.

[8] Garlan, D. and Khersonsky, S.: Model Checking Implicit-Invocation Systems. In *Proceedings of the 10th Intl. Workshop on Software Specification and Design*, Nov, 2000. IEEE Computer Society, Washington, DC, 23.

[9] Garlan, D., and Notkin, D. Formalizing design spaces: Implicit invocation mechanisms. In VDM'91: Formal Software Development Methods, 31-44, Noordwijkerhout, The Netherlands, Oct 1991. Springer-Verlag, LNCS 551.

[10] Han, M. The Request Routing Model. http://www.cse.lehigh.edu/~crh/crhlinks.html

[11] Han, M. and Hofmeister, C.: Separation of Navigation Routing Code in J2EE Web Applications. In *5th International Conference on Web Engineering*, 2005, 221-231.

[12] Han, M., Hofmeister, C. and Nord, R.: Reconstructing Software Architecture for J2EE Web Applications. In *10th Working Conference on Reverse Engineering*, Victoria, B.C., Canada, Nov 2003. 67-77

[13] Hofmeister, C., Nord, R., and Soni, D. Applied Software Architecture. Addison-Wesley 2000.

[14] The Jaza Animator. http://www.cs.waikato.ac.nz/~marku/jaza/

[15] ORA Canada. Z-Eves. http://www.ora.on.ca/z-eves/

[16] Oracle J2EE sample code. Virtual Shopping Mall. http://www.oracle.com/technology/sample_code/tech/java/j2ee/vsm13/index.html

[17] Ricca, F., and Tonelle, P. Web Application Slicing. In *Proceedings of the IEEE International Conference on Software Maintenance*, (2001)

[18] Spivey, J.M. The Z Notation: a reference manual. http://spivey.oriel.ox.ac.uk/~mike/zrm/

[19] Struts. http://struts.apache.org/

[20] Sun Developer Network. JavaServer Faces. http://java.sun.com/j2ee/javaserverfaces/

[21] Sun Java Center. The Duke's Bank Application. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Ebank.html

[22] Sun Java Center. Java Petstore 1.3.01 http://java.sun.com/developer/releases/petstore/petstore1_3_01.html

[23] Sun Java Center. Web Application Framework Tutorial. http://docs.sun.com/app/docs/doc/819-0727

[24] Tonella, P. and Ricca, F. Web Application Slicing in Presence of Dynamic Code Generation. Automated Software Engineering, 12, 259-288, 2005.

[25] Vermeulen, A., Beged-Dov, G., and Thompson, P. The Pipeline Design Pattern. OOPSLA '95 Workshop on Design

Appears in: *Proceedings of the Eighth IEEE International Symposium on Web Site Evolution (WSE 2006)*, Philadelphia, PA, pp. 103-110, September 23-24, 2006.

Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems, Oct 1995.