Process-aware Web Programming with Jolie

Fabrizio Montesi
IT University of Copenhagen
Rued Langgaards Vej 7
Copenhagen, Denmark
fmontesi@itu.dk

ABSTRACT

We present a programming framework, based upon the Jolie language, for the native modelling of process-aware web information systems. Our major contribution is to offer a unifying approach for the programming of distributed architectures based on HTTP that support typical features of the process-oriented paradigm, such as structured communication flows and multiparty sessions.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*World Wide Web (WWW)*

General Terms

Design, Languages

Keywords

Business Processes, Web Applications

1. INTRODUCTION

A Process-Aware Information System is an information system based upon the execution of business processes. Since processes can assume complex structures [4] many formal methods [13, 5], tools [14], and standards [1] have been developed to support their writing, verification, and execution.

In web applications, processes are usually implemented server-side on top of *sessions*. Sessions track incoming messages related to the same conversation, and support the latter with a local memory state that lives until it is terminated.

The major frameworks for developing web applications (e.g., PHP, Ruby on Rails, Java EE, . . .) do not support the explicit programming of structured processes. As a workaround, programmers simulate the latter by exploiting the session-local memory state. For example, a process where a user has to authenticate through a `login` operation before accessing another operation, say `postNews` (for posting a news on a website), would be implemented by defining the two operations separately. The sequentiality between them can then be enforced by using a bookkeeping variable in the session state. Although widely used, this approach is error-prone, since the bookkeeping code for processes with complex structures can be poorly readable.

The limitation described above can be tackled by adopting a multi-layered architecture. For example, we may stratify an application by employing a web server technology for serving content to web browsers; a web scripting framework (e.g., PHP) for computing dynamic response content; a process-oriented language (e.g., WS-BPEL [11]) for modelling application processes; and, finally, mediation technologies such as proxies and ESB [6] for integrating the web application within larger systems. Such an architecture would offer a good *separation of concerns*. However, the resulting system would be highly heterogeneous, requiring a specific know-how for handling each part. Thus, it could be hard to maintain and prone to breakage in case of modifications.

This paper reports on an attempt to simplify the programming of process-aware web information systems. Specifically, we have built a programming framework that captures the different components of such systems and their integration using a homogeneous set of concepts. Our results are based on Jolie [10, 2], a general-purpose service-oriented programming language that can handle both the modelling of processes (without bookkeeping code) and their integration within larger distributed systems. We have extended Jolie to support the HTTP protocol natively, and we show how this extension can be used to program web-oriented systems.

2. WEB PROGRAMMING WITH JOLIE

We refer to [2] for a presentation of Jolie. As a rough reference, each Jolie program defines a service in a distributed environment and is formed by two parts. The first part defines the input and output ports for communicating with the rest of the system. Each port specifies a location (e.g., a URL), a data protocol (e.g., SOAP), and, finally, an interface that defines the data types of the operations of a service (as in WSDL [15]). The second part is enclosed in a `main` procedure and defines the behaviour of the service. The latter is a composition of input and output actions, which refer to operations (as in WS-BPEL [11]).

In this work, we have built a new data protocol for Jolie, named `http`. Hereby, we discuss how to use `http` to cover some useful web application patterns. The interested reader may refer to [8] for a more comprehensive presentation.

2.1 Modelling Web Servers

We first address the programming of a web server that provides static content (e.g., a web user interface) to clients.

The main challenge in modelling a web server is that, in service-oriented technologies such as Jolie and WS-BPEL [11], a service interface is a statically defined set of operations. Differently, web servers expose a set of resources (from, e.g., a part of a filesystem) that can change dynamically (e.g., when a file is added). To deal with this problem, `http` features the `default` configuration parameter. `default` points to an operation, which will be used as a fallback when a client sends a request message for an operation that has not been statically declared. For instance, we can use `default` to implement the following: whenever we receive a request for the default operation, we try to find a file in the local filesystem whose filename is as the operation originally requested by the client. We have used this mechanism to implement Leonardo [3], a web server in pure Jolie:

Listing 1: Leonardo Web Server (excerpt)

```
/* ... */
interface MyInterface {
RequestResponse:
  d( DefaultOperationHttpRequest )
   ( undefined ) }
inputPort HTTPInput {
Location: "socket://localhost:80/"
Protocol: http
  { .default = "d" /* ... */ }
Interfaces: MyInterface }
main {
  d( req )( resp ) { /* ... */
    readFile@File( req.operation )( resp )
  } }
```

Above, the input port `HTTPInput` uses the `http` protocol, where we have set the `default` parameter to operation `d`. Therefore, messages for undeclared operations will be managed by the implementation of operation `d`. The latter reads a file with the same name as the originally requested operation (`req.operation`) and returns its content (`resp`).

2.2 Multiparty Sessions

We show an implementation sketch of the `login`/`postNews` process-aware scenario mentioned in the Introduction. Our `http` protocol accepts invocations with different formats (e.g., AJAX calls in JSON or HTML form encodings), so we will leave the code for the user interface unspecified.

A critical aspect of our implementation is the modelling of *sessions*. Assume that, e.g., two users are logged in the service at the same time and, therefore, are supported by two separate process instances in our Jolie service. When a message for operation `postNews` arrives, how can we know if it is from the first user or the second? We address this issue by using correlation sets [11], which specify special variables whose values can relate incoming messages to running service processes. We combine the correlation set mechanism offered by Jolie [9] with HTTP cookies, which are typically employed for storing session identifiers in web browsers. Our example will use two correlation sets consisting of one variable each, respectively `userSid` and `modSid`. The first will identify calls from the user, whereas the second will identify messages from the moderator. Having separate correlation sets is a fundamental aspect of *multiparty sessions*, such as the one in this example, for security reasons: since we will not make `modSid` known to the user, she will be unable to (maliciously) impersonate the moderator.

We can finally show the code for our service:

Listing 2: A moderated news service

```
/* Types, Interfaces, Output ports, ... */
inputPort MyInput { // ...
Protocol: http {
  .cookies.userSid = "userSid";
  .cookies.modSid = "modSid";
  .default = "d"
} }
cset { userSid: postNews.userSid }
cset { modSid:
  approve.modSid reject.modSid }
main {
  [ d( req )( resp ) ] { /* ... */ }
  [ login( cred )( r ) {
    check@Authenticator( cred )( ok );
    if ( ok ) {
      csets.userSid = new;
      r.userSid = csets.userSid
    } else { throw( AuthFailed ) }
  } ] {
    postNews( news ); csets.modSid = new;
    { log@Logger( cred.username )
     | notify@Moderator( csets.modSid ) };
    [ approve() ] { /* ... */ }
    [ reject() ]  { /* ... */ }
  } }
```

Above, we have reused the web server pattern from § 2.1 to provide the resources for the (omitted) web user interface to web browser clients. We combine that pattern with a process that starts with an input on `login`, using an input choice. When `login` is invoked, a new process is started which immediately checks if the user's credentials are valid through an external `Authenticator` service. If they are valid (condition `ok`), then we instantiate the correlation variable `csets.userSid` (`csets` is a special keyword for accessing correlation variables in the behaviour) to a *fresh* value (`new`); otherwise, we throw a fault `AuthFailed`, therefore notifying the client and interrupting the process. We return `csets.userSid` to the user through the response message for `login`. Due to our configuration for `http`, spefically `.cookies.userSid = "userSid"`, `r.userSid` will be encoded as a cookie in our HTTP response to the client. When the user's client will call our service on operation `postNews`, our cookie configuration for `userSid` will convert the cookie in the HTTP request to a subnode `userSid` inside the request message, which we will use for correlating with the correct process instance. After `postNews` is invoked, we instantiate the correlation variable `modSid`. Then, we use the parallel compositor `|` to notify a `Logger` service and a `Moderator` service in parallel. The latter is informed of the value for `modSid`, which we will expect as a cookie (per our `http` configuration) in incoming messages from the moderator's user interface. The cookie will be used to correlate with our process, which is finally waiting for a decision between the `approve` operation and the `reject` operation.

2.3 Multi-layering

In § 2.2, we have used a single service to handle both content serving and process execution. We can separate these two aspects by using *aggregation* [12], a composition mechanism where an input port delegates the implementation of some operations to an external service.

With aggregation, we can split the web server code and the process code from our news service in two separate services. The service below handles the news moderation process:

```
/* Types, Interfaces, Output ports, ... */
inputPort MyInput {
Location: "socket://localhost:8001/"
Protocol: soap
Interfaces: MyIface }
cset { userSid: postNews.userSid }
cset { modSid:
  approve.modSid reject.modSid }
main { login( cred )( r ) { /* ... */ } }
```

The code above is taken from Listing 2. We have changed input port `MyInput` to be deployed on a different location using the `soap` protocol and we have removed the code for handling content serving. The rest of the service code is unmodified (the body of input `login` is the same). Content serving is moved to the following separate service:

```
/* Types, Interfaces, Output ports, ... */
outputPort News { /* ... */ }
inputPort WebInput {
Location: "socket://localhost:80/"
Protocol: http {
  .cookies.userSid = "userSid";
  .cookies.modSid = "modSid";
  .default = "d"
} Interfaces: ContentIface
Aggregates: News }
main { d( req )( resp ) { /* ... */ } }
```

Above, input port `WebInput` takes care of receiving HTTP messages from web clients and aggregates the news service through output port `News` (which points to input port `MyInput` of the news service). When a message is received, Jolie will check whether its operation is defined in the interface of `News`. If so, then the message will be transparently forwarded (converting it with the `soap` protocol) to the news service and the subsequent response from the latter will be given back to the client. Otherwise, it will be interpreted as an invocation to be handled by the default operation `d`.

Our `http` extension can be combined with aggregation also for modelling systems where a single web application interacts with multiple services. For example, we may build a web server that supports both the user interface for users and for news moderators. Then, some web clients running the user interfaces would need to access the processes inside service `News` while others would need to access service `Moderator`. We can allow web clients to access both through the same web server by adding `Moderator` to the list of aggregated output ports inside the web server:

```
Aggregates: News, Moderator
```

Remarkably, since all client invocations for the aggregated services pass through the web server, this methodology respects the Same Origin Policy by design.

3. CONCLUSIONS

We have presented a framework for the programming of process-aware web applications. Through examples, we have shown how our solution subsumes useful web design patterns and how it captures complex scenarios involving, e.g., multiparty sessions and the Same Origin Policy. Our `http` extension is open source and is included in the standard distribution of Jolie. An important aspect of `http` is that the programmer does not need to deal with the differences between the data formats employed in HTTP messages (e.g., form encodings, querystrings, JSON, ...), since they will all be translated to Jolie data structures. This also means that the techniques developed for the verification and execution of Jolie programs (e.g., [9]) can be transparently applied to the programs written in our framework.

Our solution has been evaluated in the development of industrial products and at italianaSoftware [7], a software development company that uses Jolie as reference programming language. For instance, the company's website [7] and Web Catalogue, a proprietary E-Commerce platform with a codebase of more than 400 services, use the framework and the programming patterns presented in this paper.

4. REFERENCES

[1] BPMN 2.0. `http://www.omg.org/spec/BPMN/2.0/`.

[2] Jolie website. `http://www.jolie-lang.org/`.

[3] Leonardo. `http://www.sf.net/projects/leonardo/`.

[4] Workflow Patterns. `http://www.workflowpatterns.com/`.

[5] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.

[6] D. A. Chappell. *Enterprise Service Bus - Theory in practice*. O'Reilly, 2004.

[7] italianaSoftware s.r.l. italianaSoftware. `http://www.italianasoftware.com/`.

[8] F. Montesi. Programming Process-Aware Web Information Systems with Jolie. Draft paper, 2012. `http://www.itu.dk/~fabr/papers/jolie_web/`.

[9] F. Montesi and M. Carbone. Programming services with correlation sets. In *ICSOC*, pages 125–141, 2011.

[10] F. Montesi, C. Guidi, and G. Zavattaro. Composing Services with JOLIE. In *Proceedings of ECOWS 2007*, pages 13–22, 2007.

[11] OASIS. WS-BPEL Version 2.0. `http://docs.oasis-open.org/wsbpel/`.

[12] M. D. Preda, M. Gabbrielli, C. Guidi, J. Mauro, and F. Montesi. Interface-based service composition with aggregation. In *ESOCC*, pages 48–63, 2012.

[13] W. M. P. van der Aalst. Verification of workflow nets. In *ICATPN*, pages 407–426, 1997.

[14] W. M. P. van der Aalst and A. H. M. ter Hofstede. Yawl: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005.

[15] W3C. Web Services Description Language. `http://www.w3.org/TR/wsdl`.