Programming Process-Aware Web Information Systems with Jolie

Draft of 23 November 2012

Fabrizio Montesi

IT University of 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, i.e., web-oriented software based upon the execution of business processes. 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. Using examples, we show how our solution successfully captures the programming of all the necessary components for supporting the operation of a web user interface, from content serving to process coordination.

1 Introduction

A Process-Aware Information System (PAIS) is an information system based upon the execution of business processes. PAIS's are largely adopted in many application scenarios [14], from inter-process communication to automated business integration. Since processes can assume many different structures (see [9] for a systematic account), many formal methods [37,16,25,12], tools [38,28,19,?,15], and standards [30,40,1] have been developed to provide languages for their definition, verification, and execution.

In the last two decades, web applications have joined the trend of process-awareness. Web processes are usually implemented server-side on top of *sessions*, which track incoming messages related to the same conversation. Sessions are supported with a local memory state, which lives through different client invocations until the session 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 usually simulate the latter by exploiting the session-local memory state. For example, consider a process where a user has to authenticate through a `login` operation before accessing another operation, say `createNews` (for posting a news on a website). This would be implemented by defining the `login` and the `createNews` operations separately. The code for `login` would update a bookkeeping variable in the session state and the implementation for `createNews` would check that variable when it is invoked by

the user. Although widely used, this approach is error-prone: since processes can assume quite complex structures, it may be hard to simulate them through bookkeeping variables. Consequently, the produced code can be poorly readable and hard to maintain.

The limitations described above can be avoided by adopting a multi-tier architecture. For example, we may stratify an application by employing a web server technology (e.g., Apache Tomcat) for serving content to web browsers; a web scripting framework (e.g., PHP) for programmable request processing; a process-oriented language (e.g., WS-BPEL [30]) for modelling the application processes; and, finally, mediation technologies such as proxies and ESB [13] 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 would be hard to maintain and potentially prone to breakage in case of modifications.

The aim of this paper is to simplify the programming of process-aware web information systems. We present a programming framework that successfully captures the different components of such systems (web servers, processes, ...) and their integration using a homogeneous set of concepts. We build our results on top of Jolie [28,6] (§ 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 [29].

1.1 Contributions

We report our major contributions.

Web processes. We integrate the Jolie language with the HTTP protocol, enabling processes written in Jolie to send and receive HTTP messages (§ 3). The integration is *seamless*, meaning that the processes defined in Jolie remain abstract from the underlying HTTP mechanisms and data formats: Jolie's data structures are transparently transformed to HTTP messages and vice versa (§ 3.1). Transformations can be configured through separate parameters (§ 3.2).

Web servers as processes. We enable Jolie processes to model the programming of web servers, for serving content to clients (§ 4.1). Hence, in our framework web servers are not ad-hoc third-party programs anymore, but are instead modelled using the same language that we use for defining processes.

Multiparty sessions. A session in a web server is typically dedicated to a single web client. The latter can refer to it through a session identifier. In process-oriented languages, instead, sessions can be between more than two participants. Different participants are identified by using different session identifiers, or correlation sets [27]. We combine correlation sets with HTTP to introduce multiparty sessions in web applications (§ 4.2).

Multi-tiering. Aggregation [26,33] is a primitive of Jolie that allows for the composition of separate services in an information system, as in ESB [13]. Our HTTP

implementation supports aggregation transparently. We show how to use this combination to obtain multi-tiered architectures.

2 An Overview of Jolie

Jolie [28] is a general-purpose service-oriented programming language, released as an open source project [6] and formally specified as a process calculus [27,26,16]. In this section we describe some of its features relevant for our discussion. The interested reader may consult [29] for a more comprehensive presentation.

A Jolie program defines a service and is a composition of two parts: *behaviour* and *deployment*. A behaviour defines the implementation of the operations offered by a service; it consists of communication and computation instructions, composed into a structured process (a workflow) using constructs such as sequences, parallels, and internal/external choices. Behaviours rely on *communication ports* to perform communications, which are to be correctly defined in the deployment part. The latter can also make use of architectural primitives for handling the structure of an information system. Formally, a Jolie program is structured as:

$$D \quad \texttt{main}\ \{\ B\ \}$$

where D represents the deployment part and B the behavioral part.

Behaviours. Fig. 1 reports the (selected) syntax for service behaviours, which offers primitives for performing communications, computation, and their composition in processes. We briefly comment the syntax.

$$
\begin{array}{lll}
B ::= \eta & & \textit{(input)} \\
\quad \mid\ \overline{\eta} & & \textit{(output)} \\
\quad \mid\ \texttt{[}\ \eta_1\ \texttt{]}\ \texttt{\{}\ B_1\ \texttt{\}} \ldots \texttt{[}\ \eta_n\ \texttt{]}\ \texttt{\{}\ B_n\ \texttt{\}} & & \textit{(input choice)} \\
\quad \mid\ \texttt{if(}\ e\ \texttt{)}\ B_1\ \texttt{else}\ B_2 & & \textit{(cond)} \\
\quad \mid\ \texttt{while(}\ e\ \texttt{)}\ B & & \textit{(while)} \\
\quad \mid\ B\ \texttt{;}\ B' & & \textit{(seq)} \\
\quad \mid\ B\ \texttt{|}\ B' & & \textit{(par)} \\
\quad \mid\ \texttt{throw(f)} & & \textit{(throw)} \\
\quad \mid\ \texttt{x =}\ e & & \textit{(assign)} \\
\quad \mid\ \texttt{x -> y} & & \textit{(alias)} \\
\quad \mid\ \texttt{nullProcess} & & \textit{(inact)} \\
\quad \eta ::= \texttt{o(x)} & & \textit{(one-way)} \\
\quad\quad \mid\ \texttt{o(x)(}e\texttt{)\{}\ B\ \texttt{\}} & & \textit{(request-response)} \\
\quad \overline{\eta} ::= \texttt{o@OP(}e\texttt{)} & & \textit{(notification)} \\
\quad\quad \mid\ \texttt{o@OP(}e\texttt{)(y)} & & \textit{(solicit-response)}
\end{array}
$$

Fig. 1. Jolie behavioural syntax (selected rules)

Rules *(input)*, *(output)*, and *(input choice)* implement communications. An input η can either be a one-way or a request-response, following the WSDL standard [41]. Statement *(one-way)* receives a message for operation `o` and stores its content in variable `x`. *(request-response)* receives a message for operation `o` in variable `x`, executes behaviour B (called the *body* of the request-response input), and then sends the value of the evaluation of expression e to the invoker. *(notification)* and *(solicit-response)* dually implement the outputs towards the input primitives. *(notification)* sends a message containing the value of the evaluation of expression e. *(solicit-response)* sends a message with the evaluation of e and then waits for a response from the invoked service, storing it afterwards in variable `y`. In the output statements, `OP` is a reference to an *output port*, to be defined in the deployment part. *(input choice)* is similar to the `pick` construct in WS-BPEL: when a message for an input η_i can be received, all other branches are deactivated and η_i and, afterwards, its respective branch behaviour B_i are executed.

Rules *(cond)* and *(while)* implement respectively the standard conditional and iteration constructs. *(seq)* models sequential execution and reads as: execute B, wait for its termination, and then run B'. *(par)*, instead, runs B and B' in parallel. *(throw)* throws a fault signal `f`, interrupting execution (we omit the syntax for handling faults). If a fault signal is thrown from inside a request-response body, the invoker is automatically notified of the fault.

Rule *(assign)* stores the result of the evaluation of expression e in variable `x`. *(alias)* makes variable `x` an alias for variable `y`, i.e., accessing `x` will be equivalent to accessing `y`. Term `nullProcess` denotes the empty behaviour.

Jolie supports structured data manipulation natively. In Jolie's memory model the program state is a tree (with array nodes, see [27]), and every variable, say `x`, can be a *path* to a node of the memory tree. Paths are constructed through the dot operator. E.g., the following sequence of assignments

```
person.name = "John"; person.age = 42
```

would lead to a state containing a tree with root label `person`. For clarity, a corresponding XML representation would be:

```
<person> <name>John</name>
         <age>42</age>     </person>
```

Deployments. We introduce now the syntax for deployments (for a complete model, see [26,33]). The basic deployment primitives are *input ports*, denoted by IP, and *output ports*, denoted by OP, which respectively support input and output communications with other services. Input and output ports are dual concepts and their syntaxes are quite similar. Ports are based upon the three fundamental concepts of *location*, *protocol* and *interface*. Their (selected) syntax follows:

IP ::= **inputPort** P $\quad$ OP ::= **outputPort** P

$Port$ ::= id {
Location: Loc
Protocol: $Proto$
Interfaces: $\mathsf{iface}_1, \ldots, \mathsf{iface}_n$
}

where Loc is a URI (Uniform Resource Identifier), defining the location of the port; $Proto$ is an identifier referring to the data protocol to use in the port, specifying how input/output messages through the port should be decoded/encoded; the iface_i's are references to the interfaces accessible through the port.

Jolie supports different locations and protocols. For instance, a valid Loc for accepting TCP/IP connections on TCP port 8000 would be `"socket://localhost:8000"`. Other supported locations comprehend Unix sockets, Bluetooth communication channels, and local memory. Some supported instances of $Proto$ are `sodep` [8], `soap` [39], and `xmlrpc` [10].

The interfaces declared in a communication port define the operations accessible through it. Each interface defines a set of operations, pairing each with (i) the operation type (one-way or request-response) and (ii) the types of its carried messages. For example, the following interface

```
interface SumIface
  { RequestResponse: sum(SumT)(int) }
```

defines an interface `SumIface` with a request-response operation `sum`, which expects input messages of type `SumT` and returns messages of type `int` (integers). Data types for messages follow a tree-like structure. E.g., we could define `SumT` as follows:

```
type SumT:void { .x:int .y:int }
```

We can read the code above as: a message of type `SumT` is a tree with an empty root node (`void`) and two subnodes, `x` and `y`, that have both type `int`.

A simple example. We can now give an example of how to combine behaviour and deployment definitions, by showing a simple service defined in Jolie. The code follows:

```
type SumT:void { .x:int .y:int }
interface SumIface
  { RequestResponse: sum(SumT)(int) }

inputPort MyInput {
Location: "socket://localhost:8000"
Protocol: soap
Interfaces: SumIface
}

main {
```

```
  sum( req )( resp ) {
    resp = req.x + req.y
  }
}
```

Above, input port `MyInput` deploys the interface `SumIface` (and thus the `sum` operation) on TCP port 8000, waiting for TCP/IP socket connections by invokers using the `soap` protocol. The behavioural code in `main` defines a request-response input on operation `sum`. In this paper, we implicitly assume that all services are deployed with the `concurrent` execution modality for supporting multiple session executions, from [29,26]. This means that whenever the first input of the behavioural definition of a service can receive a message, Jolie will spawn a dedicated process instance to execute the rest of the behaviour. This instance will be equipped with a local variable state and will proceed in parallel to all the others. Therefore, in our example, whenever our service receives a request for operation `sum` it will spawn a new parallel process instance. The latter will enter into the body of `sum`, assign to variable `resp` the result of adding the subnodes `x` and `y` of the request message, and finally send back this result to the original invoker.

3 Extending Jolie to HTTP

We extend Jolie to support web applications by introducing a new protocol for communication ports, named `http`. The latter follows the HTTP protocol specifications and integrates the Jolie message semantics to that of HTTP and its different content encodings.

3.1 Message transformation

The basic issue to address for integrating Jolie with the HTTP protocol is establishing how to transform HTTP messages in messages for the input and output primitives of Jolie and vice versa. Hereby we discuss primarily how our implementation manages request messages; response messages are similarly handled. The (abstract) structure of a *request message* in HTTP is the following one:

$$Method\ \ Resource\ \ \texttt{HTTP}/Version\ \ Headers\ \ Body$$

Above, *Method* specifies the action the client intends to perform and can be picked by a static set of keywords, such as `GET`, `PUT`, `POST`, etc. *Resource* is a URI telling which resource the client is requesting. *Version* is the HTTP protocol version of the message. *Headers* include descriptive information (such as the encoding of the message body) or even configuration parameters that are supposed to be respected by the receiver (e.g., the wish to close the connection immediately after the response is sent, which our implementation handles automatically). Finally, *Body* contains the content of the HTTP message.

A Jolie message is composed by an operation name and a (structured) value. Hence, we need to establish where to retrieve (or write) them in an HTTP message. For operations, we interpret the path part of the *Resource* URI as the operation name. We have chosen not to use *Method* for operations since it cannot assume user-defined names, as operations require. *Method* can still be read and written by Jolie programs through a configuration parameter of our extension, described in § 3.2. The message value, instead, is obtained from the *Body* part and the rest of the *Resource* URI. We need the latter to access REST interfaces and to be able to decode *querystring* parameters as Jolie values.

An HTTP message content may be encoded in different formats. Our `http` extension handles querystrings, form encodings (simple and multipart), XML, JSON [4], and GWT-RPC [3] [1]. Programmers can use the `format` parameter (§ 3.2) to control the data format to use for encoding and decoding messages. For incoming request messages, if the `Content-Type` HTTP header is present then it is used to auto-detect the data format of *Body*. If a response is sent back from Jolie and the request format was JSON and GWT-RPC, then `http` defaults to the same format for encoding the response content. As an example of message translation, the HTTP message:

```
GET /sum?x=2&y=3 HTTP/1.1
```

would be interpreted as a Jolie message for operation `sum`. The querystring `x=2&y=3` would be translated to a structured value with subnodes `x` and `y`, containing respectively the strings "2" and "3".

Querystrings and other common message formats used in web applications, such as HTML form encodings, present the problem of not carrying type information. Instead, they simply carry string representations of values that were potentially typed on the invoker's side. However, type information is necessary for supporting services such as the sum service in § 2, which requires integers for its input values. To cope with such cases, we introduce the notion of *automatic type casting*. Automatic type casting reads incoming messages that do not carry type information (such as querystrings or HTML forms) and tries to cast their content values to the types expected by the service interface for the message operation. As an example, consider the querystring `x=2&y=3` above. Since its HTTP message is a request for operation `sum`, the automatic type casting mechanism would retrieve the typing for the operation and see that nodes `x` and `y` should have type `int`. Therefore, it would try to re-interpret the strings "2" and "3" as integers before giving the message to the Jolie interpreter. Of course, type casting may fail; e.g., in `x=hello` the string `hello` cannot be cast to an integer for `x`. In such cases, our `http` protocol will send back a `TypeMismatch` fault to the invoker. The latter may catch the fault in its web user interface code.

[1] We have also developed a companion GWT-RPC client library, called `jolie-gwt`, for a more convenient access to web service written in Jolie. In this library, the operation name is encoded in the HTTP message content instead of the *Resource* field, following the GWT specifications.

3.2 Configuration Parameters

We augment the deployment syntax of Jolie to support *configuration parameters* for our `http` protocol. Specifically, these can be accessed through *(assign)* and *(alias)* instructions put aside the protocol declaration of a port. For instance, the following input port definition

```
inputPort MyInput {
  /* ... */
  Protocol: http {
    .default = "d"; .debug = true;
    .method -> m
  }
}
```

would set the `default` parameter to `"d"`, set the `debug` parameter to `true`, and bind the `method` parameter to the value of variable `m` in the current Jolie process instance.

We briefly describe some notable configuration parameters. All of them can be modified at runtime using the standard Jolie constructs for dynamic port binding [29] (omitted here). `default` allows to mark an operation as a special fallback operation that will receive messages that cannot be handled by any operation defined in its service's interface. `cookies` allows to store and retrieve data from browser cookies, by mapping cookie values in HTTP messages to subnodes in Jolie messages. `method` allows to read/write the *Method* field for the latest received/next to send HTTP message through the port. `format` can be used to force the data format of output HTTP messages, such as `json` (for JSON), or `xml` (for XML). `alias` allows to map values inside a Jolie message to resource paths in the HTTP message, to support interactions with REST services. `redirect` gives access to the `Location` field in HTTP, allowing to redirect clients to other locations. `cacheControl` allows to send directives to the client on how the responses sent to it should be cached. Finally, `debug` allows to print debug messages on screen whenever an HTTP message is sent or received.

4 Web Programming

In this section we discuss how `http` can be used to cover some useful web application patterns.

4.1 Modelling Web Servers

We first address how to program a web server for providing static content (e.g., the code and resources for a web user interface) to web clients.

The main challenge in dealing with modelling a web server is that, in Jolie and other service-oriented technologies such as WS-BPEL [30], a service interface is a statically defined set of operations. Differently, web servers make a dynamic

set of resources available to clients: the code for implementing a web server does not change if its set of exposed resources (from, e.g., a part of a filesystem) is modified.

To deal with dynamic resource names, i.e., resource names that we do not know at design time, we need a means to bridge the static definition of interfaces to them. We solve this point by using our `default` operation parameter. The default operation is a special operation marked as a fallback in case a client sends a request message with an operation that is not statically defined by the service. Instead, the message is wrapped in the following data structure (we omit some subnodes not relevant for this discussion):

```
type DefaultOperationHttpRequest:void {
        .operation:string
        .data:undefined
}
```

where `operation` is the name of the operation that has been requested by the client and `data` is the data content of the message.

`default` allows us to model a simple web server easily: whenever we receive a request for the default operation, we try to find a file in the local filesystem that has the same name as the operation originally requested by the client. We have used this mechanism to implement Leonardo [7], a web server implementation written in pure Jolie. For clarity, here we report a simplified version [2]:

Listing 1.1. 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 )
  }
```

[2] Remarkably, the entire implementation of Leonardo is made of only about 80 LOCs.

```
}
```

Above, we have set the `default` parameter for the `http` protocol in input port `HTTPInput` to operation `d`. Therefore, when a message for an unhandled operation is received through input port `HTTPInput`, it will be managed by the implementation of operation `d`. The body of the latter invokes operation `readFile` of the `File` service from the Jolie standard library, which reads the file with the same name as the originally request operation (`req.operation`). Finally, the data read from the file (`resp`) is returned back to the client.

4.2 Multiparty Sessions

We present an implementation sketch of an extended version of the process-aware scenario mentioned in the Introduction, where a user can access a `createNews` operation for posting a news on a website after she has successfully logged in through a `login` operation. We remind the reader that our `http` protocol accepts invocations with different formats, so we will leave the code for the user interface unspecified. E.g., we could use AJAX calls with the JSON format for calling operation `login`, or the following HTML form:

```
<form action="login" method="POST">
  <input type="text" name="user"/>
  <input type="password" name="pwd"/>
  <input type="submit"/>
</form>
```

Our scenario will execute as follows. First, the user will download the web interface from our service implementation. Afterwards, she will call the `login` operation for authenticating. We will use an external `Authenticator` service for checking the user's credentials. If the authentication is not successful, we will interrupt the process by throwing a fault (which will also automatically notify the user of the error). Otherwise, if the authentication is successful we will wait for the user to invoke the `createNews` operation. When the latter is invoked, we will notify in parallel two external services of the news creation request: `Logger` and `Moderator`. The former is used to log the user's operation. The latter, instead, is an external service handling another web application for moderating news creation requests. Specifically, service `Moderator` (omitted here) is responsible for showing the news creation request on a moderation list. Moderators can access the list through a web interface and get redirected to our service for approving or rejecting the news creation request.

A critical aspect of our implementation is the modelling of *sessions*, or conversations. 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 `createNews` arrives, how can we know if it is from the first user or the second? We address this issue by using correlation sets [30]. A correlation set specifies special variables that identify an internal service process from the others. In this example, we combine the correlation set

mechanism offered by Jolie [27] with HTTP cookies, which are usually employed for storing session identifiers in web browsers. Our example will have two correlation sets consisting of one variable each, respectively `userSid` and `modSid`. We will use the first to identify calls from the user, and the second to identify messages from the moderator. Having two separate identifiers for our process instance is a fundamental aspect of *multiparty sessions*, such as the one in this example, for reasons of security: 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 1.2. A moderated news service

```
/* Types, Interfaces, Output ports, ... */

inputPort MyInput {
Location: "socket://localhost:8000/"
Protocol: http {
  .cookies.userSid = "userSid";
  .cookies.modSid = "modSid";
  .default = "d"
}
Interfaces: MyIface
}

cset { userSid: createNews.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 ) }
  } ] {
    createNews( news );
    csets.modSid = new;
    { log@Logger( cred.username )
      | notify@Moderator( csets.modSid ) };
    [ approve() ] { /* ... */ }
    [ reject() ]  { /* ... */ }
  }
}
```

Above, we have reused the web server pattern from § 4.1 to provide the resources for the (omitted) web user interface to web browser clients. We combine that pattern with a process that starts with a request-response 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, given by primitive `new`; otherwise, we throw a fault `AuthFailed`, therefore notifying the client and interrupting the execution of the process. We return `csets.userSid` to the user through the response message for `login`. Observe that we configured `http` with `.cookies.userSid = "userSid"`. Hence, `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 `createNews`, 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 `createNews` is invoked, we instantiate our second correlation variable: `modSid`. Then, we use the parallel compositor `|` to notify the `Logger` and `Moderator` services 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.

Observe that our two correlation set definitions (the `cset` blocks before `main`) specify that operation `createNews` can be invoked only through correlation variable `userSid`, whereas `accept` and `reject` can be accessed only through `modSid` [3], modelling our aforementioned security aspect.

A remarkable aspect of the combination between our `http` extension and Jolie is that we abstract from where correlation data is encoded in the HTTP message. Instead of using a cookie, the web user interface may also send the value for a correlation variable through a querystring (enabling *process-aware hyperlinks*), or inside the HTTP message content. Our extension transparently support these different methods without requiring specific configuration.

4.3 Multi-tiering

In § 4.2, we have used a single service to handle both the content serving and the process execution. However, usually it is more desirable to separate the service responsible for serving content (the web server) from the service responsible for process execution. Ideally, this separation of concerns should allow to perform changes in the web server (e.g., implementing sitemaps or optimisations) abstracting from the internal code of the process executors and vice versa. More

[3] With respect to [27], we are assuming that operations are declared with request types with the same respective names.

in general, we want to stratify our service in different *tiers*, as in classical multi-tiered web architectures. We reach this objective by exploiting the *aggregation* mechanism [33]. Aggregation is a composition primitive where an input port exposes operations that are not implemented in its service behaviour, but are instead delegated to another external service. Using aggregation we can split the web server code and the process code from our news service in two separate services.

We show first the code for the service responsible for handling the news moderation process:

```
/* Types, Interfaces, Output ports, ... */

inputPort MyInput {
Location: "socket://localhost:8001/"
Protocol: soap
Interfaces: MyIface
}

cset { userSid: createNews.userSid }
cset { modSid:
  approve.modSid reject.modSid }

main {
  login( cred )( r ) { /* ... */ }
}
```

The code above is taken from Listing 1.2. 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 service is moved to the following separate service:

```
/* Types, Interfaces, Output ports, ... */

outputPort News { /* ... */ }

inputPort WebInput {
Location: "socket://localhost:8000/"
Protocol: http {
  .cookies.userSid = "userSid";
  .cookies.modSid = "modSid";
  .default = "d"
}
Interfaces: ContentIface
Aggregates: News
}
```

```
main {
  d( req )( resp ) { /* ... */ }
}
```

The service above implements the web content server for our web application. `ContentIface` is an interface defining only operation `d`. 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 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 through the default operation `d`.

Our `http` extension can be combined with aggregation also for handling Multi-Service Architectures, i.e. web architectures 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 the invocations from the web client to the aggregated services pass through the web server, this programming methodology respects the Same Origin Policy by design.

5 Related Work

The frameworks most similar to ours are those for modelling business processes, such as WS-BPEL [30], WS-CDL [40], and YAWL [38]. Differently from our approach, these tools are integrated with web applications through third-party tools, increasing the overall complexity of the system. Some of the ideas presented in this paper (e.g., the `default` parameter for implementing web servers) may be easily applied to WS-BPEL, making our work a useful reference.

Other works offer tools for supporting the development of process-aware web applications. [34] presents a process-based approach to deal with user actions through web interfaces using EPML; like Jolie, EPML is formally specified and comes with an execution engine. JOpera comes with an integration layer for offering REST-based interfaces to business processes [31]. These solutions are formed by integrating separate modules for process modelling, computation, and system integration. In contrast, our framework addresses all these aspects using the same language. EPML can integrate with other languages to integrate user interfaces with process execution; we are currently investigating in a similar direction (see § 6.1, *Scaffolding of User Interfaces*).

Hop [36,11] and GWT [3] are programming frameworks that deal with the programming of both the user interface and the server-side application logic using a single codebase, which gets then compiled in the code for the client interface and the services. Differently, in this paper we do not deal with the generation of client code. Instead, we support the seamless integration of existing technologies (HTML, AJAX calls, JSON, ...) with our services. The client code compiled from GWT projects can be reused with our `http` extension, which is able to parse GWT requests. Hop and GWT do not support the process-aware modelling primitives offered by our framework, nor its ability to compose many services on a same HTTP communication endpoint (through aggregation).

Our `default` configuration parameter for `http` allows a service implementation to catch and reply to invocations for operations that were not known at design time. The same aspect has been previously modelled through mobility mechanisms for names in process calculi, e.g. in [35,17]. However, our approach is less powerful because the cited approaches elevate the received operation names at the language level; e.g., a service may receive an operation name through a variable and then use the latter in *(input)* and *(output)* primitives as operations. This is not possible in our behavioural language, since operations in input and output statements must be statically defined. We purposefully chose not to support this kind of mobility, since it would have made the definitions of Jolie interfaces change at runtime. This would break the basic assumption of statically defined operations, which is used in some static analyses developed for Jolie (see [27,29,33]), in the WSDL standard for Web Services [41], and in formal theories models for the verification of concurrent programming languages [20,32].

6 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 elegantly 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 [5]. Remarkably, our integration is *seamless*, meaning that existing Jolie code can easily be ported to HTTP by changing only the `Protocol` part of its communication ports to `http` and its configuration language. An important consequence 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 all the techniques developed for the verification and execution of Jolie programs [27,29] can be transparently applied to the process-aware web application logic written in our framework.

Our framework has been evaluated in the development of industrial products and is now used in production at italianaSoftware [23], a software development company that uses Jolie as reference programming language. For instance, the company's website [23] and Web Catalogue, a proprietary E-Commerce plat-

form with a codebase of more than 400 services, use the framework and the programming patterns presented in this paper.

6.1 Future Work

Hereby, we discuss possible future extensions of our work.

Reversibility. A common problem in handling the interaction between a web user interface and a business process is that the user may decide to take a step back in the execution flow (e.g., by pressing the "Back" button). This possibility must be manually taken into account in the design of the process, increasing its complexity. We plan to extend Jolie with *reversibility techniques* [24], which allow distributed processes to be reversed to previous states by transparently dealing with the required communications to the involved parties.

Scaffolding of User Interfaces [4]. We will develop a scaffolding tool for user interfaces, starting from the process structure of a service. Specifically, given a behaviour B in Jolie, it would be possible to automatically generate a user interface that follows the communication structure of the behaviour. This would be in line with the notions of *duality* formalised in [20,18].

Behavioural analyses. Since our framework makes the process logic of a web application explicit, it would be possible to develop a tool for checking that the invocations performed by a web user interface written in, e.g., Javascript, match the structure of their corresponding Jolie service. We plan to investigate this possibility using solutions similar to the ones presented in [21,15].

Declarative data validation. Our framework exploits the message data types declared in the interfaces of a Jolie service to *validate* the content of incoming messages from web user interfaces. We plan to extend this declarative support to data validation by introducing an assertion language for message types that can check more complex properties (e.g., integer ranges, regular expressions, ...).

Extensions to other web protocols. We will extend our work by adding support to new emerging protocols for web application communications, such as WebSocket [22] and SPDY [2].

Acknowledgments

We have benefited from discussions with Claudio Guidi and Dimitrios Kouzapas.

References

1. BPMN Version 2.0. `http://www.omg.org/spec/BPMN/2.0/`.
2. Google SPDY. `https://developers.google.com/speed/spdy/`.

[4] The initial idea for this point comes originally from Claudio Guidi, during a private conversation.

3. Google Web Toolkit. http://code.google.com/webtoolkit/.
4. JavaScript Object Notation. http://www.json.org/.
5. Jolie HTTP extension. https://jolie.svn.sourceforge.net/svnroot/jolie/trunk/extensions/http.
6. Jolie website. http://www.jolie-lang.org/.
7. Leonardo Web Server. http://www.sourceforge.net/projects/leonardo/.
8. SODEP protocol. http://www.jolie-lang.org/wiki.php?page=Sodep.
9. Workflow Patterns. http://www.workflowpatterns.com/.
10. XML-RPC. http://www.xmlrpc.com/.
11. G. Boudol, Z. Luo, T. Rezk, and M. Serrano. Reasoning about web applications: An operational semantics for hop. *ACM Trans. Program. Lang. Syst.*, 34(2):10, 2012.
12. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.
13. D. A. Chappell. *Enterprise Service Bus - Theory in practice*. O'Reilly, 2004.
14. M. Dumas, W. M. P. van der Aalst, and A. H. M. ter Hofstede. *Process-Aware Information Systems: Bridging People and Software Through Process Technology*. Wiley, 2005.
15. S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL*, pages 299–312, 2010.
16. C. Guidi. *Formalizing languages for Service Oriented Computing*. PhD. thesis, University of Bologna, 2007. http://www.cs.unibo.it/pub/TR/UBLCS/2007/2007-07.pdf.
17. C. Guidi and R. Lucchi. Formalizing mobility in service oriented computing. *JSW*, 2(1):1–13, 2007.
18. D. Hirschkoff, J.-M. Madiot, and D. Sangiorgi. Duality and i/o-types in the -calculus. In *CONCUR*, pages 302–316, 2012.
19. K. Honda, A. Mukhamedov, G. Brown, T.-C. Chen, and N. Yoshida. Scribbling interactions with a formal foundation. In *ICDCIT*, pages 55–75, 2011.
20. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, pages 122–138, 1998.
21. R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, and K. Honda. Type-safe eventful sessions in java. In *ECOOP*, pages 329–353, 2010.
22. IETF. WebSocket protocol. http://tools.ietf.org/html/rfc6455.
23. italianaSoftware s.r.l. italianaSoftware. http://www.italianasoftware.com/.
24. I. Lanese, C. A. Mezzina, and J.-B. Stefani. Reversing higher-order pi. In *CONCUR*, pages 478–493, 2010.
25. A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *Proceedings of ESOP'07*, volume 4421 of *LNCS*, pages 33–47, 2007.
26. F. Montesi. Jolie: a Service-oriented Programming Language. Master's thesis, University of Bologna, Department of Computer Science, 2010.
27. F. Montesi and M. Carbone. Programming services with correlation sets. In *ICSOC*, pages 125–141, 2011.
28. F. Montesi, C. Guidi, and G. Zavattaro. Composing Services with JOLIE. In *Proceedings of ECOWS 2007*, pages 13–22, 2007.
29. F. Montesi, C. Guidi, and G. Zavattaro. Service-oriented Programming with Jolie. Draft paper (submitted for publication), 2012. http://www.itu.dk/people/fabr/papers/soc_jolie/.

30. OASIS. WS-BPEL Version 2.0. http://docs.oasis-open.org/wsbpel/.
31. C. Pautasso and E. Wilde. Push-enabling restful business processes. In *ICSOC*, pages 32–46, 2011.
32. B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.
33. M. D. Preda, M. Gabbrielli, C. Guidi, J. Mauro, and F. Montesi. Interface-based service composition with aggregation. In *ESOCC*, pages 48–63, 2012.
34. D. Rossi and E. Turrini. Designing and architecting process-aware web applications with epml. In *SAC*, pages 2409–2414, 2008.
35. D. Sangiorgi and D. Walker. *The π-Calculus. A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.
36. M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *OOPSLA Companion*, pages 975–985, 2006.
37. W. M. P. van der Aalst. Verification of workflow nets. In *ICATPN*, pages 407–426, 1997.
38. W. M. P. van der Aalst and A. H. M. ter Hofstede. Yawl: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005.
39. W3C. SOAP Specifications. http://www.w3.org/TR/soap/.
40. W3C. Web Services Choreography Description Language Version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/.
41. W3C. Web Services Description Language. http://www.w3.org/TR/wsdl.