Process-aware web programming with Jolie

Fabrizio Montesi

University of Southern Denmark
Department of Mathematics and Computer Science, Campusvej 55, 5230 Odense M, Denmark.
fmontesi@imada.sdu.dk

arXiv:1410.3712v2 [cs.DC] 15 Oct 2014

Abstract. We present a programming language and runtime, which extends the Jolie programming language, for the native modelling of process-aware web information systems, i.e., web information systems based upon the execution of business processes. Our main contribution is to offer a unifying approach for the programming of distributed architectures on the web, which can capture web servers, stateful process execution, and the composition of services via mediation in a system. We discuss many examples around these aspects and show how they can be captured using our approach, covering, e.g., static content serving, multiparty sessions, and the evolution of web systems.

1 Introduction

A Process-Aware Information System (PAIS) is an information system based upon the execution of business processes. These systems are required in many application scenarios, from inter-process communication to automated business integration [1]. Processes are typically expressed as structures that determine the order in which communications should be performed in a system. These structures can be complex and of different kinds; a systematic account can be found at [2]. For this reason, many formal methods [3, 4, 5, 6], tools [7, 8, 9, 10, 11], and standards [12, 13, 14] have been developed to provide languages for the definition, verification, and execution of processes. In these works, compositionality plays a key role to make the development of processes manageable. For example, in approaches based on process calculi, complex process structures are obtained by composing simpler ones through the usage of standard composition operators such as sequence, choice, and parallel (see, e.g., [15]). Other approaches follow similar ideas using graphical formal models, e.g., Petri Nets [16, 17].

In the last two decades, web applications have become increasingly process-aware. Web processes – i.e., processes inside of a web information system – are usually implemented server-side on top of *sessions*, which track incoming messages related to the same conversation. Sessions are supported by a shared memory space that lives through different client invocations. Differently from the aforementioned approaches for designing processes, the major languages and platforms for developing web applications (e.g., PHP, Ruby on Rails, and Java EE) do not support the explicit programming of process structures. As a workaround, programmers have to simulate processes using bookkeeping variables in the shared memory space of sessions. For example, consider a process in a Research Information Service (RIS) where a user has to authenticate through a `login` operation before accessing another operation, say `addPub`, for registering a

المنارة للاستشارات

www.manaraa.com

publication. This would be implemented by defining the `login` and the `addPub` operations separately. The code for `login` would update a bookkeeping variable in the session state and the implementation for `addPub` would check that variable when it is invoked by the user. Although this approach is widely adopted, it is also error-prone: since processes can assume quite complex structures, simulating them through bookkeeping variables soon becomes cumbersome. Consequently, the produced code may be poorly readable and hard to maintain.

The limitations described above can be avoided by adopting a multi-layered architecture. For example, it is possible to stratify an application by employing: a web server technology (e.g., Apache Tomcat) for serving content to web browsers; a web scripting language (e.g., PHP) for programmable request processing; a process-oriented language (e.g., WS-BPEL [12]) for modelling the application processes; and, finally, mediation technologies such as proxies and ESB [18] 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, consisting of a language and its runtime, 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, a general-purpose service-oriented programming language that can handle both the structured modelling of processes and their integration within larger distributed systems [11, 19]. We briefly introduce Jolie in § 2.

1.1 Contributions

Our main contribution is to obtain a unifying technology for the programming of processes, web technologies (web servers and scripting), and mediation services (e.g., proxies). We proceed as described below.

Web processes. We extend the Jolie language to support 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: data structures in Jolie are transparently transformed to HTTP messages and vice versa (§ 3.1). These transformations can be configured using parameters that allow developers to map information in HTTP headers, e.g., cookies, to application data (§ 3.3).

Web servers as processes. We develop a web server, called Leonardo (§ 4), using our approach. The web server is given as a simple process that (i) receives the name of the resource a client wants to access, then (ii) reads the content of such resource, and (iii) sends the content back to the client. Leonardo is an example of the fact that, in our framework, a web server is not a separate technology but it is instead a simple case of a process. We also show how to extend Leonardo to handle simple CRUD operations over HTTP.

www.manaraa.com

Sessions. We combine our HTTP extension for Jolie with message correlation, a mechanism used in service-oriented technologies to route incoming messages to their respective processes running inside of a service [12, 11]. We first show that this combination is adequate wrt existing practice: it enables Jolie processes to use the standard methodology of tracking client-server web sessions using unique session identifiers (§ 5.1). Then, we generalise such methodology to program multiparty sessions, i.e., structured conversations among a process and multiple external participants [20] (§ 5.2).

Architectural programming. We present how to obtain separation of concerns in a web architecture implemented with our approach, by combining HTTP with aggregation, a Jolie primitive for programming the structure of service networks [21, 22, 11] (§ 6). We demonstrate the usefulness of this combination by implementing a multi-layered system that integrates different components. We also discuss how to deal with the evolution of software architectures obtained with our approach (§ 6.2).

2 Overview of Jolie

Jolie [23] is a general-purpose service-oriented programming language, released as an open-source project [19] and formally specified as a process calculus [4, 24]. In this section, we briefly describe some aspects of Jolie that are relevant for our discussion. We refer the interested reader to [11] and [19] for a more comprehensive presentation of the language. Readers who are already familiar with the Jolie language may skip this section and resume reading from § 3.

2.1 Jolie programs

Every Jolie program defines a service and consists of two parts: *behaviour* and *deployment*. A behaviour defines the implementation of the operations offered by the 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 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:

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

Above, D represents the deployment and B the behavior of the program.

2.2 Behaviour

We report (a selection of) the syntax of behaviours in Figure 1. A behaviour B can use primitives for performing communications, computation, and their composition in processes. We briefly comment the syntax. Terms *(input)*, *(output)*, and *(input choice)* implement communications. An input η can either be a one-way or a request-response,

www.manaraa.com

$$
\begin{array}{llr}
B ::= \eta & & \textit{(input)} \\
\quad | \ \overline{\eta} & & \textit{(output)} \\
\quad | \ [\ \eta_1\ ]\ \{\ B_1\ \} \ldots [\ \eta_n\ ]\ \{\ B_n\ \} & & \textit{(input choice)} \\
\quad | \ \mathbf{if}\,(\ e\ )\ B_1\ \mathbf{else}\ B_2 & & \textit{(cond)} \\
\quad | \ \mathbf{while}\,(\ e\ )\ B & & \textit{(while)} \\
\quad | \ B\ \mathbf{;}\ B' & & \textit{(seq)} \\
\quad | \ B\ \mathbf{|}\ B' & & \textit{(par)} \\
\quad | \ \mathbf{throw}\,(\mathtt{f}) & & \textit{(throw)} \\
\quad | \ \mathtt{x}\ = e & & \textit{(assign)} \\
\quad | \ \mathtt{x}\ \mathtt{->}\ \mathtt{y} & & \textit{(alias)} \\
\quad | \ \mathbf{nullProcess} & & \textit{(inact)}
\end{array}
$$

$$
\begin{array}{llr}
\eta\ ::= \mathtt{o}\,(\mathtt{x}) & & \textit{(one-way)} \\
\quad | \ \mathtt{o}\,(\mathtt{x})\,(e)\,\{\ B\ \} & & \textit{(request-response)} \\
\\
\overline{\eta}\ ::= \mathtt{o@OP}\,(e) & & \textit{(notification)} \\
\quad | \ \mathtt{o@OP}\,(e)\,(\mathtt{y}) & & \textit{(solicit-response)}
\end{array}
$$

Fig. 1: Jolie, syntax of behaviours (selection).

following the WSDL standard [25]. Statement *(one-way)* receives a message for operation o and stores its content in variable x. Term *(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. Dual to input statements, an output $\overline{\eta}$ can be a *(notification)* or a *(solicit-response)*. A *(notification)* sends a message to OP containing the value of the evaluation of expression e. Term *(solicit-response)* sends a message to OP containing the evaluation of e and then waits for a response from the invoked service, storing it afterwards in variable y. In both *(notification)* and *(solicit-response)*, OP is the name of an *output port*, which acts as a reference to an external service. Output ports are concretely defined in the deployment part of a program; we will present them in § 2.3.

Term *(input choice)* implements an input-guarded choice; it is similar to the pick primitive in WS-BPEL [12]. Specifically, the construct waits for a message for any of the inputs in $\eta_1, \ldots, \eta_n$. When a message for one of these inputs is received, say for η_i where $1 \leq i \leq n$, then the statement is executed as follows, in order: (i) all the other branches in the choice (i.e., all the $[\ \eta_j\ ]\ \{\ B_j\ \}$ such that $j \neq i$) are discarded; (ii) η_i is executed; and, finally, B_i is executed.

Terms *(cond)* and *(while)* implement, respectively, the standard conditional and iteration constructs. Term *(seq)* models sequential execution and reads as: execute B, wait for its termination, and then run B'. In term *(par)*, instead, B and B' are run in parallel. Term *(throw)* throws a fault signal f, interrupting execution. If a fault signal is thrown from inside a request-response body, the invoker of the request-response statement is automatically notified of the fault [26]. We omit the syntax for handling faults, which is not necessary for reading this paper.

المنارة للاستشارات
www.manaraa.com

Term *(assign)* stores the result of the evaluation of expression e in variable x. Term *(alias)* makes variable x an alias for variable y, i.e., after its execution accessing x will be equivalent to accessing y. Term *(inact)* denotes the empty behaviour (no-op).

Example 1 (Structured data). Jolie natively supports the manipulation of structured data. In Jolie's memory model the program state is a tree (possibly with arrays as nodes, see [21]), and every variable, say x, can be a *path* to a node in the memory tree. Paths are constructed through the dot "." operator; for instance, the following sequence of assignments

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

would lead to a state containing a tree with root label person. For the reader familiar with XML, a corresponding XML representation would be:

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

2.3 Deployment

We introduce now (a selection of) the syntax of deployments. A deployment includes definitions of *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 one the dual concept of the other, and their respective syntaxes are quite similar. Both kinds of ports are based on the three basic elements of *location*, *protocol* and *interface*. Their syntax is reported in Figure 2. In the syntax of ports, i.e.,

```
IP ::= inputPort Port        OP ::= outputPort Port
Port ::= id {
        Location: Loc
        Protocol: Proto
        Interfaces: iface_1, ..., iface_n
        }
```

Fig. 2: Jolie, syntax of ports (selection).

term $Port$, Loc is a URI (Uniform Resource Identifier) that defines the location of the port; $Proto$ is an identifier referring to the data protocol to use in the port, which specifies how input or output messages through the port should be respectively decoded or encoded; the identifiers $\mathsf{iface}_1, \ldots, \mathsf{iface}_n$ are references to the interfaces accessible through the port.

Jolie supports several kinds of 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 are based, respectively, on Unix sockets, Bluetooth communication channels, and local in-memory channels (channels implemented using shared

www.manaraa.com

memory). Some supported instances of *Proto* are `sodep` [19] (a binary protocol, optimised for performance), `soap` [27], and `xmlrpc` [28].

The interfaces referred to by a communication port define the operations that can be accessed through that port. Each interface defines a set of operations, along with their respective (i) operation types, defining if an operation is to be used as a one-way or a request-response, and (ii) types of carried messages. For example, the following code

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

defines an interface named `SumIface` with a request-response operation, called `sum`, that expects input messages of type `SumT` and replies with messages of type **`int`** (integers). Data types for messages follow a tree-like structure; for example, we could define `SumT` as follows:

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

We 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`**.

Example 2 (A complete Jolie program). We 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 SumInput {
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 behaviour of the service is contained in the **`main`** procedure, the entry point of execution in Jolie. The behaviour 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 [21]. This means that whenever the first input of the behavioural definition of a service receives a message from the network, Jolie will spawn a dedicated process with

المنارة للاستشارات
www.manaraa.com

a local memory state to execute the rest of the behaviour. This process 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 `req`, 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`, and by extending the language of deployments to support configuration parameters for protocols. The protocol follows the specifications of HTTP, and integrates the message semantics of Jolie to that of HTTP and its different content encodings. In this section, we discuss the main aspects of our implementation.

3.1 Message transformation

The central 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:

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

Above, $Method$ specifies the action that the client intends to perform and can be picked by a static set of keywords, such as `GET`, `PUT`, `POST`, etc. Term $Resource$ is a URI path telling which resource the client is requesting. Term $Version$ is the HTTP protocol version of the message. The term $Headers$ may include descriptive information on the message $Body$, e.g., the type of its content (`Content-Type`), or parameters that influence the behaviour of the receiver, e.g., the wish to keep the underlying connection open for future requests (`Connection:keep-alive`). Finally, $Body$ contains the content (payload) of the HTTP message.

A Jolie message consists of an operation name (the operation the message is meant for) and a structured value (the content of the message) [21]. Hence, we need to establish where to read or write these elements in an HTTP message. For operation names, we interpret the path part of the $Resource$ URI as the operation name[1]. The $Method$ of an HTTP message, instead, is read and written by Jolie programs through a configuration parameter of our extension, described later in § 3.3. The value of a Jolie message is

[1] The reader familiar with HTTP and the bridging of the RPC and REST paradigms may ask why we did not choose $Method$ instead of $Resource$ to read/write operation names. The reason is generality: operation names are picked by the programmer from an infinite set of identifiers, whereas $Method$ can only be picked from the static set of methods defined in the specifications of HTTP. Therefore, choosing $Method$ would mean that some Jolie programs could not be ported to HTTP without requiring to change the names of the operations they use.

المنارة للاستشارات
www.manaraa.com

obtained from $Body$ and the rest of the $Resource$ URI (query and fragment parts). We use the latter to decode querystring parameters as Jolie values.

The content of an HTTP message may be encoded in one of different formats. Our `http` extension handles querystrings, form encodings (simple and multipart), XML, JSON [29], and GWT-RPC[2] [30]. Programmers can use the `format` parameter (§ 3.3) to control the data format for encoding and decoding messages. Most of the times, however, this decision is performed automatically and the programmer does not need to know which format is used. For example, for incoming request messages from clients, if the `Content-Type` HTTP header is present then it is used to auto-detect the data format of $Body$; by default, the `http` protocol will use the same data format to encode the reply for the respective client. 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"`.

3.2 Automatic type casting

Querystrings and other common message formats used in web applications, such as HTML form encodings, do not carry type information. Instead, they carry only string representations of values; the information on the types that these values may have had in the code of the sender (e.g., in Javascript) is therefore lost. However, type information is necessary for supporting services such as the sum service in Example 2, which specifically requires its input values to be integers. To handle such cases, we introduce the mechanism 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` that we discussed before. 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 behaviour of the Jolie program. There are cases that type casting may fail to handle; for example, in `x=hello` the string `hello` cannot be cast to an integer for `x`. In such cases, our `http` protocol will send a `TypeMismatch` fault back to the invoker.

3.3 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)*

[2] We have also developed a companion GWT-RPC client library, called `jolie-gwt`, for a more convenient access to web services written in Jolie by integrating with the standard GWT development tools.

المنارة للاستشارات
www.manaraa.com

statements that can be written inside a code block immediately after declaring the `http` protocol in a port. For instance, consider the following input port definition:

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

The code above 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 configuration parameters. All of them can be modified at runtime using the standard Jolie constructs for dynamic port binding, from [21], which we omit here. Parameter **default** allows to mark an operation as a special fallback operation for receiving messages that cannot be handled by any other operation defined in the interface of the enclosing input port. Parameter `cookies` allows to store and retrieve data from browser cookies, by mapping cookie values in HTTP messages to subnodes in Jolie messages. Parameter `method` allows to read/write the $Method$ field of HTTP messages. Parameter `format` can be used to force the data format of HTTP messages, such as `json` (for JSON), or `xml` (for XML). The parameter `alias` allows to map values inside a Jolie message to resource paths in the HTTP message, to support interactions with REST services. Parameter `redirect` gives access to the **Location** header in HTTP, allowing to redirect clients to other locations. The parameter `cacheControl` allows to send directives to the client on how the responses sent to it should be cached. Finally, parameter `debug` allows to print the HTTP messages sent and received through the network on screen.

3.4 Examples

We report some examples about how our `http` protocol implementation integrates with some standard mechanisms of web technologies.

Example 3 (Access from web browsers). Let us consider a modification of the sum service from Example 2, where we change the input port to use the `http` protocol that we developed:

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

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

inputPort SumInput {
Location: "socket://localhost:8000"
Protocol: http
Interfaces: SumIface
```


المنارة للاستشارات
www.manaraa.com

```
}

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

Now, our implementation of `http` allows us to access the service above in multiple ways. The most obvious is to write a Jolie client using an output port with the `http` protocol. A more interesting way is to use a web browser. For example, we can use the service by passing parameters through a querystring; navigating to the following URL is valid:

`http://localhost:8000/sum?x=2&y=3`

Accessing the URL above would show the following content on the browser:

```
<sumResponse>5</sumResponse>
```

The standard format used for responses is XML, as above. Responses from Jolie services using `http` can of course also be themed using, e.g., HTML and Javascript (we refer to the online documentation for more information about this aspect [19]).

Another possibility is to use HTML forms, such as the one that follows:

```
<form action="sum" method="GET">
  <input type="text" name="x"/>
  <input type="text" name="y"/>
  <input type="submit"/>
</form>
```

The content displayed as a response in the web browser would be the same XML document as before.

We also offer support for AJAX programming. The following Javascript snippet calls the `sum` operation using jQuery [31]: first, it reads the values for `x` and `y` from two text fields (respectively identified in the DOM by the names `x` and `y`); then, it sends their values to the Jolie service by encoding them as a JSON structure; and, finally, it displays the response from the server in the DOM element with id `result`:

```
$.ajax(
  'sum', { x: $('#x').val(), y: $('#y').val() },
  function( response ) { $("#result").html( response ); }
);
```

Our implementation of the `http` protocol for Jolie auto-detects the format of messages sent by clients, so the sum service does not need to distinguish among all the different access methods shown above: they are all handled using the same Jolie code.

www.manaraa.com

Example 4 (Accessing REST services). We exemplify how to access REST services, where resources are identified by URLs, using our configuration parameters. In this example we invoke the DBLP server, which provides bibliographic information on computer science articles [32]. We use DBLP to retrieve the BibTeX entry of an article, given the dblp key of the latter (i.e., the identifier of such article in dblp). The code follows:

```
include "console.iol"

type FetchBib:void { .dblpKey:string }

interface DBLPIface {
RequestResponse: fetchBib( FetchBib )( string )
}

outputPort DBLP {
Location: "socket://dblp.uni-trier.de:80/"
Protocol: http {
  .osc.fetchBib.alias = "rec/bib2/%!{dblpKey}.bib";
  .method = "html" }
Interfaces: DBLPIface
}

main
{
        r.dblpKey = args[0];
        fetchBib@DBLP( r )( bibtex );
        println@Console( bibtex )()
}
```

In the example above, we start by importing the `Console` service from the Jolie standard library. We then declare an output port towards the DBLP server. The interesting part here is the usage of parameter `osc.fetchBib.alias`, which passes to our implementation a configuration for parameter `alias` that is specific to operation `fetchBib` (`osc` stands for operation specific configuration). The value of the alias for operation `fetchBib` specifies how to map calls for that operation to resource paths that the DBLP server understands. The interface offered by DBLP for retrieving bibtex entries is REST-based, with paths rooted at "rec/bib2/". As an example, assume that we wanted to retrieve the bibtex entry for the famous book on the C language by Kernighan and Ritchie [33]. Its dblp key is "books/ph/KernighanR78"; therefore, the bibtex entry can be accessed at the URL:

http://dblp.uni-trier.de/rec/bib2/books/ph/KernighanR78.bib

In our implementation, we capture this kind of patterns for REST paths by providing a syntax for replacing parts of paths with the value of a subnode in a request message. For instance, the term `%!{dblpKey}` in the alias for operation `fetchBib` means

www.manaraa.com

that that part of the path will be replaced with value of the sub node `dblpKey` in messages sent for that operation on port `DBLP`. The behaviour of the service is simple: we invoke operation `fetchBib` reading the dblp key we want from the first command line argument that Jolie is invoked with; then, we print the received bibtex entry on screen.

An extended version of this example is deployed as a tool at [34].

4 Web Servers

In Example 3 we have seen how to make operations in a Jolie service accessible by invokers using HTTP (in the example, operation `sum`): any operation in a Jolie service can be exposed to HTTP clients just by changing the protocol of its related input port(s) to `http`. This technique covers scenarios in which the interface that we want to expose over HTTP is statically defined as a finite set of operations, which is the typical situation when using service-oriented technologies such as Jolie or WS-BPEL [12]. However, web servers do not fall into this category. A web server allows clients to access files, e.g., web pages, images, and JavaScript libraries. Since files can be created and deleted during execution, we cannot statically map each single file to an operation name as would be required by our methodology in § 3. In this section, we discuss how to deal with this kind of situations by introducing default operations.

Default operations bridge the mutating nature of dynamic resource sets that web servers have to offer (such as parts of a filesystem) to the static operation names used in processes. Specifically, a default operation is a special operation marked as a fallback in case a client sends a request message for an operation that is not statically defined by the service. In this case, 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 (or of the resource) that has been requested by the client and `data` is the data content of the message.

A default operation is set through the parameter **`default`** of the `http` protocol, and can either be associated to the $Method$ field of incoming HTTP messages or be defined as a "catch-all" operation in case no other more specific operation can be found. For example, the following configuration states that requests for undefined operations with HTTP method `PUT` should be handled by operation `put`, requests for undefined operations with method `GET` should be handled by operation `get`, and all other requests for undefined operations should be handled by operation `d`.

```
Protocol: http {
  .default = "d";
  .default.get = "get";
  .default.put = "put"
```


www.manaraa.com

```
}
```

Example 5 (Leonardo Web Server). Parameter **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 [35], a web server implementation written in pure Jolie. For clarity, here we report a simplified version. The entire implementation of Leonardo consists of only about 80 LOCs, showing that our language pushes many of the details of dealing with HTTP to the underlying implementation; many of these details can be accessed through configuration parameters when needed. Leonardo can be downloaded at [35].

```
/* ... */

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, 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.

Example 6 (CRUD Web Servers). We extend Leonardo to a simple web server supporting CRUD operations (Create, Read, Update, Delete). As usual, we map create and update to `PUT` requests, read to `GET`, and delete to `DELETE`. The code follows:

```
/* ... */

interface MyInterface {
```


المنارة للاستشارات
www.manaraa.com

```
RequestResponse:
  get( DefaultOperationHttpRequest )( undefined )
  put( DefaultOperationHttpRequest )( void )
  delete( DefaultOperationHttpRequest )( bool )
}

inputPort HTTPInput {
Location: "socket://localhost:80/"
Protocol: http {
  .default.get = "get";
  .default.put = "put";
  .default.delete = "delete"
}
Interfaces: MyInterface
}

main {
  [ get( req )( resp ) {
    readFile@File( req.operation )( resp )
  } ] { nullProcess }

  [ put( req )() {
    f.filename = req.operation;
    f.content -> req.data;
    writeFile@File( f )( resp )
  } ] { nullProcess }

  [ delete( req )( resp ) {
    delete@File( req.operation )( resp )
  } ] { nullProcess }
}
```

In the code above, `GET` requests are served by operation `get`, which reads the requested file and replies with its content. Similarly, operation `put` uses the Jolie standard library to write a file with the data sent by the invoker, and operation `delete` deletes a file from the filesystem.

5 Sessions

A main aspect of web-based information systems is the modelling of *sessions*, which allow to relate different incoming messages to the same logical "conversation". In this section, we present how to program sessions over HTTP with our extension of the Jolie language. A major benefit is that sessions are process-aware: the order in which messages are sent and received over different operations is syntactically explicit, and it is enforced without requiring bookkeeping variables.

المنارة للاستشارات
www.manaraa.com

5.1 Binary sessions

We start by addressing binary sessions, i.e., sessions with exactly two participants [36]. Consider the scenario mentioned in the Introduction about a Research Information Service (RIS), where the RIS allows users to add a publication to a repository after having successfully logged in. This structure is expressed by the following behaviour:

```
login( cred )( r ) { checkCredentials };
addPub( pub )
```

Above, `login` is a request-response operation that, when invoked, checks the received credentials by calling the subprocedure `checkCredentials`. If the latter does not throw a fault, the process proceeds by making operation `addPub` available.

Suppose now that, e.g., two users are logged in at the same time in a service with the behaviour above. The service would have then two separate process instances, respectively dedicated to handle the two clients. When a message for operation `addPub` arrives in this situation, how can we know if it is from the first user or the second? We address this kind of issues by using correlation sets, as defined in [24]. A correlation set declares special variables that identify an internal service process from the others. In our example we use the following correlation set declaration:

```
cset { userKey: addPub.userKey }
```

Above, we used the **`cset`** keyword to declare a correlation set consisting of variable `userKey`. We will use `userKey` to distinguish users that have logged in. Variable `userKey` is associated to the subnode `userKey` in incoming messages for operation `addPub`. This means that whenever a message for operation `addPub` is received from the network, Jolie will assign the message to the internal running process with the same value for the correlation variable `userKey`. We can now write a working implementation of the service:

```
inputPort RISInput {
/* ... */
Protocol: http
}

cset { userKey: addPub.userKey }

define checkCredentials { /* ... */ }
define updateDB { /* ... */ }

main
{
  login( cred )( r ) {
    checkCredentials;
    r.userKey = csets.userKey = new
  };
  addPub( pub );
```


www.manaraa.com

```
  updateDB
}
```

Our RIS allows the creation of new processes by invoking operation `login`. If the procedure `checkCredentials` does not throw a fault, then the process creates a fresh value for the correlation variable **csets**.`userKey` using the **new** keyword. The process sends the value of **csets**.`userKey` back to the client through variable `r`. Then, the process waits for an invocation of operation `addPub` and stores the incoming message in variable `pub`. The correlation set declaration in the program guarantees that only invocations with the same user key as that returned by operation `login` will be given to this process. We finally update the internal database of the RIS using the (unspecified) procedure `updateDB`.

Integrating cookies with correlation sets Our implementation of the RIS requires clients to write the `userKey` as a subnode in the messages they send to operation `addPub`. Since this may be cumbersome in the case of many operations that require correlation, web applications typically use HTTP cookies to store this kind of information. Our `http` protocol integrates cookies with message correlation through the `cookies` parameter, which allows to map cookies to subnodes in Jolie variables. We change the definition of input port `RISInput` to the following:

```
inputPort RISInput {
/* ... */
Protocol: http { .cookies.userKeyCookie = "userKey" }
}
```

The parameter assignment `.cookies.userKeyCookie = "userKey"` instructs our `http` protocol implementation to store the value of the cookie `userKeyCookie` in subnode `userKey` for incoming messages, and vice versa for outgoing messages.

In general, our `http` extension allows developers to abstract from where correlation data is encoded when programming a service behaviour. Instead of using a cookie, the web user interface may also send the value for a correlation variable in other ways, e.g., through a querystring (enabling *process-aware hyperlinks*), a JSON or XML subnode, or an element in an HTML form encoding. Our extension transparently support these different methods without requiring changes in the behaviour of a service.

5.2 Multiparty Sessions

As far as binary sessions are concerned, there is not much difference between standard session identifiers as used, e.g., in PHP, and correlation sets, aside from the fact that the generation and sending of correlation variables is explicit programmed in Jolie behaviours. However, correlation sets are more expressive when it comes to providing (i) compound session identifier based on multiple values, as in BPEL [12], and (ii) multiple identifiers for the same process. We are particularly interested in the second aspect, since it allows us to model *multiparty* sessions, i.e., sessions with more than two participants.

المنارة للاستشارات
www.manaraa.com

Multiparty sessions are useful when considering scenarios with multiple actors that need to be coordinated to reach a common goal. As an example, we extend our RIS implementation to deal with a use case from the Pure software by Elsevier [37]. In Pure, when a user (e.g., a research scientist) adds a publication, a moderator (e.g., the head of the scientist's department) has to be notified of the change. Then, the moderator has to choose whether to approve or reject the newly added publication for confirmation in the database, after reviewing the data inserted by the user. We show the code for this multiparty version of our RIS implementation in the following:

```
inputPort RISInput {
/* ... */
Protocol: http { .cookies.userKeyCookie = "userKey" }
}

outputPort Logger { /* ... */ }
outputPort Moderator { /* ... */ }

cset { userKey: addPub.userKey }
cset { modKey: approve.modKey reject.modKey }

define checkCredentials { /* ... */ }
define updateDB { /* ... */ }

main
{
  login( cred )( r ) {
    checkCredentials;
    r.userKey = csets.userKey = new
  };
  addPub( pub );
  noti.bibtex = pub.bibtex;
  noti.modKey = csets.modKey = new;
  { log@Logger( pub.bibtex ) | notify@Moderator( noti ) };
  [ approve() ] {
    log@Logger( "Accepted " + pub.bibtex );
    updateDB
  }
  [ reject() ] {
    log@Logger( "Rejected " + pub.bibtex )
  }
}
```

Above, we have added the output ports `Logger`, an external service that maintains a log of actions that we assume the user can read, and `Moderator`, an external service playing the role of the moderator in our scenario. We have also added a new correlation set for variable `modKey` (moderator key), which we use to track incoming messages

www.manaraa.com

from the moderator of a session. The correlation set declares also that the moderator may use modKey to invoke operations approve and reject. In the behaviour, the code is unchanged until after we receive an invocation for operation addPub. Now, after we receive a request for operation addPub, we prepare a notification noti for the moderator containing (i) the descriptor of the publication (we assume that it is given by the user in BibTeX format), and (ii) the moderation key modKey (which is instantiated as a fresh value with the keyword **new**). Then we use the parallel construct of Jolie to concurrently send a message to, respectively, the Logger on operation log (to log the user's request) and the Moderator on operation notify (to notify the moderator of the user's request). The process now enters into an input choice on operations approve and reject, which can be invoked only by the moderator; this is because the correlation set declaration of variable modKey requires it to be present for invocations of these operations, and we sent the value of modKey only to the moderator. If approve is invoked, then we log the approval and we update the database of publications. Otherwise, if reject is invoked, we log the rejection only.

6 Layering

In the previous sections, we focused separately on how to use our extension of Jolie to program web servers (§ 4) and structured process-aware sessions (§ 5). Typically, a real-world web architecture has to deal with both aspects. In this section, we show how they can be combined in our context by building multi-layered architectures.

6.1 Aggregation

A simple way of designing a service that serves content *and* provides process-aware sessions is to combine the respective operations in the same behaviour as an input choice. Consider the following code:

```
/* ... */
main
{
  [ get( req )( resp ) { /* ... */ } ] { nullProcess }
  [ login( cred )( r ) { /* ... */ } ] { /* ... */ }
}
```

Above, we assume that the ports and correlation sets are configured by merging the configurations found in Example 6 and the RIS implementation in § 5.2. Then, operation get would serve HTML and JavaScript files to clients, which could also invoke operation login to access the behaviour of the RIS.

While combining the code for a web server with that of sessions with complex structures as done above is simple, in the long term it also leads to code that is hard to maintain due to poor separation of concerns: all concerns are mixed in the same service. Ideally, separate concerns should be addressed by separate services. This methodology, however, raises the question of how services addressing separate concerns can be composed together as a single system that clients can access without knowing the inner

www.manaraa.com

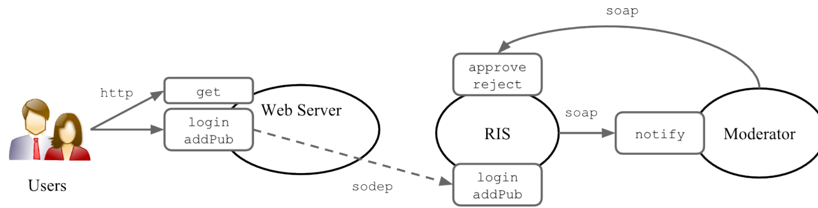

Fig. 3: Architecture of the RIS scenario.

complexity of the system. We tackle this issue by integrating our `http` protocol implementation with the notion of service aggregation found in Jolie [21, 11].

Aggregation is a Jolie primitive that allows a service to expose the interfaces of other services on one of its input ports, in addition to its own interfaces. In the remainder, we refer to the service using aggregation as aggregator and to the services it aggregates as aggregated services. The semantics of aggregation is a simple generalisation of the mechanism used in proxy services: when a message from the network reaches an aggregator, the aggregator checks whether the message is for an operation in (i) one of its own interfaces or (ii) the interfaces of an aggregated service. In the first case, the message is given to the behaviour of the aggregator; in the second case, the aggregator forwards the message to the aggregated service providing the operation requested in the message.

Using aggregation in combination with our `http` protocol we can easily build a multi-layered web architecture for our RIS scenario, where services communicate using different protocols as needed. We depict the architecture in Figure 3, where circles represent services, rectangles represent the interfaces exposed by services, full arrows represent dependencies from actors (users or services) to services, and dashed arrows represent aggregations; each arrow is annotated with the protocol used for communications. We comment the architecture. Users can access the web server using a web browser, through the `http` protocol. Requests for files, intended to be for the user interface (e.g., HTML pages), are handled directly by the web server through operation `get`. Instead, invocations of operations `login` and `addPub` are forwarded to the RIS by aggregation. The web server and the RIS communicate using the `sodep` protocol, for performance (`sodep` is a binary protocol). As in § 5.2, the RIS uses an additional service, Moderator, to decide whether publications should be accepted into the system. The RIS and the Moderator services communicate using the `soap` protocol. Below, we exemplify how our architecture can be implemented. We assume that the Moderator service is externally provided, and focus instead on the web server and the RIS.

Web server. The code of the web server follows:

```
/* ... */

outputPort RIS {
Location: "socket://www.ris-example.com:8090/"
```


المنارة للاستشارات
www.manaraa.com

```
Protocol: sodep
Interfaces: RISIface
}

inputPort WebServerInput {
Location: "socket://www.webserver-example.com:80/"
Protocol: http {
  .default.get = "get";
  .cookies.userKeyCookie = "userKey"
  /* ... */
}
Interfaces: GetIface
Aggregates: RIS
}

main {
  get( req )( resp ) {
    /* ... */
    readFile@File( req.operation )( resp )
  }
}
```

Our web server implements only the operation `get`, which serves static files to clients. It also aggregates the RIS by using the aggregation instruction **Aggregates:**`RIS` in its input port, where `RIS` is an output port pointing to the RIS. Therefore, all invocations from users for the operations offered by the RIS will be automatically forwarded to the latter. Observe that output port `RIS` uses the `sodep` protocol: our implementation automatically takes care of translating incoming HTTP messages from users destined to the RIS into binary `sodep` messages. In general, the programmer does not need to worry about data format transformations in our extension of the Jolie language: messages are implicitly converted to/from the HTTP format as needed.

RIS (Research Information Service). The code for the RIS is the same as that shown in § 5.2, with the exception that we now use `sodep` as communication protocol and that we removed the usage of the external service `Logger` for simplicity:

```
inputPort RISInput {
Location: "socket://www.ris-example.com:8090/"
Protocol: sodep
Interfaces: RISIface
}

outputPort Moderator {
Location: "socket://www.moderator-example.com:8080/"
Protocol: soap
Interfaces: ModeratorIface
}
```


www.manaraa.com

```
cset { userKey: addPub.userKey }
cset { modKey: approve.modKey reject.modKey }

define checkCredentials { /* ... */ }
define updateDB { /* ... */ }

main
{
  login( cred )( r ) {
    checkCredentials;
    r.userKey = csets.userKey = new
  };
  addPub( pub );
  noti.bibtex = pub.bibtex;
  noti.modKey = csets.modKey = new;
  notify@Moderator( noti );
  [ approve() ] {
    updateDB
  }
  [ reject() ] {
    /* ... */
  }
}
```

6.2 Evolvability

Implementing multi-layered web architectures using our approach, i.e., combining `http` with aggregation, results in systems that are robust wrt future modifications, or their *evolution*. We distinguish between *vertical* and *horizontal modifications*, which respectively represent modifications that influence an existing chain of aggregations or new ones.

A vertical modification is a modification of an interface aggregated by another service. In our example, changing the code of the RIS to add, remove, or change the type of an operation in interface `RISIface` would be a vertical modification, because `RISIface` is aggregated by the web server. Vertical modifications do not require any intervention on the rest of the architecture, as aggregation is a parametric mechanism: the web server simply needs to be restarted to read the new definition of interface `RISIface`.

Horizontal modifications deal with the addition or removal of operations without requiring an intervention on the behaviour of existing services. Assume that, as an example, we wanted to add the possibility to import publications from the DBLP bibliography service to our RIS by offering a new operation called `import`. We could implement this new feature by changing the code of the RIS, both its interface and behaviour. However, in some scenarios this may not be possible, e.g., the RIS may be a

المنارة للاستشارات
www.manaraa.com

black box to which we do not have access (third-party proprietary code), or the RIS cannot be modified due to quality or security regulations. We deal with this kind of situations by developing the new operations we need in a new service, and then by aggregating this service together with the RIS in the web server. The resulting situation in our example scenario is depicted in Figure 4. The only difference between Figure 4

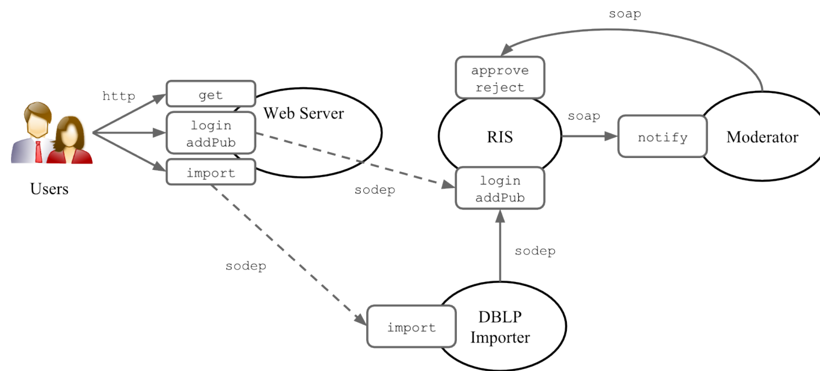

Fig. 4: Architecture of the RIS scenario with DBLP importer.

and our previous architecture from Figure 3 is the presence of a new service, called Importer, offering operation `import`; the web server now aggregates Importer together with the RIS, to make operation `import` accessible by users through web browsers. We report the updated code for the web server and the importer.

Web server. For the web server, we simply need to add an output port towards the importer service and aggregate it in the input port of the server. We report only the code interested by our changes, the rest remains the same as in § 6.1.

```
/* ... */

outputPort Importer {
Location: "socket://localhost:8009/"
Protocol: sodep
Interfaces: ImporterIface
}

outputPort RIS { /* ... */ }

inputPort WebServerInput {
/* ... */
Aggregates: RIS, Importer
}
```


المنارة للاستشارات
www.manaraa.com

```
/* ... */
```

By changing the web server as done above, invocations for operation `import` will now be redirected to the importer service.

Observe that, by using aggregation, all the invocations from the web client to the aggregated services pass through the web server. This implies that our programming methodology respects the standard Same Origin Policy by design, allowing the web application run by users to access the aggregated services regardless of where the latter are located.

Importer service. The code for the importer service follows:

```
/* ... */

outputPort DBLP {
Location: "socket://dblp.uni-trier.de:80/"
Protocol: http {
  .osc.fetchBib.alias = "rec/bib2/%!{dblpKey}.bib"
  /* ... */
}
Interfaces: DBLPIface
}

outputPort RIS { /* ... */ }

inputPort ImporterInput {
Location: "socket://localhost:8009/"
Protocol: sodep
Interfaces: ImporterIface
}

main
{
  import( request );

  dblpReq.dblpKey = request.dblpKey;
  fetchBib@DBLP( dblpReq )( result );

  addReq.bibtex = result;
  addReq.userKey = request.userKey;

  addPub@RIS( addReq )
}
```

The importer service offers a single operation, `import`, which takes as input a message containing two subnodes: `dblpKey`, the dblp key of the publication to import

المنارة للاستشارات
www.manaraa.com

from DBLP, and `userKey`, which must be a valid user key for a session inside of the RIS. The idea is that a user has to invoke operation `login` before using operation `import`, thus opening a session in the RIS, and that she then invokes `import` with the `userKey` it got as a response from `login`. After receiving a message for operation `import`, the importer service proceeds by invoking the DBLP service to retrieve the BibTeX record stored therein for the dblp key passed by the user. Finally, after retrieving the BibTeX record, the importer asks the RIS to add it through operation `addPub`.

7 Related Work

To the best of our knowledge, our work is the first to propose a unified language for dealing with the programming of web servers, scripting, and the architecture of service systems in the web by means of mediator services. We have been deeply inspired by related work in these areas, described below.

The frameworks most similar to ours are those for modelling business processes, such as WS-BPEL [12], WS-CDL [13], and YAWL [7]. Differently from our approach, these tools are integrated with web applications through third-party tools. Some of the ideas presented in this paper (e.g., the **`default`** parameters for implementing web servers) may be easily applied to WS-BPEL, making our work a potential reference.

Other works offer tools for supporting the development of process-aware web applications. The papers [38, 39] propose a formally-specified language, implemented in Java, for defining processes that can transparently access resources on the web using a fixed set of primitive operations; the language supports similar process structured as those found in Jolie behaviours, although in our case operations are user-defined. In [40], the authors present 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 [41]. 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 § 8, *Scaffolding of User Interfaces*).

Hop [42, 43] and GWT [30] 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 developed an integration between existing technologies (HTML, AJAX calls, JSON, etc.) and our services, by using our `http` protocol to convert the data structures handled by these technologies to/from those handled by Jolie. This leaves the choice of which framework to use for implementing the web user interface to the developer. The client code compiled from GWT projects can be reused with our `http` extension, which is able to parse GWT requests. HipHop [44] is an extension of Hop based on the synchronous language Esterel [45], which introduces orchestration primitives to Hop. The major difference between HipHop and our solution is that behavioural code in Jolie is kept separated from deployment information, making it reusable in different environ-

المنارة للاستشارات
www.manaraa.com

ments, whereas HipHop code mixes the two aspects (for example, cookies in Hop are handled in behavioural code).

Another work that shares some of our aims is the Bigwig project, which offers a language for the programming of session-aware web applications [46]. Our language for behaviours is more expressive than that of Bigwig, which does not support, e.g., the programming of processes using multiparty sessions; however, in our setting we obtain this expressiveness by requiring the programmer to manually handle session identifiers in processes, whereas in Bigwig these are handled automatically. Bigwig is based on the Apache web server, whereas our approach is self-contained: web servers, services, and service mediators (which Bigwig does not handle) are all written in Jolie.

Our `default` configuration parameters 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 theoretically modelled through mobility mechanisms for names in process calculi, e.g., in [47, 48, 49]. Our approach is less powerful because these theoretical models elevate the received operation names at the language level: a service may receive an operation name, store it in a variable, and then use the latter in *(input)* and *(output)* primitives as an operation. This is not possible in our behavioural language, since operations in input and output statements are statically defined. We chose not to support this kind of mobility, since it would make the definitions of Jolie interfaces change at runtime. This would break the basic assumption of statically defined operations used in the formal model and implementation of the Jolie language, which goes out of the scope of this paper. It would also make Jolie fundamentally different from other standards for web services, such as WSDL [25], with unclear consequences on their integration. Static operation names are also used in many formal models for the verification of concurrent programming languages (e.g., session types [36]), which we are interested in adopting for Jolie in the future.

8 Discussion and Future Extensions

We discuss some aspects of web programming with Jolie and future extensions related to the work that we presented in this article.

Performance and Evaluation in Production Environments. We leave a rigorous benchmarking of our framework as future work. However, preliminary informal tests and experience in production environments show that the performance of Jolie-based HTTP services is certainly comparable with that of programs using Servlet applications running on J2SE, services written in BPEL, or web applications written in other web frameworks such as PHP or Ruby on Rails. This is not surprising, as these frameworks have to deal with the transformation of HTTP messages to their own internal data representation as in our case; the only difference is that Jolie supports also other message formats than those supported in web technologies.

Our framework has been evaluated in the development of industrial products and is now used in production systems at italianaSoftware, a software development company that uses Jolie as reference programming language [50]. For instance, the website of the company and Marco Polo, a proprietary E-Commerce platform with a codebase of

المنارة للاستشارات
www.manaraa.com

more than 400 services, use the framework and the programming techniques presented in this paper.

Holystic Approach. The main motivation of this work is to lower the complexity of programming web-based systems by offering a unified language to capture their different aspects. However, the current widespread approach of having a specialised technology for each of such aspects may have an advantage when it comes to the required knowledge to use them, as each technology can be studied in isolation. For example, the administrator of a web server in a larger system has to learn only how to use the web server software she uses, abstracting from the other technologies in the rest of the system (where, e.g., WS-BPEL or ESB technologies may be present).

When dealing with only one aspect of web programming, learning how to use a specific software to deal with such aspect may be less time consuming than learning the Jolie language, which is more general. A possible solution for this problem could be to develop Domain Specific Languages (DSLs), supported by Integrated Development Environments (IDEs), that are compiled to Jolie code. The idea is that a specific DSL would deal with one aspect of web programming, while retaining the benefits of having a single underlying language for the different components of a web system. It is still uncertain whether this step would really be necessary, for two reasons. The first reason is that for simple tasks, such as serving static content, we can offer a reference implementation such as the Leonardo web server in § 4, as an alternative to other standard implementations such as the Apache Web Server. The second reason is that many web systems require dealing with multiple aspects of web programming. In those cases, it can take less time to learn Jolie than learning about the available specific technologies to cover all the use cases that the work we presented can address; this would amount to learning, at least, a web server, an orchestration (e.g., WS-BPEL), and a service mediation technologies.

Adoption. How and when should a solution such as that proposed in this paper be adopted in real-world software projects? We discuss an answer by distinguishing between two main cases: the development of new systems and the extension of existing systems.

When dealing with the development of an entirely new web system, Jolie offers a simple and unified language for dealing with the architectural aspects (layering, deployment), the behavioural aspects (application logic), and the serving of static content (web servers). Therefore, Jolie is now a candidate for the rapid prototyping of a web system. Since Jolie integrates with other technologies, starting with our framework does not imply that the final system must be written entirely in Jolie: different parts may be refined later either by using Jolie or other technologies (e.g., Java, WS-BPEL).

When dealing with the extension of an existing system, or even the development of a new system that has to integrate with other existing systems, Jolie can be considered as a glue framework for bridging services based on different technologies. In particular, it is convenient to use the simple syntax of Jolie for writing processes that direct the behaviour of other services in a system. In general, the integration capabilities of Jolie allow for its introduction in a development team by starting from a single service in a larger system, which can be used by the team to assess whether Jolie should be adopted

المنارة للاستشارات
www.manaraa.com

in other parts of the system after seeing how it performs. We conjecture that this step-by-step introduction of web services written in Jolie will be key for its adoption by expert web developers. We are currently following this development methodology in some software projects at the University of Southern Denmark, for the improvement of the web-based tools provided to students and staff.

Since Jolie is a relatively new language, most programmers are still unfamiliar with it and therefore their training must be taken into account in a project. An advantage of a unified framework such as ours, though, is that it allows to understand multiple aspects of web-based systems by learning a single language. With the rise of more complex and structured web-based systems, we believe that there can be a motivation to learn Jolie even for developers who are expert in more established technologies.

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. We plan to extend Jolie with *reversibility techniques* [51], 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. The explicit structure of processes written in Jolie allows us to statically see the workflow that a user interface should follow when interacting with a Jolie service. We could use this aspect to develop a scaffolding tool for user interfaces, starting from the process structure of a service. Specifically, given a behaviour 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 [36, 52].

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. The techniques presented in [8, 9] may offer useful first steps towards this aim.

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 (§ 3.2). We plan to extend this declarative support for data validation by introducing an assertion language for message types that can check more complex properties (e.g., integer ranges and regular expressions).

Extensions to other web protocols. Our work lays the foundations for using Jolie as a fully-fledged language to handle HTTP-based systems. By following the same approach, it would be possible to develop support for new emerging protocols for the web, such as WebSockets [53] and SPDY [54].

المنارة للاستشارات
www.manaraa.com

9 Conclusions

We have presented a framework for the programming of process-aware web systems, where processes are used as a holystic approach to capture the development of the different components of such systems, such as web servers, orchestrators, and service mediators. 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 evolvability. Our `http` extension is open source and is included in the standard distribution of Jolie, along with the language additions that we introduced to support protocol configurations [55, 19]. 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` with some configuration parameters. An important consequence is that the programmer does not need to deal with the differences between the data formats employed in HTTP messages (form encodings, querystrings, JSON, etc.), since they are all automatically translated to Jolie data structures. This also means that all the techniques developed for the verification and execution of Jolie programs (as the typing system in [24] for correlation sets) can be transparently applied to the process-aware web application logic written in our framework.

Acknowledgements

The author thanks Claudio Guidi, Saverio Giallorenzo, and the anonymous referees of the original conference version of this paper for their useful comments and discussions. This work has been supported by the Danish Council for Independent Research (Technology and Production), grant n. DFF–4005-00304.

References

1. M. Dumas, W. M. P. van der Aalst, A. H. M. ter Hofstede, Process-Aware Information Systems: Bridging People and Software Through Process Technology, Wiley, 2005. 1
2. Workflow Patterns, `http://www.workflowpatterns.com/`. 1
3. W. M. P. van der Aalst, Verification of workflow nets, in: ICATPN, 1997, pp. 407–426. 1
4. C. Guidi, Formalizing languages for Service Oriented Computing, PhD. thesis, University of Bologna, `http://www.cs.unibo.it/pub/TR/UBLCS/2007/2007-07.pdf` (2007). 1, 3
5. A. Lapadula, R. Pugliese, F. Tiezzi, A calculus for orchestration of web services, in: ESOP, 2007, pp. 33–47. 1
6. M. Carbone, F. Montesi, Deadlock-freedom-by-design: multiparty asynchronous global programming, in: R. Giacobazzi, R. Cousot (Eds.), The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013, ACM, 2013, pp. 263–274. `doi:10.1145/2429069.2429101`. URL `http://doi.acm.org/10.1145/2429069.2429101` 1
7. W. M. P. van der Aalst, A. H. M. ter Hofstede, Yawl: yet another workflow language, Inf. Syst. 30 (4) (2005) 245–275. 1, 24
8. R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, K. Honda, Type-safe eventful sessions in java, in: ECOOP, 2010, pp. 329–353. 1, 27

www.manaraa.com

9. S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, A. Z. Caldeira, Modular session types for distributed object-oriented programming, in: POPL, 2010, pp. 299–312. 1, 27
10. K. Honda, A. Mukhamedov, G. Brown, T.-C. Chen, N. Yoshida, Scribbling interactions with a formal foundation, in: ICDCIT, Vol. 6536 of LNCS, Springer, 2011, pp. 55–75. 1
11. F. Montesi, C. Guidi, G. Zavattaro, Service-oriented programming with jolie, in: Web Services Foundations, 2014, pp. 81–107. doi:10.1007/978-1-4614-7518-7_4.
URL http://dx.doi.org/10.1007/978-1-4614-7518-7_4 1, 2, 3, 19
12. OASIS, Web Services Business Process Execution Language, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html. 1, 2, 3, 4, 12, 16, 24
13. W3C WS-CDL Working Group, Web services choreography description language version 1.0, http://www.w3.org/TR/ws-cdl-10/ (2004). 1, 24
14. Business Process Model and Notation, http://www.omg.org/spec/BPMN/2.0/. 1
15. R. Lucchi, M. Mazzara, A pi-calculus based semantics for WS-BPEL, J. Log. Algebr. Program. 70 (1) (2007) 96–118. doi:10.1016/j.jlap.2006.05.007.
URL http://dx.doi.org/10.1016/j.jlap.2006.05.007 1
16. W. M. P. van der Aalst, The application of petri nets to workflow management, Journal of Circuits, Systems, and Computers 8 (1) (1998) 21–66. doi:10.1142/S0218126698000043.
URL http://dx.doi.org/10.1142/S0218126698000043 1
17. J. L. Peterson, Petri nets, ACM Comput. Surv. 9 (3) (1977) 223–252. doi:10.1145/356698.356702.
URL http://doi.acm.org/10.1145/356698.356702 1
18. D. A. Chappell, Enterprise Service Bus - Theory in practice, O'Reilly, 2004. 2
19. Jolie, Programming Language, http://www.jolie-lang.org/. 2, 3, 6, 10, 28
20. K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, in: POPL, Vol. 43(1), ACM, 2008, pp. 273–284. 3
21. F. Montesi, Jolie: a Service-oriented Programming Language, Master's thesis, University of Bologna, Department of Computer Science (2010). 3, 5, 6, 7, 9, 19
22. M. D. Preda, M. Gabbrielli, C. Guidi, J. Mauro, F. Montesi, Interface-based service composition with aggregation, in: ESOCC, 2012, pp. 48–63. 3
23. F. Montesi, C. Guidi, G. Zavattaro, Composing Services with JOLIE, in: ECOWS, 2007, pp. 13–22. 3
24. F. Montesi, M. Carbone, Programming services with correlation sets, in: ICSOC, 2011, pp. 125–141. 3, 15, 28
25. Web Services Description Language, http://www.w3.org/TR/wsdl. 4, 25
26. F. Montesi, C. Guidi, I. Lanese, G. Zavattaro, Dynamic Fault Handling Mechanisms for Service-Oriented Applications, in: ECOWS, 2008, pp. 225–234. 4
27. SOAP Specifications, http://www.w3.org/TR/soap/. 6
28. XML-RPC, http://www.xmlrpc.com/. 6
29. JavaScript Object Notation, http://www.json.org/. 8
30. Google Web Toolkit, http://code.google.com/webtoolkit/. 8, 24
31. The jQuery Foundation, jQuery, http://www.jquery.com/. 10
32. University of Trier and Schloss Dagstuhl, dblp: computer science bibliography, http://www.jquery.com/. 11
33. B. W. Kernighan, D. Ritchie, The C Programming Language, Prentice-Hall, 1978. 11
34. F. Montesi, DBLP Tools, http://www.fabriziomontesi.com/dblp/. 12
35. Leonardo Web Server, http://www.sourceforge.net/projects/leonardo/. 13
36. K. Honda, V. Vasconcelos, M. Kubo, Language primitives and type disciplines for structured communication-based programming, in: ESOP'98, Vol. 1381 of LNCS, Springer-Verlag, Heidelberg, Germany, 1998, pp. 22–138. 15, 25, 27

المنارة للاستشارات
www.manaraa.com

37. Elsevier, Pure, http://www.elsevier.com/online-tools/research-intelligence/products-and-services/pure. 17
38. M. Bravetti, File managing and program execution in web operating systems, CoRR abs/1005.5045.
URL http://arxiv.org/abs/1005.5045 24
39. M. Bravetti, Formalizing restful services and web-os middleware, in: Web Services and Formal Methods - 10th International Workshop, WS-FM 2013, Beijing, China, August 2013, Revised Selected Papers, 2013, pp. 48–68. doi:10.1007/978-3-319-08260-8_4.
URL http://dx.doi.org/10.1007/978-3-319-08260-8_4 24
40. D. Rossi, E. Turrini, Designing and architecting process-aware web applications with epml, in: SAC, 2008, pp. 2409–2414. 24
41. C. Pautasso, E. Wilde, Push-enabling restful business processes, in: ICSOC, 2011, pp. 32–46. 24
42. M. Serrano, E. Gallesio, F. Loitsch, Hop: a language for programming the web 2.0, in: OOPSLA Companion, 2006, pp. 975–985. 24
43. G. Boudol, Z. Luo, T. Rezk, M. Serrano, Reasoning about web applications: An operational semantics for hop, ACM Trans. Program. Lang. Syst. 34 (2) (2012) 10. 24
44. G. Berry, M. Serrano, Hop and hiphop: Multitier web orchestration, in: Distributed Computing and Internet Technology - 10th International Conference, ICDCIT 2014, Bhubaneswar, India, February 6-9, 2014. Proceedings, 2014, pp. 1–13. doi:10.1007/978-3-319-04483-5_1.
URL http://dx.doi.org/10.1007/978-3-319-04483-5_1 24
45. G. Berry, The foundations of esterel, in: Proof, Language, and Interaction, Essays in Honour of Robin Milner, 2000, pp. 425–454. 24
46. C. Brabrand, A. Møller, M. I. Schwartzbach, The ¡Bigwig¿ Project, ACM Trans. Internet Technol. 2 (2) (2002) 79–114. doi:10.1145/514183.514184.
URL http://doi.acm.org/10.1145/514183.514184 25
47. R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, I and II, Information and Computation 100 (1) (1992) 1–40,41–77. 25
48. D. Sangiorgi, D. Walker, The π-calculus: a Theory of Mobile Processes, Cambridge University Press, 2001. 25
49. C. Guidi, R. Lucchi, Formalizing mobility in service oriented computing, JSW 2 (1) (2007) 1–13. 25
50. italianaSoftware s.r.l., http://www.italianasoftware.com/. 25
51. I. Lanese, C. A. Mezzina, J.-B. Stefani, Reversing higher-order pi, in: CONCUR, 2010, pp. 478–493. 27
52. D. Hirschkoff, J.-M. Madiot, D. Sangiorgi, Duality and i/o-types in the -calculus, in: CONCUR, 2012, pp. 302–316. 27
53. IETF, WebSocket protocol, http://tools.ietf.org/html/rfc6455. 27
54. Google SPDY, https://developers.google.com/speed/spdy/. 27
55. Jolie HTTP extension, https://jolie.svn.sourceforge.net/svnroot/jolie/trunk/extensions/http. 28

المنارة للاستشارات
www.manaraa.com