Service-oriented programming with Jolie
DRAFT of 29 February 2012

Fabrizio Montesi[1], Claudio Guidi[2], and Gianluigi Zavattaro[3]

[1] IT University of Copenhagen
fmontesi@itu.dk
[2] University of Padova, Dipartimento di Matematica
cguidi@math.unipd.it
[3] University of Bologna, INRIA Focus Research Team
zavattar@cs.unibo.it

Abstract. The wide adoption of service-oriented computing has led to a heterogeneous scenario formed by different technologies and specifications. Examples can be found both at the design level —the frameworks for defining services and those for defining their coordination feature fundamentally different primitives— and at the implementation level — different communication technologies are used depending on the context. In this paper we present Jolie, a fully-fledged service-oriented programming language. Jolie addresses the aforementioned heterogeneity in two ways. On the one hand, it combines computation and composition primitives in an intuitive and concise syntax. On the other hand, the behaviour and deployment of a Jolie program are orthogonal: they can be independently defined and recombined as long as they have compatible typing.

1 Introduction

Service-Oriented Computing (SOC) is a design methodology that focuses on the composition of autonomous entities in a system, called *services*. This requires abstracting from the implementation details of the involved services, which is usually done by imposing a standard communication mechanism between the entities in an SOA (Service-Oriented Architecture). For instance, the Web Services specifications [42] impose the use of the SOAP protocol [41], which builds on XML [39] as a data format and HTTP [40] as the transport. Applying such a restriction, it is possible to have SOAs where each service is potentially implemented in a different technology, such as Java, C, or C#.

SOC is widely adopted in many different settings; the following is just a short list of notable examples. Web Services are widespread and supported by many industrial technologies, such as Java and .NET; they are especially used in enterprise software development. Applications in modern Linux distributions, e.g. hardware information services and desktop environment components as in the KDE SC [6] and GNOME [3], communicate locally using the D-Bus technology [2]; in the Windows operating system, DCOM was created to address the same issue. Many web applications expose REST APIs to allow external applications to interact with them. All of the aforementioned technologies make it possible for applications to communicate by means of loosely coupled messaging systems. The adoption of SOC, however, has led to a problem of fragmentation. Many different service-oriented technologies and specifications, such as the ones listed above, target specific requirements and cannot be integrated without ad-hoc interventions, which usually imply the writing of some adapters for the message formats and the communication semantics. In other words, there are many technologies and applications based on common conceptual ground that are unable to interoperate without ad-hoc interventions, which can be very costly, hard to maintain, and prone to breaking wrt future system modifications.

From the perspective of the methodologies and tools for composing SOAs the situation is less fragmented, but there is a marked separation between *behavioural* and *architectural* composition. Behavioural composition deals with the specific series of interactions (message exchanges) to be performed in order to reach a goal. For example, an E-Commerce service supporting the purchase of some products may offer a `buy` functionality implemented by composing a warehouse service for sending the product to the client and a bank service for handling the payment. Services that behaviourally compose other services are usually called *orchestrators*. The most renowned technology for performing behavioural composition is WS-BPEL [34], a language based on the Web Services specifications. Architectural composition, on the other hand, deals with the topological structure of an SOA, managing its execution and integration. For example, an application server may manage the execution of multiple applications in the same environment; or, a proxy may be used to bridge two SOAs that run in separate networks. A more generic approach to bridging is represented by *mediators*. These may work on different levels, e.g., by allowing a service available on Bluetooth or a LAN to communicate with another service on the Internet, or by performing data format conversion. Notable examples of mediators are all the Enterprise Service Bus (ESB) technologies [14] and the aforementioned D-Bus [2]. Differently from the case of behavioural composition, we are not aware of programming languages supporting architectural composition: the latter is usually obtained through tools that are specific to some architectural patterns.

To the best of our knowledge, the literature lacks proposals of languages that enable SOA designers to deal effectively with both behavioural and architectural aspects, by providing a satisfactory support for solving the technological fragmentation problem reported above. We argue that offering such a language would simplify greatly the design of SOAs, since designers would have to deal with a single and homogeneous set of concepts instead of many different tools.

In this paper we present the Jolie programming language [23], our proposal for filling this gap. Jolie is the result of our attempt to obtain a common denominator that coherently offers the main features of SOC and their integration with existing technologies. We aim at offering a programming language for defining the base services, their organization in an SOA, and the behaviour of the orchestrators responsible for the supervision of the interactions among the services, possibly using different communication technologies. In our opinion, Jolie is the first language that positively responds to the problems of heterogeneity of both service communication technologies and compositional aspects.

A Jolie program defines a service and is a composition of two parts, called *behaviour* and *deployment*. A behaviour defines the implementation of the functionalities offered by a service; behavioural primitives include communication and computation constructs. However, these do not deal with how communications are supported: they abstractly refer to *communication ports*, which are assumed to be correctly defined in the deployment part. The latter deals with the actual definition of the necessary information for supporting communications. Therefore, communication ports establish a notion of compatibility between the behaviour and deployment parts of a program. This separation of concerns addresses the first form of heterogeneity mentioned above: a behaviour can be deployed using various communication media and data protocol combinations.

The deployment part can also make use of architectural primitives for handling the structure of an SOA. For instance, Jolie supports *embedding* and *aggregation*. Embedding deals with the structure of the execution contexts in which services operate, establishing a hierarchy of services. It allows a service to run another one as a sub-service. An embedder can communicate with an embedded service through an ordinary communication port; the behaviour abstracts completely from embedding, so if the programmer decides in the future not to embed a service and instead to refer to an external one, the behaviour does not need to be changed. Embedding has also some performance benefits. Aggregation, on the other hand, deals with the architecture of the connections between the services in an SOA. It allows for the creation of proxy services that can forward invocations to other services. Aggregation is purely related to deployment, since it takes only communication ports as parameters and creates bridges between them. The flexible aggregation and embedding mechanisms are examples of how Jolie addresses the second form of heterogeneity mentioned above. Remarkably, their design also elicits that the behavioural and architectural composition mechanisms can abstractly interact through the shared concept of communication ports.

Structure of the paper. § 2 presents the basic constructs of the language; § 3 shows how Jolie handles complex behavioural composition by supporting stateful sessions and error recovery; § 4 introduces architectural composition with embedding and aggregation; we show a practical example that uses our main composition primitives in § 5; § 6 reports conclusions, references to additional resources, related work, and future work.

2 Language basics: behaviour and deployment

A Jolie program defines an entity in an SOA (a service). Programs are run by the Jolie interpreter, and are usually stored inside files with the `.ol` extension[4]. A program is made by two parts, called *behaviour* and *deployment*.

[4] A Jolie program definition may even be retrieved from URLs or local memory.

The behavioural part defines the actions to be performed by the service, such as internal computations and input/output communications. This part abstracts from *how* communications will actually be supported. For example, a behavioural primitive may express the action "ask the calculator service to add the numbers 2 and 6 and then get a result back", without knowing exactly how to reach this calculator service (or which kind of communication protocol it uses).

The deployment part complements the behavioural part, introducing the necessary information for establishing communication links between services. It can also be used to define the structure of an SOA, as we will show later.

The structure of a Jolie *Program* is thus given by the following syntax:

$$Program \quad ::= \quad D \quad \texttt{main}\ \{\ B\ \}$$

where D represents the deployment part and B the behavioural part. The `main` procedure is the execution entry point.

2.1 Behaviours

The syntax for expressing service behaviours in Jolie combines the message-passing and the imperative programming styles. The former models composition of the behaviours of other services, whereas the latter enables internal computation. Table 1 reports a selection of the syntax for behaviours.

$B ::=$	η	*(input)*	$\eta ::=$	`o(x)`	*(one-way)*
$\mid$	$\overline{\eta}$	*(output)*	$\mid$	`o(x)(`e`){` B `}`	*(request-response)*
$\mid$	`if(` e `)` B_1 [`else` B_2]	*(cond)*			
$\mid$	`while(` e `)` B	*(while)*	$\overline{\eta} ::=$	`o@OP(`e`)`	*(notification)*
$\mid$	B `;` B'	*(seq)*	$\mid$	`o@OP(`e`)(y)`	*(solicit-response)*
$\mid$	$B \mid B'$	*(par)*			
$\mid$	`{` B `}`	*(block)*			
$\mid$	`x =` e	*(assign)*			
$\mid$	`nullProcess`	*(inact)*			
$\mid$	`[` η_1 `]` `{` B_1 `}` $\ldots$ `[` η_n `]` `{` B_n `}`	*(input choice)*			

Table 1. Jolie behavioural syntax (selected rules)

Communications. Rules *(input)*, *(output)*, and *(input choice)* implement communications. An input η can either be a one-way or a request-response. 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; the response value will be assigned to variable `y`. In the output statements, `OP` is an *output port name*. This name acts as a reference to an output port (cf. § 2.2) specified in the deployment definition D of the same service in which the behaviour is defined. Output port `OP` will contain the information (e.g., a URL) for contacting the target service. Finally, *(input choice)* implements input-guarded choice. Namely, it supports the receiving of a message for any of the operations in the inputs in the choice. When a message for an input η_i can be received, then all the other branches are deactivated and η_i is executed. Afterwards, the related branch behaviour B_i is also executed. A static check enforces all the η_i in an input choice to have different operations, so to avoid ambiguity.

Statement compositions. Rules *(cond)* and *(while)* implement the standard if-then-else choice and while iteration constructs. In *(cond)*, the `else` block is optional (denoted by its enclosure in square brackets). Rule *(seq)* enables the sequential composition of behaviours: B is executed, waited for termination, and then B' is executed. Rule *(par)* runs B and B' in parallel, i.e. in separate threads. The sequential operator `;` binds tighter than the parallel operator `|`. Operator precedence can be overridden using the *(block)* construct.

Assignments and empty behaviour. Rule *(assign)* evaluates expression e and assigns its value to variable `x`. Finally, term `nullProcess` is a no-op statement, denoting the empty behaviour.

Remark 1 (Sequence-Parallel interaction). Despite its C/Java-like syntax, it is interesting to observe that the constructs for behaviour composition in Jolie follow the workflow tradition. For instance, it is easy to program the fork-join pattern, as in `{`B_1`|`B_2`} ;` B_3, which is not natively supported, e.g., by Java. □

Example 1 (Store service). We give an implementation example of the behaviour of a store service. The service allows for retrieving information about a product (available quantity and price) and then placing an order for buying it.

```
getProductInfo( prod )( info ) {
  { getQuantity@Warehouse( prod )( q ) |
    getPrice@PriceList( prod )( price )
  }; info = "Price: " + price + "; Quantity: " + q
}; [ order( orderDesc ) ] { /* handle order */ }
   [ cancel() ] { nullProcess }
```

The behaviour starts with a request-response input on operation `getProductInfo`. When the input is invoked, its body is executed. First, it invokes services `Warehouse` and `PriceList` to retrieve the information about the product. Then, it concatenates a string using the retrieved information and stores it in variable `info`. After the body is executed, the original invoker of `getProductInfo` is sent the content of variable `info`. The behaviour now enters into a choice, waiting for an input for either operation `order` or `cancel`. If the first one is received, then the behaviour will handle the order of the invoker (we leave the handling code unspecified); instead, if `cancel` is invoked the behaviour simply terminates. □

Handling data. Jolie supports classic basic data types such as integers, strings, and booleans. More generally, variables and expressions can handle *structured data trees* using a concise and powerful syntax.

The variable state of a Jolie program is organised as a data tree. A variable then is simply a *path* for traversing the state and obtaining a subtree. Variables are dynamically allocated at runtime. It is easy to understand how this works by making a comparison to XML trees [5]. Consider the following behaviour:

```
x = 5 ; y = 10
```

Executing the code above would yield a state with two subnodes, `x` and `y`, respectively containing the integers 5 and 10. An XML representation would be:

```
<state> <x>5</x> <y>10</y> </state>
```

Executing now the statement `z = y / x` would yield the following state:

```
<state> <x>5</x> <y>10</y> <z>2</z> </state>
```

State traversing is obtained through the dot operator `.`, which can be used to specify paths. For instance, we can store information on a `person`:

Listing 1.1. A tree with personal information

```
person.name = "John"; person.age = 42;
person.contact.email = "john@smith.org";
person.contact.phoneNumber[0] = "123";
person.contact.phoneNumber[1] = "456"
```

The code above shows two features. The first is nesting: `email` is a subnode of `contact` which is a subnode of `person`. The second is vectors, obtained with the usual square bracket notation. An XML representation would be:

```
<state> <person> <name>John</name> <age>42</age> <contact>
      <email>john@smith.org</email>
      <phoneNumber>123</phoneNumber>
      <phoneNumber>456</phoneNumber>
</contact> </person> </state>
```

Jolie also comes with some native operators for manipulating data trees. In the following we show the deep copy operator `<<` and the vector size operator `#`. Assume that the following code is run with the state represented above:

```
x << person.contact ; numbers = #x.phoneNumber
```

In the resulting state `x` would then contain a copy of the tree pointed by `person.contact` and `numbers` the size of the vector `phoneNumber` inside that tree.

In the rest of the paper, we will simply refer to paths as variables.

[5] We observe, however, that Jolie trees are different from XML trees, as they are designed for performance. An example of a consequence is that Jolie tree nodes store typed values (strings, integers, . . .), whereas XML does not: all XML node values are strings, and their type is just an optional annotation.

2.2 Deploying services

We introduce now the syntax for deployment. The basic deployment primitives are *input ports* and *output ports*, which support input and output communications with other services. Ports are based on *interfaces* and *data types*.

A deployment D is simply a list of *deployment instructions* among which we can have input and output ports, type definitions, and interfaces:

$$D \quad ::= \quad D\ D \ \mid\ IP \ \mid\ OP \ \mid\ T_{\mathsf{def}} \ \mid\ I \ \mid\ \ldots$$

We leave this definition open with . . . as it will be extended in the next sections.

Communication ports. Communication ports define how communications with other services are actually performed. There are two kinds of ports. *Input ports* deal with exposing input operations to other services. *Output ports*, instead, define how to invoke the operations of 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*. The former two define the concrete binding information between a Jolie program and other services. The last, instead, defines type information that is expected to be satisfied by the behaviour that receives invocations through the port.

A location expresses the communication medium, along with its configuration parameters, a service uses for exposing its interface (in the case of an input port) or contacting another service (in the case of an output port). Examples of communication mediums are TCP/IP sockets, Unix sockets, Bluetooth communication channels, local memory channels, etc. A protocol defines how data to be sent or received should be, respectively, encoded or decoded following an isomorphism. Examples of protocols are SOAP, SODEP [8] (a binary protocol specifically developed for Jolie), HTTP forms, etc. Finally, a port must specify the interface that is accessible through it.

The syntax for input and output ports is:

```
IP ::= inputPort id {                      OP ::= outputPort id {
       Location: URI                              Location: URI
       Protocol: p                                Protocol: p
       Interfaces: iface_1, ..., iface_n          Interfaces: iface_1, ..., iface_n
       }                                          }
```

where URI is an URI (Uniform Resource Identifier), defining the location of the port; id, p, and iface_i are identifiers representing, respectively, the name of the port, the data protocol to use, and the interfaces accessible through the port.

The URI of a location must indicate the communication medium the port has to use and its related parameters, in this form: medium [:parameters] , where medium is a medium identifier and the optional parameters is a medium-specific string. Jolie currently supports four mediums: `btl2cap` (Bluetooth L2CAP), `localsocket` (Unix local sockets), `rmi` (Java RMI) and `socket` (TCP/IP sockets). An example of a valid location is: `"socket://www.mysite.com:80/"`, where `socket` is the medium and the following part represents the parameters.

Protocols are referred by name. Examples of valid protocol names are `http`, `https`, `soap`, `sodep`, and `xmlrpc`. The HTTP protocol can dynamically detect client invocations using different formats (e.g., GWT-RPC [4] and JSON [5]).

Data types and interfaces. Communication ports require interfaces to be defined. An interface is a collection of operation types. The latter define the data types of the values that can be communicated over each specified operation.

We start from data types. We remind that Jolie values are data trees. A data type specifies (i) the structure of a data tree, (ii) the type of the content of each node in a data tree, and (iii) the allowed number of occurences of each node (also called cardinality). Let us see an example first. We write a type for the data tree pointed by `person` in Listing 1.1.

```
type Person:void { .name:string .age:int
                   .contact[0,1]:void
                     { .email:string .phoneNumber*:string } }
```

A value of type `Person` must not contain anything in its root node (it is `void`). It *must* have the subnodes `firstName` (which must contain a `string`), `lastName`, `age` (an integer). It *may* have a subnode `contact` (this is specified by the notation `[0,1]`, to be read as "from zero to one occurences"). If it does, subnode `contact` must not contain anything in its root node (`void`), but it must have an `email` subnode and *any* number of `phoneNumber` subnodes (specified by the `*` notation).

The syntax for data types T_{def} is as follows:

$$
\begin{array}{rl}
T_{\mathsf{def}} & ::= \ \texttt{type}\ \mathsf{id}\ T \\
T & ::= \ \texttt{:}\ BT\ [\ \texttt{\{}\ .\mathsf{id}_1\ R_1\ T_1\ \ \ldots\ \ .\mathsf{id}_n\ R_n\ T_n\ \texttt{\}}\ ]\ |\ \texttt{undefined} \\
R & ::= \ \texttt{[}\ \mathsf{min}\ \texttt{,}\ \mathsf{max}\ \texttt{]}\ |\ \texttt{*}\ |\ \texttt{?} \qquad BT ::= \texttt{int}\ |\ \texttt{string}\ |\ \texttt{void}\ |\ \ldots
\end{array}
$$

Type definitions assign a type to a name id. Each type has definition T, comprehending a basic type BT and (optionally) a list of named subnode types or the `undefined` keyword, which makes the type accepting any subtree. Each subtype comes with a range R, defining the range of allowed number of occurences of the related subnode in a value. A range R can be an interval from min (an integer major or equal than zero) to max (an integer major or equal than its associated min), or `*`, meaning any number of occurences. `?` is a shortcut for `[0,1]`.

The syntax for interfaces I is:

$$
\begin{array}{l}
I ::= \texttt{interface}\ \mathsf{id}\ \texttt{\{}\ [\ \texttt{OneWay:}\ OW^+\ ]\ [\ \texttt{RequestResponse:}\ RR^+\ ]\ \texttt{\}} \\
OW ::= \mathsf{id}\texttt{(}\ OT\ \texttt{)} \qquad RR ::= \mathsf{id}\texttt{(}\ OT_{\mathsf{req}}\ \texttt{)(}\ OT_{\mathsf{resp}}\ \texttt{)} \qquad OP_T ::= BT\ |\ \mathsf{type}
\end{array}
$$

An interface I is a list of one-way and request-response operation declarations, respectively OW and RR. OW associates an operation identifier id to an operation type OT, which can be either directly a basic type BT or a reference to a user-defined type. RR is similar, but it distinguishes between the type for the request OT_{req} and the response OT_{resp}.

Remarkably, it is possible to define multiple input ports that expose the same interface through different communication technologies. This way, for example, a Jolie program may expose the same set of functionalities through a web interface and over Bluetooth, retaining simplicity in the behaviour.

Deployment introduces runtime type checking to behaviours. When a message is sent or received through a port, its type is checked against that specified for its operation in the related interface in the port. An invoker sending a message with a wrong type receives a TypeMismatch fault. Also, an output statement may throw the same fault when trying to send a message with wrong type.

2.3 Putting it all together

We can finally use the syntax shown so far to implement working Jolie programs, defining their behavioural and deployment parts. The following examples are complete, and therefore executable. The next listing defines a service that offers an operation for performing the summation of some numbers.

```
type SumRequest:void { .number[2,*]:int }
interface SumInterface
  { RequestResponse: sum(SumRequest)(int) }
inputPort SumInput { Location: "socket://localhost:8000/"
                     Protocol: soap Interfaces: SumInterface }
main {
  sum( request )( result ) {
    i = 0; while( i < #request.number )
             { result = result + request.number[i++] }
  }
}
```

The code above implements a service that exposes an operation sum that takes at least two number nodes in its request message and then replies with the sum of the numbers. The service is deployed accepting socket connections at TCP port 8000 and uses the soap protocol. Let us see a program that invokes the service above. Below, we use the include primitive for importing the output port Console from the Jolie standard library unit console.iol and print the result.

```
include "console.iol"
type SumRequest:void { .number[2,*]:int }
interface SumInterface
  { RequestResponse: sum(SumRequest)(int) }
outputPort SumServ { Location: "socket://localhost:8000/"
                     Protocol: soap Interfaces: SumInterface }
main {
  request.number[0] = 3; request.number[1] = 5;
  sum@SumServ( request )( response );
  println@Console( response )() /* will print 8 */ }
```

We can already see how the separation between behaviour and deployment helps in addressing the heterogeneity of communication technologies. For example, if we want to invoke our service from a web browser it is sufficient to change its communication protocol to http, without considering the behaviour:

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

Now we can sum numbers from a web browser by opening an URL such as:

```
http://localhost:8000/sum?number=10&number=2&number=4
```

which would display the response 16.

Remark 2 (Automatic Type Casting). In the example above, we passed some integers as parameters to our service through a *query string* in an HTTP URL, which does not carry data typing. In this case, Jolie is actually casting such string parameters to integers, referring to the operation type. Automatic type casting for untyped data also allows for rejecting immediately messages with a wrong type. For example, trying to access the service above with the URL:

```
http://localhost:8000/sum?number=wrong
```

would cause a `TypeMismatch` error to appear in the browser. □

3 Sessions and error recovery

Until now we have presented services that execute a behaviour and then terminate. We also never accounted for errors in the execution of behaviours. However, in service-oriented computing, services should be able to offer their functionalities multiple times and engage in *sessions*, i.e. stateful conversations with other entities with a shared goal. For example, a web browser, an E-Commerce service, and a bank service may engage in a session to perform a payment. Then, it would also need to handle possible errors in such an activity. In this section, we introduce the Jolie primitives for session programming and error recovery.

3.1 Behaviour instances

A service participates in a session by executing an instance of its behaviour. Until now, we have always executed behaviours only once, e.g. the sum service in the example in Section 2.3 supports a single session with a client for receiving some numbers and replying with their summation. After this session is executed, the service terminates and must be executed again manually if it is needed again.

Jolie allows to reuse a behavioural definition multiple times with the *execution modality* deployment primitive [30]:

$D ::= \ldots \quad | \quad$ `execution { ` M ` }` $\qquad M ::=$ `single` | `sequential` | `concurrent`

`single` is the default execution modality (so the `execution` construct may be omitted in this case) and causes the program behaviour to be executed only once. `sequential`, instead, causes the program behaviour to be made available again after the current instance has terminated. This is useful, for instance, for modelling services that need to guarantee exclusive access to a resource. Finally, `concurrent` causes a program behaviour to be instantiated and executed *whenever its first input statement can receive a message.* Jolie also supports special procedures for initialising a service before it makes its behaviour available, omitted here. The interested reader may refer to [31].

In the `sequential` and `concurrent` cases, Jolie enforces the behavioural definition inside the `main` procedure to be an input statement (an input η or an input choice, cf. Table 1). In such a case, we refer to the operations in this first input statement as *starting operation*.

Variable state. A crucial aspect of behaviour instances is that each instance has its own private state, determining variable scoping. This lifts programmers from worrying about race conditions in most cases. For instance, we could simply add the deployment instruction `execution { concurrent }` to the sum service in § 2.3 to make it supporting multiple clients at the same time. Access to variables would be safe since each behaviour instance would have its private state.

Jolie also provides *global variables* to support sharing of data among different behaviour instances. These can be accessed using the `global` prefix:

```
global.myGlobalVariable = 3; // Global variable
myLocalVariable = 1 // Local to this behaviour instance
```

Concurrent access to global variables can be restricted through the `synchronized` statement, similarly to Java: $B ::= \ldots \mid$ `synchronized ( id ) {` B `}` which ensures that only one process at a time will enter any `synchronized` block sharing the same id (which is a constant identifier).

Dynamic binding. Jolie allows output ports to be dynamically bound, i.e., their locations and protocols (called *binding information*) can change at runtime. Changes to the binding information of an output port is local to a behaviour instance: output ports are considered part of the local state of each instance. Dynamic binding is obtained by treating output ports as variables. For instance, the following would print the location and protocol name of output port `Printer`

```
include "console.iol"   include "Printer.iol"
outputPort Printer { Location: "socket://p:80/"
  Protocol: sodep    Interfaces: PrinterInterface }
main { println@Console( P.location )();
       println@Console( P.protocol )() }
```

where the file `Printer.iol` contains the interface:

```
interface PrinterInterface { OneWay: printText(string) }
```

Binding information may be entered at runtime by making simple assignments:

```
include "Printer.iol"
outputPort P { Interfaces: PrinterInterface }
main { P.location = "socket://p:80/"; P.protocol = "sodep" }
```

Example 2 (Binding registry). We show a usage example of dynamic binding and binding transmission by implementing a binding registry, i.e., a service that shares binding information. The registry offers a request-response operation, getBinding, that returns the binding information for contacting a service. We identify services by simple names. The interface of the registry is thus:

```
interface RIf { RequestResponse: getBinding(string)(Binding) }
```

where Binding is the type of port bindings defined in the standard Jolie library. Below we implement the registry behaviour, which supplies binding information for an inkjet printer and a laser printer (whose services we leave unspecified).

```
main {
  getBinding( name )( b ) {
    if ( name == "LaserPrinter" ) {
      b.location = "socket://p1.com:80/"; b.protocol = "sodep"
    } else if ( name == "InkJetPrinter" ) {
      b.location = "socket://p2.it:80/";  b.protocol = "soap"
    }
  }
}
```

Finally, we define a client that calls getBinding for discovering the laser printer:

```
outputPort Registry { /* omitted */ }
outputPort Printer  { Interfaces: PrinterInterface }
main { getBinding@Registry( "LaserPrinter" )( Printer );
       printText@Printer( "My text" ) }
```

3.2 Message routing with Correlation Sets

Having multiple instances of a behaviour running in a service introduces the problem of routing incoming messages to the right instances. Let us clarify with an example. Assume that an E-Commerce service has two behaviour instances opened for buying two products, respectively product A and product B. If a message for performing a payment comes from the network, how can we determine if the payment is for A or it is for B? Supposedly, we should require that the payment message contains some information that allows us to relate it to the correct behaviour instance, e.g., a serial number. In common web application frameworks this issue is covered by the *sid* session identifier, a unique key usually stored as a browser cookie.

Jolie supports incoming message routing to behaviour instances by means of *correlation sets* [32]. Correlation sets are a generalisation of session identifiers: instead of referring to a single variable for identifying behaviour instances, a correlation set allows the programmer to refer to the combination of multiple variables, called *correlation variables*. Correlation set programming in Jolie deals both with the deployment and behavioural parts. The former must declare the correlation sets to be used during execution, instructing the interpreter on how to relate incoming messages to internal behaviour instances. The latter instead has to assign the actual values to the correlation variables.

Correlation set declaration. Correlation sets are declared in the deployment part of a program using the following syntax:

$$D ::= \ldots \mid C \qquad C ::= \texttt{cset } \{\ C_{\mathsf{Var}}^{+}\ \} \qquad C_{\mathsf{Var}} ::= \texttt{x}\ \texttt{:}\ \mathsf{T_{path}}^{+}$$

A correlation set declaration C is a list of correlation variable declarations. A correlation variable declaration C_{Var} links a correlation variable `x` to a list of aliases. A correlation alias $\mathsf{T_{path}}$ is a path (using the same syntax for variable paths) starting with a message type name, indicating where the value for comparing the correlation variable can be retrieved in the message. Aliases are useful for ensuring a loose coupling between the names of the correlation variables and the data structures of the incoming messages.

The fact that correlation aliases are defined on message types makes correlation definitions statically strongly typed. The Jolie interpreter statically checks that each alias path points to a node in a message type that will surely be present in every incoming message of that type; technically, this means that the node itself and all its ancestor nodes are not optional in the type. For instance, the following is an invalid correlation set definition:

```
type MyType:void { .a:int { .b?:string } }
cset { myVar: MyType.a.b }
```

because `b` is an optional node in the type. In the sequel we refer to paths such as `a.b`, i.e. the path that follows after the type name, as the aliasing path for the correlation variable for the type (`MyType` above).

Jolie performs many other static checks for ensuring correctness of correlation set declarations (see [32]). Here we highlight that, for services using `sequential` or `concurrent` execution modalities, for each operation used in an input statement in the behaviour there is exactly one correlation set that links all its variables to the type of the operation. Since there is *exactly one* correlation set referring to an operation, we can unambiguosly call it the correlation set for the operation. We can now define how correlation works (see [32] for a formal definition).

> Let `o` be an operation and C be the correlation set for `o`. We say that an incoming message for `o` *correlates* with a behaviour instance if, for every variable `x` with `y` as aliasing path for the input type of `o` in C, we have that the value of `x` in the state of the behaviour instance is the same as the value of `y` in the message.

Whenever a service receives a message through an input port (and the operation in the message is correctly declared in an interface of the input port) there are three possibilities, defined below.

– The message correlates with a behaviour instance. In this case the message is received and given to the behaviour instance, which will be able to consume it through an input statement for the related operation of the message.
– The message does not correlate with any behaviour instance and its operation is a starting operation in the behavioural definition. In this case, a new behaviour instance is created and the message is assigned to it. If the starting operation has an associated correlation set, all the correlation variables in the correlation set are atomically assigned (from the values of the aliases in the message) to the behaviour instance before starting its executing.

– The message does not correlate with any behaviour instance of its operation is not a starting operation in the behavioural definition. In this case, the message is rejected and a `CorrelationError` fault is sent back to the invoker.

Correlation values. In the behavioural part of a program, correlation variables must be explicitly prefixed with the `csets` keyword. So, for instance, assigning the value `"MyValue"` to the correlation variable `myVar` above looks like:

```
csets.myVar = "MyValue"
```

It is often the case that the programmer needs to assign a *fresh* value to a correlation variable, to ensure that this value will be unique for each behaviour instance. Jolie provides the primitive `new` for this task, for example:

```
csets.myVar = new
```

We observe that a programmer can make mistakes when programming correlation. As an example, assume that in the following code snippet operation `close` (for closing a behaviour instance) has input type `CloseType`:

```
cset { x: CloseType.closeIdentifier }
main { open(); close() }
```

The code above is wrong because `x` is not instantiated before the input statement `close()`. Therefore, it is impossible for any message to correlate with that input and let the behaviour instance terminate. Jolie comes with a static checker that can detect some common problems in correlation programming [32]. Here we highlight that among the properties guaranteed by the static checker is the fact that correlation is always deterministic: whenever a message is received by a service there is at most one behaviour instance that can correlate the message.

Example 3 (Distributed authentication). We report an example from [32] inspired by the OpenID Authentication specifications [35], a largely adopted decentralised Single Sign-On protocol that allows a service, called *relying party*, to authenticate a user, the *client*, by relying on another external service that is responsible for handling identities, the *identity provider*. Therefore, OpenID specifies a *multiparty session*. When the client requests access to the relying party, the latter starts an authentication session with the identity provider and redirects the client to it. The client then sends its authentication credentials to the identity provider, which will inform the relying party on the result of the authentication attempt. The example can be downloaded at [7]. Here, we show an implementation sketch for the relying party.

```
cset { clientToken: /* ... */ }
cset { secureToken: AuthMessage.secureToken }
interface RelyingPartyInterface {
OneWay: authSucceeded(AuthMessage), authFailed(AuthMessage)
RequestResponse: login(LoginRequest)(Redirection) }
main {
  login( loginRequest )( redirection ) {
    openRequest.clientToken = csets.clientToken = new;
    openRequest.secureToken = csets.secureToken = new;
    openRequest.relyingPartyIdentifier = MY_IDENTIFIER;
    openAuth@IdentityProvider( openRequest );
    /* ... build redirection message for client ... */
  };  [ authSucceeded( message ) ] { /* ... */ }
      [ authFailed( message ) ]    { /* ... */ }
}
```

The service receives a request on the starting operation `login` from the client for initiating the protocol. The body of `login` generates two fresh correlation tokens, `clientToken` and `secureToken`, and also stores them under the `openRequest` variable. We will use `clientToken` for receiving messages from the client and `secureToken` for receiving messages from the identity provider. The client is not informed about `secureToken`, preventing it to maliciously act as the identity provider. The body of `login` performs a call to the identity provider, starting an authentication session and communicating `secureToken`. The reply will redirect the client to the identity provider. The relying party will then wait for a notification about the result of the authentication attempt, hence the input choice on `authSucceeded` and `authFailed`, which correlate through `secureToken`. □

3.3 Fault handling

Fault handling in Jolie involves four basic concepts: *scope*, *fault*, *termination* and *compensation*. We now describe the first three concepts: the reader interested in compensation handling can refer to [17]. A scope is a behaviour container denoted by a unique name and able to manage faults. A fault is a signal raised by a behaviour towards the enclosing scope when an error state is reached, in order to allow for its recovery. Termination is a mechanism exploited to recover from errors: termination is automatically triggered when a scope is unexpectedly terminated from a parallel behaviour and must be smoothly stopped. We say that a scope terminates successfully if it does not raise any fault signal; this is obtained by the scope by handling all the faults thrown by its internal behaviour. Recovery mechanisms are implemented by exploiting *handlers*, which contain the recovery code to be executed when faults or terminations are triggered.

We extend the syntax of behaviours with the primitives for fault handling:

$$
\begin{array}{rll}
B ::= \dots & |\ \texttt{scope(s) \{ } B \texttt{ \}} & \textit{(scope)} \\
 & |\ \texttt{install(} \mathsf{h}_1 \texttt{ => } B_1 \texttt{ , ... , } \mathsf{h}_n \texttt{ => } B_n \texttt{)} & \textit{(inst)} \\
 & |\ \texttt{cH} & \textit{(cH)} \\
 & |\ \texttt{throw(f [, x])} & \textit{(throw)}
\end{array}
$$

Above, *(scope)* defines a scope with a unique scope name `s` and a behaviour B. *(inst)* dynamically installs the handlers B_i for their respective names h_i in the enclosing scope, where `h` can be either a fault name or one of the special keywords `this` and `default`. If it is a fault name, then the handler is installed as a fault handler; if it is `this`, then the handler is installed as a termination handler for the enclosing scope; if, finally, it is `default`, then the handler is installed as a generic fallback fault handler for all faults that do not have a specific fault handler. Installing a handler overwrites the previous one for the same fault or scope name; however, handlers can be composed by using the `cH` placeholder, which is replaced by the code of the previously installed handler. Finally, *(throw)* throws a fault f with some optional data referred by variable `x`.

Automatic fault transmission. Faults signals thrown from inside a request-response statement body that reach the input statement are automatically transmitted to the invoker. This ensures that invokers are always notified of unhandled faults. We update the syntax for request-response operation types (cf. § 2.2) to declare the faults f_i that could be sent back to invokers with data of type OT_i:

$$RR ::= \mathsf{id}(\ OT_{\mathsf{req}}\ )(\ OT_{\mathsf{resp}}\ )\ [\ \texttt{throws}\ \mathsf{f}_1(\ OT_1\ )\ \ \ldots\ \ \mathsf{f}_n(\ OT_n\ )\ ]$$

It follows from the fact that request-response operations may return a fault, that now the solicit-response output statement may throw the received fault.

Handler composition. The `cH` element allows for the dynamic composition of behavioural code. Consider the following example:

```
scope(s) { install( f => i = i+2 ); install( f => i++; cH ) }
```

In the second `install` a handler using `cH` is installed for fault `f`. At runtime, `cH` will be replaced with the previously installed handler. So the second install instruction is actually equivalent to:

```
install( f => i++; i = i + 2 )
```

Install statement priority. An install statement may execute in parallel to other behaviours that may throw a fault. This introduces a problem of nondeterminism: how can the programmer ensure that the correct handlers are installed regardless of the scheduling of the parallel activities? Jolie solves this issue by giving priority to the install primitive w.r.t. fault processing, making handler installation predictable. As an example, consider the following code:

```
scope(s) { throw(f) | install(f => println@Console("Hi")()) }
```

where, inside the scope `s`, we have a parallel composition of a `throw` statement for fault `f` and an installation of a handler for the same fault. The priority given to the `install` primitive guarantees that the handler will be installed before the fault signal for `f` reaches the scope construct and its handler is searched for.

4 Architectural composition

Until now we have shown how a behaviour can compose other behaviours abstracting from its deployment. In this section we show how composition can be obtained from the opposite perspective. Namely, we present *architectural composition*, a different kind of composition that a deployment definition can obtain abstracting from the specific behavioural definitions of the involved services.

Architectural composition can be roughly divided in two main categories. The first deals with the structuring of the execution contexts in which services operate. For instance, a service may execute other sub-services in the same execution engine, in order to gain advantages in terms of resource control. Other examples can be the *wrapping* and *hiding* of an entity in an SOA. The second category deals with the topology of the connections between services in an SOA. Jolie supports mechanisms for both categories [19, 31]. Here we introduce two representatives, respectively *embedding* [31] and *aggregation* [31, 15].

4.1 Embedding

Embedding is a mechanism for executing multiple services in the same virtual machine. We say that a service *embeds* another service when the former, called *embedder*, uses the `embedded` primitive targeting the latter, called *embedded* service. The syntax for embedding is:

$$\begin{aligned} D &::= \ldots \quad | \quad E \\ E &::= \texttt{embedded \{ } E_{\mathsf{type}} \texttt{ : } \mathsf{path} \ [\ \texttt{in OP}\ ] \texttt{ \}} \\ E_{\mathsf{type}} &::= \texttt{Jolie} \mid \texttt{Java} \mid \texttt{JavaScript} \end{aligned}$$

where E is the embedding construct, E_{type} specifies the type (technology) of the service to embed, and path is an URL (possibly in simple form) pointing to the definition of the service to embed. Jolie currently supports the embedding of Jolie, Java, and JavaScript service definitions; this support can be modularly extended [31]. Embedding may optionally specify an output port `OP`; in this case, as soon as the service is loaded, the output port `OP` is bound to the "local" communication input port of the embedded service. The meaning of local communication input port is dependent on the embedding type; we will show examples for Jolie and Java services. This makes embedding a *cross-technology* mechanism: it can load services defined using different languages. Embedding produces a hierarchy of services where the embedder is the parent service of the embedded ones; this hierarchy handles recursive termination, i.e., whenever a service terminates all its embedded services are recursively terminated. The hierarchy is also useful for enhancing performance: services in the same virtual machines may communicate using fast local memory communication channels.

When embedding a Jolie service, the path URL must point to a file containing a Jolie program provided as source code or a binary form produced by the `joliec` tool (not reported here). Command line parameters can also be passed. Local in-memory communication between embedder and embedded is enabled by means of the `local` communication medium, which must be specified by the embedded service. In this case no protocol definition is needed.

Example 4 (Embedded Jolie *service).* We embed the sum service from § 2.3. First, we add the following input port to allow for local communications:

```
inputPort LocalInput
  { Location: "local" Interfaces: SumInterface }
```

Now we can design a modified version of the client program in § 2.3 to embed the sum service (whose definition we assume to be stored in file `sum_service.ol`) and call it using an output port bound by embedding. We omit interfaces.

```
outputPort SumService { Interfaces: SumInterface }
embedded { Jolie: "sum_service.ol" in SumService }
main {
  request.number[0] = 3; request.number[1] = 5;
  sum@SumService( request )( response )
}
```

□

When embedding a Java service, the path URL must unambiguously identify a Java class, which can also be in the current Java classpath of the Jolie interpreter. The class is expected to extend the `JavaService` abstract class, offered by the Jolie Java library for supporting automatic conversion between Java values and their Jolie representations. Each method of the Java service class is seen as an operation from the embedder, which will instantiate an object using the class and bind it to the output port. Embedding Java services is particularly useful for reusing some existing Java code, perform some task where computational performance is important, or interoperating with some existing legacy software. Many services of the Jolie standard library (like `Console`) are Java services.

Example 5 (Java service embedding). We embed a simple Java service that offers a `length` Request-Response operation that takes a string as request and replies with the length of the string. Consider the following Java code:

```
package example; import jolie.runtime.JavaService;
public class MyService extends JavaService {
  public Integer length( String request )
    { return request.length(); }
}
```

We can embed and use the code above from a Jolie program such as the following:

```
interface MyServiceInterface
  { RequestResponse: length(string)(int) }
outputPort MyService { Interfaces: MyServiceInterface }
embedded { Java: "example.MyService" in MyService }
main { length@MyService( "Hi" )( l ) }
```

□

We end our presentation of embedding by showing how to use it at runtime, i.e., embedding can be *dynamic*. Dynamic embedding can be used to implement features such as code mobility (an important feature for implementing cloud computing middleware) and service adaptation.

Example 6 (Platform-as-a-service). We implement a simple platform-as-a-service solution. We offer customers the possibility to execute services by means of service mobility. Each customer has a certain amount of allowed time: a loaded service can not execute for more time than what the customer is allowed for, and when the service terminates the allowed time gets proportionally decreased. We implement our program using the `MetaService` service, from the Jolie standard library, which allows to dynamically load and unload services respectively through the `loadEmbeddedService` and `unloadEmbeddedService` operations. We omit some code; a more complete version of this example can be found in [31].

```
execution { concurrent } csets { sid: /* ... */ }
main {
  login( l )( csets.sid )
    { auth@AccountManager( l )( account ); csets.sid = new };
  startService( s )() { loadEmbeddedService@MetaService( s )();
                    setNextTimeout@Time( account.allowedTime ) };
  [ timeout() ] { nullProcess }
  [ stopService( sid ) ] { nullProcess };
  { unloadEmbeddedService@MetaService( s.resourceName )() |
    updateAllowedTime@AccountManager( account )() }
}
```

The service supports multiple sessions, using the `concurrent` modality. First, the customer is required to login, thus creating a behaviour instance. An (omitted) `AccountManager` service is composed for handling customer accounts; if `auth` fails, we rely on automatic fault transmission (cf. § 3.3) to send the fault to the customer through `login`. After the customer has been successfully authenticated, a fresh session identifier `sid` is sent to the customer. The session id is assumed to be in the service correlation set, so the customer can use it to refer to the created behaviour instance. The `startService` operation is then made available, which can be called in order to start a new service; the latter is loaded by composing `MetaService`. When the service is successfully embedded, a `Time` service is used in order to handle a timeout that is set to the allowed time of the customer. This timeout is used in the following input choice, where either the timeout occurs or the `stopService` operation gets called first. In both cases the service gets unloaded and, concurrently, the account allowed time gets updated. □

4.2 Aggregation

Aggregation is a generalisation of network proxies that allows a service to expose operations without implementing them in its behaviour, but instead delegating them to other services. Aggregation can also be used for programming various architectural patterns – such as load balancers, reverse proxies, and adapters – omitted here (see [15]). The syntax for aggregation extends that for input ports:

$$IP ::= \texttt{inputPort}\ \mathsf{id}\ \{\ \texttt{Location:}\ URI\ \texttt{Protocol:}\ \mathsf{p}\ \texttt{Interfaces:}\ \mathsf{iface}_1, \ldots, \mathsf{iface}_n \\ [\texttt{Aggregates:}\ \mathtt{OP}^+]\ \}$$

by introducing the `Aggregates` primitive, which expects a list of output port names. The interfaces of the aggregated output ports must not share any operation name. We can now define how aggregation works. Let IP be an input port aggregating some output ports. Whenever a message for operation `o` is received through IP we have the three following possibilities.

– `o` is an operation declared in one of the interfaces of IP. In this case, the message is normally received by the program as described in § 3.2.
– `o` is not declared in one of the interfaces of IP and is declared in the interface of an output port OP aggregated by IP. In this case, the message is forwarded to OP as an output from the aggregator.
– `o` is not declared in any interface of IP or of its aggregated output ports. Then, the message is rejected and an `IOException` fault is sent to the caller.

From the second item above, we can observe that aggregation *merges* the interfaces of the aggregated output ports and makes them accessible through a single input port. Thus, an invoker would only see the whole aggregated set of services as a single one. Aggregation is also useful because of its implicit interplay with communication abstraction, giving the possibility to expose services using one communication technology with a different one.

Remarkably, aggregation handles the request-response pattern seamlessly: when forwarding a request-response invocation to an aggregated service, the aggregator will automatically also take care of relaying the response to the initial original invoker.

Example 7 (Forwarder). Aggregation can be used for system integration, e.g. by bridging services that speak different protocols (see [31, 15] for more integration patterns using aggregation). The following deployment snippet creates a service that forwards incoming SODEP calls on TCP port 8000 to the output port `MyOP`, taking care of converting the received messages to SOAP.

```
outputPort MyOP    { Location: "socket://someurl.ex:80/"
                     Protocol: soap Interfaces: MyIface }
inputPort MyInput { Location: "socket://localhost:8000/"
                     Protocol: sodep Aggregates: MyOP }
```

□

Example 8 (Aggregation and embedding). We give an example where three services – A, B, and C – are aggregated by a service M, which also embeds C. The code follows, where we have an output port for each service with the same name:

```
outputPort A { Location: "socket://someurlA.com:80/"
               Protocol: soap Interfaces: InterfaceA }
outputPort B { Location: "socket://someurlC.com:80/"
               Protocol: xmlrpc Interfaces: InterfaceB }
outputPort C { Interfaces: InterfaceC }
embedded     { Java: "example.serviceB" in B }
inputPort  M { Location: "socket://urlM.com:8000/"
               Protocol: sodep Aggregates: A, B, C }
```

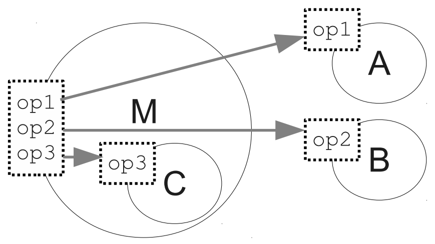

Fig. 1. An aggregator M that exposes the union of all the interfaces of the services it aggregates (A, B, C). Service C executes inside the virtual machine of M, by embedding. Interfaces are here represented with dotted rectangles.

Observe that the code for aggregating service C abstracts from the fact that the service is actually embedded and not external; this abstraction is given by using output ports for aggregating, creating a dependency only on the interface of the port instead of the implementation and location of the target service. The obtained architecture is graphically represented in Figure 1. In there, we assume that each service exposes just one operation. The gray arrows represent how the messages will be forwarded. For instance, an incoming message for operation `op3` will be forwarded to the embedded service C. □

5 Example: an automotive case study

We present a Jolie implementation of the automotive case study in the EU project SENSORIA [36]. We describe the main aspects of the implementation. A complete description and executable source code can be found at [1].

In the automotive case study a car experiments a failure during a travel. An onboard computer is responsible for helping the driver in finding and booking the necessary services for handling the situation: a garage for receiving the car, a tow truck for towing the car to the garage, and a car rental for renting a car as temporary replacement. We describe the execution flow of the system step by step. All entities are coded in Jolie, unless otherwise stated.

Getting assistance after a failure. When the Jolie program running in the car onboard computer (called *car service*) detects a failure, it sends the failure description to the *assistance service* of the car manufacturer. The latter analyses the description and sends back to the car service a Jolie program, called *local assistant*, that is specific for the kind of failure. The car service now dynamically embeds the local assistant (similarly to Example 6), and starts to interact with it. Both the car service and the local assistant implement predefined static interfaces, which specify the operations that are expected to be respectively implemented in order to be able to interact. The mechanism is depicted in Fig. 2.

Local assistant behaviour. The behaviour of the local assistant depends on the kind of failure. For instance, we distinguish between failures that make the car unable to move or not. Here we describe only the case in which the car is

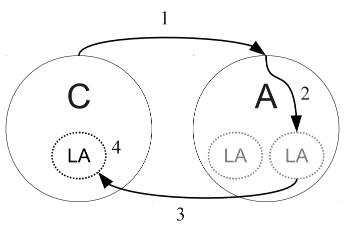

Fig. 2. *(Local assistant retrieval).* The car service C calls the assistance service A (1), which selects the appropriate local assistance code LA (2) and sends it back to C (3). C can now dynamically embed and run LA (4).

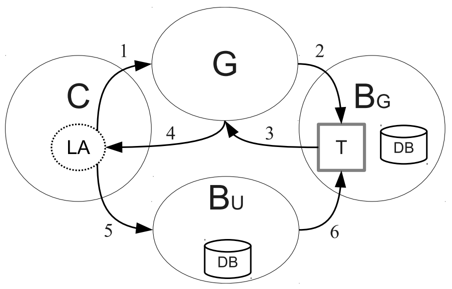

Fig. 3. *(Payment workflow).* The local assistant LA inside the car service C calls the garage G (1), which opens a bank transaction T inside its bank B_G (2) and obtains a transaction identifier (3), forwarded to LA (4). LA then asks C to send the identifier to the user's bank B_U (5), which finally closes the money transfer correlating with T (6).

unable to move, where we need to find a garage, a tow truck, and a car rental to successfully handle the situation. First, the local assistant asks the car service for the GPS coordinates of the car. The car service actually aggregates a secondary service, the *sensors service*, for making some read-only instrumentation data transparently available to the local assistant. Then, it will use such information for building an ordered list of suitable garage, tow truck, and car rental services, which are dynamically discovered through a registry provided on the public network. The assistant asks now the car service to display the list to the user, which also contains price information. The latter does so by using a Java User Interface, embedded as a java service. The assistant is then notified of the user's selection, and is now responsible for booking the selected services.

Bookings and payments. The local assistant has now to book and pay for the selected garage, tow truck, and car rental services. For each service, we perform the booking and then the payment. Here we exploit dynamic fault handling for elegantly adapting our error recovery strategy based on the reached point of execution. Consider for instance the following (simplified) code sketch:

```
scope( s ) {
  book@Garage( gb )( gr );
  install( default => cancelBook@Garage( gr ) );
  pay@Bank( gr )( gp );
  install( default => cH | cancelPay@Bank( gp ) );
  book@TowTruck( tb )( tr );
  install( default => cH | cancelBook@TowTruck( tr ) );
  /* ... */
}
```

Above, scope `s` takes care of the bookings and payments. Whenever one of those is successfully carried out, we update the fault handler for the scope by adding the code for reverting it (in parallel, for efficiency). So, for example, if the booking of the tow truck fails we would revert both booking and payment for the garage.

Bank transactions. The example comes with different bank services that could be involved in the money exchange. Here we describe the case for garage payment, depicted in Fig. 3. Let us call $\mathsf{B_U}$ the bank service handling the user's bank account and $\mathsf{B_G}$ the bank service handling the bank account of the garage. When the local assistant books the garage, the garage opens a behaviour instance in $\mathsf{B_G}$ for handling the bank transaction, gets back a transaction identifier (which is a correlation value for the behaviour instance in $\mathsf{B_G}$) and returns it to the assistant. The assistant now asks the car service to perform the payment. We chose to delegate payment to the car service to ensure that sensitive data such as credit card information is always managed only by the software that was already running in the car. In order to perform the payment, the car service contacts $\mathsf{B_U}$ passing the user's account data, the binding information to $\mathsf{B_G}$, and the transaction identifier. $\mathsf{B_U}$ can now close the transaction by contacting $\mathsf{B_G}$, using the binding information for reaching its input port and the transaction identifier for correlating with the right behaviour instance. All the provided bank implementations use persistent databases using the Jolie standard library for SQL-based DBMS, which is implemented with an embedded Java service.

6 Conclusions

We have introduced Jolie, a programming language that synthesizes a coherent programming paradigm from the technologies and practices that emerged in service-oriented computing in the recent years. It deals with both the heterogeneity of communication technologies and that of composition mechanisms. We addressed the former by separating the behavioural and deployment definitions of Jolie programs and reducing their coupling to communication ports and their respective interfaces. We covered composition mechanisms by offering both behavioural composition primitives for managing complex workflows and more high-level architectural primitives that act on the topology of systems.

Jolie comes with formal specifications (in terms of a process calculus) of its semantics, omitted in this paper [18, 31]. This formal approach has been instrumental for reasoning on the underlying model of many constructs of the language. For instance, correlation sets and their properties are formalised in [32]. Dynamic fault handling has been developed purposefully for Jolie; its formalisation is reported in [17]. A formal account of aggregation can be found in [15].

Jolie has also been a source of inspiration for other work. For example, dynamic fault handling has been proven to be more expressive than classic static fault handling [27]. [33] reports some programming patterns for component-based systems that can be implemented in Jolie [31]. [19] presents some engineering concepts that have been generalised from practical experience in Jolie programming.

Applications. The design of Jolie has been validated (and influenced) by covering a broad spectrum of applications, from low-level software tightly combined to hardware to more high-level systems such as web portals and dynamic SOAs.

Jorba [25] is a framework for supporting the dynamic adaptation of distributed applications wrt their execution context; it is based on the usage of dynamic embedding for implementing code mobility. Leonardo [29] is a Web Server for Web 2.0 applications written in pure Jolie. Vision [38] is a peer-to-peer application for sharing live presentations and synchronising presentation viewers. [9] presents a distributed architecture for the management of virtual machines written in Jolie. Central Watcher is a software for managing and monitoring phone centrals. Marco Polo is a web catalogue that integrates a DBMS-based backend with a web and smartphone frontends. Central Watcher and Marco Polo are proprietary products of italianaSoftware s.r.l. [21], a software development company using Jolie as the reference language.

Tool support. Jolie comes with many supporting tools (see [23]) . Examples are: joliedoc, a documentation generator similar to javadoc; jolie2dummy, a tool for the quick initial prototyping of executable Jolie code with "dummy" data starting from an interface; jolie2java, a converter from Jolie data types to Java class definitions; jolie2wsdl, a tool that generates a WSDL [43] document describing a Jolie service from an interface; viceversa, wsdl2jolie generates a Jolie output port and interface for calling an external Web Service from its WSDL descriptor. Joliepse is a prototype IDE for Jolie built on Eclipse [16]. jEye [22] is a graphical editor for Jolie programs. Finally, QtJolie is a C++ library for communicating with Jolie services, developed in the scope of the KDE project [6].

Related work. Related work falls into the categories of orchestration languages and integration middleware for SOAs.

WS-BPEL [34] is the reference language for orchestration in the context of Web Services. Some concepts present in Jolie have been inspired by WS-BPEL and WSDL. For instance, the basic uni- and bi-directional operations considered in Jolie are essentially WSDL operations: this was an initial design choice to support the possibility to easily expose Jolie interfaces in WSDL format. Also basic constructs like communication ports, correlation sets, fault and compensation handling were already present in WS-BPEL. Nevertheless, Jolie significantly extends these concepts. For instance, (i) in Jolie communication ports can be dynamically rebound with a simple assignment operation, (ii) we have developed static analysis techniques to provide guarantees on the correlation based message delivery mechanism, and (iii) in Jolie we consider dynamic handler installation to guarantee the execution of the right fault recovery policy. Another significant difference is that Jolie uses a programmer friendly C/Java-like syntax instead of the XML-based syntax of WS-BPEL. Some other orchestration languages in the literature come equipped with a formal semantics. An example is Blite [28] which was proposed as a solution to some WS-BPEL ambiguities: Blite captures a subset of WS-BPEL and equips it with a formally defined operational semantics. Differently from Jolie, Blite does not have its own interpreter: Blite programs are translated into WS-BPEL and can be executed by WS-BPEL engines. HomeBPEL [10] is an extension of WS-BPEL for handling the mobility of behaviour instances. The main difference wrt Jolie is that Jolie does not support instance mobility, but only stateless service mobility through dynamic embedding (state mobility can be obtained, but it must be coded manually by the programmer). This means that it is easy PiDuce [12] is a distributed implementation of pi-like process calculi with native XML datatypes. One of the distinguishing features of PiDuce is its powerful pattern mechanism used to deconstruct XML documents. Finally, Orc [24] is an orchestration language which follows a data-flow oriented approach. Orc programs are based on function invocations and three composition operators: parallel, sequential and pruning. The sequential composition operator forwards the data produced by an Orc subprogram to another subprogram, while pruning is used to forward only the first datum and discard the subsequent ones. Pruning elegantly captures the renowned "speculative parallelism" composition pattern, which invokes several services in parallel and considers only the first reply.

We now move to integration middlewares for SOAs. In this context the Enterprise Application Integration (EAI) framework [37] is often used, along with a collection of other technologies and services. There are several mature EAI technologies on the market, developed by leading IT companies such as IBM, Oracle, and Microsoft. Usually these are implemented by enhancing standard middleware products such as an Enterprise Service Bus (ESB) [14] via an event-driven messaging engine. These solutions cover a similar role to that of aggregation, as reported in [15] (where a more powerful version of aggregation is also presented). Differently from our approach, however, all these tools are specific to some application domain (e.g., Web Services) and are thus less general.

Future Work. We plan to implement a type system for dynamic binding so that the passed bindings guarantee that the target service implements a given interface. A similar extension is planned also for dynamic embedding. Another future work is to develop a static analysis for verifying the absence of "dangling bindings", i.e., a service should never use an output port that points to a location that is not covered by a respective input port of some service.

We will investigate how Jolie can be combined with techniques for the specification of protocols such as those based on session types, contracts, and choreographies [20, 13, 26]. Our aim is to produce tools for supporting the verification and sound implementation of SOAs wrt global descriptions of system behaviour. The high grade of granularity that embedding introduces makes it interesting to consider relations such as that presented in [11], where the authors present a notion of conformance between choreography and orchestration in which a choreography role can be mapped to multiple orchestrators. This is relevant because in practice it is often the case that a Jolie service makes use of several small sub-services in order to function. More generally, it would be interesting to observe how the architectural primitives of Jolie may influence the design of global specification languages, in which architectures are usually very simple.

References

1. Automotive example. `http://www.jolie-lang.org/files/ws_handbook2012/automotive.zip`.

2. D-Bus website. `http://www.freedesktop.org/wiki/Software/dbus/`.
3. GNOME. `http://www.gnome.org/`.
4. Google Web Toolkit. `http://code.google.com/webtoolkit/`.
5. JavaScript Object Notation. `http://www.json.org/`.
6. K Desktop Environment. `http://www.kde.org/`.
7. OpenID implementation. `http://www.jolie-lang.org/files/ws_handbook2012/openid.zip`.
8. SODEP: Simple Operation Data Exchange Protocol. `http://www.jolie-lang.org/wiki.php?page=Sodep`.
9. P. Anedda, M. Gaggero, S. Manca, O. Schiaratura, S. Leo, F. Montesi, and G. Zanetti. A general service oriented approach for managing virtual machines allocation. In *Proceedings of ACM Symposium on Applied Computing (SAC), 2009*, pages 2154–2161, 2009.
10. Mikkel Bundgaard, Arne J. Glenstrup, Thomas T. Hildebrandt, Espen Højsgaard, and Henning Niss. Formalizing Higher-Order Mobile Embedded Business Processes with Binding Bigraphs. In *Proceedings of COORDINATION 2008*, pages 83–99, 2008.
11. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and Orchestration conformance for system design. In *Proceedings of 8th International Conference on Coordination Models and Languages (COORDINATION 2006)*, volume 4038 of *Lecture Notes in Computer Science*, pages 63–81, 2006.
12. Samuele Carpineti, Cosimo Laneve, and Luca Padovani. Piduce - a project for experimenting web services technologies. *Sci. Comput. Program.*, 74(10):777–811, 2009.
13. Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for Web services. *ACM Trans. Program. Lang. Syst.*, 31(5), 2009.
14. David A. Chappell. *Enterprise Service Bus - Theory in practice*. O'Reilly, 2004.
15. M Dalla Preda, M Gabbrielli, C Guidi, J Mauro, and F Montesi. A language for (smart) service aggregation: Theory and practice of interface-based service composition. Technical Report UBLCS-2011-11, University of Bologna, 2011.
16. The Eclipse Foundation. Eclipse. `http://www.eclipse.org/`.
17. C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro. Dynamic Error Handling in Service Oriented Applications. *Fundamenta Informaticae*, 95(1):73–102, 2009.
18. C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: A Calculus for Service Oriented Computing. In *Proc. of ICSOC 2006*, pages 327–338, 2006.
19. Claudio Guidi and Fabrizio Montesi. Reasoning About a Service-oriented Programming Paradigm. In *Proceedings of YR-SOC 2009*, pages 67–81, 2009.
20. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proc. of POPL'08*, volume 43(1), pages 273–284. ACM Press, 2008.
21. italianaSoftware s.r.l. italianaSoftware. `http://www.italianasoftware.com/`.
22. jEye. jEye: a graphical designer for Jolie. `http://sourceforge.net/projects/jeye/`.
23. Jolie. JOLIE: Java Orchestration Language Interpreter Engine. `http://www.jolie-lang.org/`.
24. David Kitchin, Adrian Quark, William R. Cook, and Jayadev Misra. The Orc programming language. In David Lee, Antónia Lopes, and Arnd Poetzsch-Heffter, editors, *Proceedings of FMOODS/FORTE 2009*, volume 5522 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2009.
25. Ivan Lanese, Antonio Bucchiarone, and Fabrizio Montesi. A Framework for Rule-based Dynamic Adaptation. In *Proceedings of TGC 2010, 5th International Symposium on Trustworty Global Computing*, Lecture Notes in Computer Science. SV, 2010.

26. Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. Bridging the Gap between Interaction- and Process-Oriented Choreographies. In *Proceedings of Sixth IEEE International Conferences on Software Engineering and Formal Methods (SEFM 2008)*, pages 323–332. IEEE Computer Society, 2008.
27. Ivan Lanese, Catia Vaz, and Carla Ferreira. On the Expressive Power of Primitives for Compensation Handling. In *Proceedings of ESOP 2010*, Lecture Notes in Computer Science. SV, 2009.
28. Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. Using formal methods to develop ws-bpel applications. *Sci. Comput. Program.*, 77(3):189–213, 2012.
29. Leonardo. Leonardo Web Server. `http://www.sourceforge.net/projects/leonardo/`.
30. F. Montesi, C. Guidi, and G. Zavattaro. Composing Services with JOLIE. In *Proceedings of ECOWS 2007*, pages 13–22, 2007.
31. Fabrizio Montesi. Jolie: a Service-oriented Programming Language. Master's thesis, University of Bologna, Department of Computer Science, 2010.
32. Fabrizio Montesi and Marco Carbone. Programming services with correlation sets. In *ICSOC*, pages 125–141, 2011.
33. Fabrizio Montesi and Davide Sangiorgi. A model of evolvable components. In *Proceedings of Fifth Symposium on Trustworthy Global Computing (TGC 2010)*, 2010.
34. OASIS. Web Services Business Process Execution Language Version 2.0. `http://docs.oasis-open.org/wsbpel/`.
35. OpenID. OpenID Specifications. `http://openid.net/developers/specs/`.
36. SENSORIA. Software Engineering for Service-Oriented Overlay Computers. `http://www.sensoria-ist.eu/`.
37. Mostafa Hashem Sherif. *Handbook of Enterprise Integration*. Auerbach Publishers, Incorporated, 2009.
38. Vision. Vision: a distributed presentation framework. `https://jolie.svn.sourceforge.net/svnroot/jolie/trunk/playground/vision/`.
39. World Wide Web Consortium (W3C). Extensible Markup Language (XML). `http://www.w3.org/XML/`.
40. World Wide Web Consortium (W3C). HTTP - Hypertext Transfer Protocol. `http://www.w3.org/Protocols/`.
41. World Wide Web Consortium (W3C). SOAP Specifications. `http://www.w3.org/TR/soap/`.
42. World Wide Web Consortium (W3C). Web Services Architecture. `http://www.w3.org/TR/ws-arch/`.
43. World Wide Web Consortium (W3C). Web Services Description Language. `http://www.w3.org/TR/wsdl`.