Composing services with JOLIE*

Fabrizio Montesi Claudio Guidi
Gianluigi Zavattaro
Department of Computer Science, University of Bologna, Italy
{fmontesi,cguidi,zavattar}@cs.unibo.it

Abstract

Service composition and service statefulness are key concepts in Web Service system programming. In this paper we present JOLIE, which is the full implementation of our formal calculus for service orchestration called SOCK. JOLIE inherits all the formal semantics of SOCK and provides a C-like syntax which allows the programmer to design the service behaviour and the service deployment information separately. The service behaviour is exploited to design the interaction workflow and the computational functionalities of the service, whereas the service deployment information deals with service interface definition, statefulness and service session management. On the one hand, JOLIE offers a simple syntax for dealing with service composition and efficient multiple request processing; on the other hand, it is based on a formal semantics which offers a solid development base, along with the future possibility of creating automated tools for testing system properties such as deadlock freeness.

1 Introduction

Usually related to orchestration languages, service composition and service statefulness are key concepts in Web Service system programming. Service composition deals with the ability to aggregate existent services in order to obtain a new one which offers more complex functionalities, whereas service statefulness deals with the ability to mantain the state of a conversation among different services until the end of the so-called *business activity*. Orchestration languages, on the one hand, provide workflow constructs such as sequence, parallelism and non-deterministic choice for composing communication interactions and, on the other hand, they deal with statefulness by activating different workflow instances for each business task to manage. At the present, the most credited orchestration language is WS-BPEL (BPEL for short) [OAS], which is an XML-based language developed by OASIS. BPEL provides a great number of constructs, both for dealing with a workflow design paradigm and event based activities. Due to its very rich XML-based syntax, programming directly in BPEL is rather complex. For this reason specific tools for supporting BPEL designers and programmers are needed, such as those provided by activeBPEL [act] or Oracle [ora]. Moreover, BPEL is not equipped with a formal semantics.

In this paper we present JOLIE, Java Orchestration Language Interpreter Engine, an opensource project released under the LGPL license [Pro] and publicly available for consultation and use [JOLb]. JOLIE is the full implementation of our formal calculus for service orchestration, called SOCK [GLG+06]. In SOCK we have formalized the basic features of the Service Oriented Computing paradigm and we have provided a language syntax with few basic constructs which allows for the composition of services. SOCK is equipped with a formal semantics and is structured on three layers: the *service behaviour layer*, the *service engine layer* and the *services system layer*. The service behaviour layer allows for the design of service behaviours by supplying computational and external communication primitives inspired to Web Services operations and workflow operators (e.g. sequence, parallel and choice). The service engine layer is built on top of the former and allows for the specification of the service deployment, where it is possible to design in an orthogonal way three main features which directly deal with service instances, called *sessions*, and service statefulness: *execution modality*, *state persistence* and *correlation sets*. Execution modality deals with the possibility to execute service sessions in a sequential order or in a concurrent way; state persistence allows to specify if each session has its own independent state or if the state is shared among all the sessions of the service engine. Depending on the execution modality and state persistence features we distinguish four service categories:

1. *concurrent/not persistent*: they are services which concurrently execute their sessions where each of them is equipped with its own state that expires when the ses-

*Research partially funded by EU Integrated Project Sensoria, contract n. 016004.

0-7695-3044-3/07 $25.00 © 2007 IEEE
DOI 10.1109/ECOWS.2007.19

sion is terminated. Usually, BPEL processes belong to this service category.

2. *concurrent/persistent*: they are services which concurrently execute their sessions. Sessions share a common state which does not expire when session terminates but the stored information are available for next session executions. Usually, services which allow for the management of databases belong to this service category.

3. *sequential/not persistent*: they are services whose sessions are forced to be executed following a sequential order where each session is equipped with its own state that expires when the session is terminated. For example, a video game is accessed sequentially by each player and each game is different from the previous one.

4. *sequential/persistent*: they are services whose sessions are forced to be executed following a sequential order, where the state does not expire after a session termination but the stored information are available for next session executions. A cashpoint machine could be considered as an example of service belonging to this category.

The correlation sets mechanism, which is inspired by the BPEL one, allows us to distinguish sessions initiated by different dialoguers by means of the values received within some specified variables. It is worth noting that session statefulness can be easily achieved by exploiting correlation sets jointly to not persistent states. In these cases, indeed, each session has its own state identified by a correlation set which expires when the session terminates. Finally, the services system layer defines the semantics of service interactions (i.e. the functioning of message exchanging between services), allowing for the reasoning about the behaviour of the whole system.

JOLIE, whose preliminary behavioural language has already been presented in [MGLZ06], inherits all the formal semantics of SOCK and provides a syntax which resembles that of the C language. This is in contrast with the most credited Web Services orchestration languages, such as XLANG [Tha] and BPEL, which are based upon XML. In particular, JOLIE supports the three layers of SOCK separately. The service behaviour layer and the service engine one are user specifiable by means of two different files: the *behavioural file* and the *deployment file*, respectively. On the contrary, the service system layer is embedded in the Communication Core of the JOLIE implementation as detailed in Section 4. The interpreter implementation has been written in Java, making a strong use of concepts such as cohesion, encapsulation and modularization, which we exploit in our internal architecture in order to permit the integration with different technologies and heterogeneous environments.

The behavioural file (denoted by the `.ol` extension) contains the workflow definition of the orchestrator, whereas the deployment file (denoted by the `.dol` extension) contains directives for the execution engine and specifies information for the integration of the orchestrator in the service oriented architecture. The fact that JOLIE provides two different files for programming the behaviour and the system integration information of the service is a key feature; this allows the programmer for the reuse of existing behavioural files in different service environments (by coupling it with a new deployment file) and for the reuse of deployment files with compatible workflows. It is worth noting that the syntax offered by SOCK is mapped by the languages of the two files. In fact, the behavioural language maps the syntax of the service behaviour layer, whereas the deployment language maps the syntax of the service engine layer. As the services system layer is only a semantic layer and does not specify a syntax, there is no corresponding JOLIE language for it. As far as the behavioural language is concerned, it is possible to interact with other services by means of communication primitives inspired by WSDL operations (One-Way, Request-Response, Notification and Solicit-Response), to model timeouts, to synchronize internal parallel processes, to use the classic `while` loop instruction and the `if-then-else` conditional statement. Moreover, the programmer is allowed to compose statements in a workflow by making sequences, parallelisms and non-deterministic choices. Using its communication primitives and its compositional operators, JOLIE can compose other services by exploiting their input operations. Finally, the behavioural language provides statements for user input/output console interaction. As far as the deployment language is concerned, its grammar structure is composed by two main parts. The first part contains the deployment directives (execution modality, the state mode (persistent or not persistent) and the correlation set of the orchestrator; these directives map the same features provided by the service engine layer of SOCK for dealing with sessions and service statefulness. The second part deals with interfaces and contains all the information needed for interaction with other services: operations, port types, protocol/port type bindings and service deployment endpoints.

Summarizing, the main advantages of JOLIE are: (i) the programmer-friendly syntax that permits fast orchestration prototyping and the subsequent step-by-step incremental extension; (ii) the distinction between the behavioural and the deployment files supporting the full decoupling between the orchestration logics and the actual development technology; (iii) the solid formal semantics provided by the SOCK calculus [GLG$^+$06] allowing for the formal reason-

ing about JOLIE orchestrators.

2. JOLIE behavioural language overview

This section is devoted to a presentation of the JOLIE behavioural language, which corresponds to the syntax of the service behaviour layer found in SOCK. For the sake of brevity, we report only the basic features of the language, without considering fault and compensation handling instructions.

2.1. Program structure

A JOLIE program is defined as follows:

Program ::=
  `locations` { *id-list* } | ϵ
  `operations` { *Operation-declarations** } | ϵ
  `variables` { *id-list* } | ϵ
  `links` { *id-list* } | ϵ
  definition*
  `main` { *Process* }
  definition*
definition ::= `define` *id* { *Process* }

where we represent non-terminal symbols in italic and the Kleene star represents a zero or more time repetition. An *id* represents an unambiguous identifier (i.e. a unique name), whereas an *id-list* is a list of them. The first program part is declarative: the programmer uses it to specify the locations, operations, variables and internal synchronization links it is going to use in the workflow code. The second part defines the workflow of the orchestrator, formed by an entry point (`main`) and user-defineable procedures (*definition*s). For the sake of clarity, the various program parts are individually explained in the following.

Locations A *location* represents a communication endpoint to a service, used by JOLIE to create a communication channel. The behavioural program requires only an *id-list* for location declarations: their real value is to be specified in the deployment file. This allows us to decouple the workflow design from a communication detail: location changes of other services are reflected only in the deployment file, leaving the behaviour of the orchestrator unmodified.

Operations Operations are used to interact with other services, invoking one of their exposed functionalities. JOLIE supports the four operation types defined in the WSDL specification: One-Way, Request-Response, Notification, Solicit-Response. One-Way and Request-Response are input operations: a One-Way operation simply waits for a message and receives it, whereas a Request-Response operation waits for a message, executes a code block and then sends a response message to the invoker. Notification and Solicit-Response are, respectively, the output counterparts of input operations: a Notification operation is used to invoke a One-Way operation of another service by sending a message, while a Solicit-Response operation invokes a Request-Response operation by sending a message and then remains blocked until it receives the response. The non-terminal follows:

Operation-declarations ::=
  `OneWay:` *SingleWayOp-decl**
  | `RequestResponse:` *RequestResponse-decl**
  | `Notification:` *SingleWayOp-decl**
  | `SolicitResponse:` *SolicitResponse-decl**
SingleWayOp-decl ::= *id*<*var-type-list*>
RequestResponse-decl ::=
  id<*var-type-list*><*var-type-list*>
SolicitResponse-decl ::=
  id<*var-type-list*><*var-type-list*>
var-type-list ::= *var-type**
var-type ::= `int` | `string` | `variant`

A One-Way operation needs a list of variable types (*var-type-list*) in its declaration. When the interpreter receives a message for that One-Way operation, it checks the incoming message value types with the given *var-type-list*: if the types do not correspond, the message is rejected. A Notification operation variable types are used when the interpreter sends a message with that operation: the variables used in the workflow are automatically cast to the types written here before sending the message. Request-Response operations need two lists, the first for input variable types and the second for output variable types. Solicit-Response operations, inversely, use the first list for output variable types and the second list for input variable types. Supported variable types are `int` (a Java based integer), `string` (a Java based string) and `variant`. The variant type matches with both `int` and `string`: if specified in an input *var-type-list*, the interpreter accepts any type for that value and the corresponding incoming variable will implicitly take that type, while if specified in an output *var-type-list* the related variable in the workflow maintains its current implicit type. Let us comment the following example:

```
operations {
OneWay:
  myFirstOW< int, string >, mySecondOW<>
RequestResponse:
  myRR< int, int, variant >< int, variant >
Notification:
  myNotification<>
SolicitResponse:
  mySR< int, string, variant >< int, int >
}
```

where `myFirstOW` receives two values, which must be an integer and a string value, respectively. If the received values are not of the right type, JOLIE will refuse to receive the message. In the case of `mySR`, the interpreter will convert its first value to an integer, its second to a string and its third will maintain its current type.

Variables JOLIE makes use of dynamic typing: variable types are not declared and errors are caught during program execution (similar behaviours can be found in other languages, e.g. Perl, PHP and JavaScript). The interpreter needs only that the program declares in advance the variables it is going to use during the execution, specifying an *id-list* in the `variables` block. Implicit supported variable types are integers and strings.

Links Links are used for internal parallel processes synchronization. As seen for variables, the `links` declarative block requires only a list of identifiers.

Definitions Definitions allow the programmer to define procedures, which will be callable thereafter by using their identifiers as statements. An example of a procedure definition follows:

```
define printHello { out( "Hello, world!" ) }
```

Main The `main` block is the starting procedure of the program execution. Informally, it is comparable to the main function of a C or Java program.

2.2. Statements

For the sake of brevity, we show only a short overview of the available instructions; a more detailed description is available in [MGLZ06].
The communication primitives are:
One-Way. *id*<*id list*> : waits for a message on the operation whose name is specified with *id* and stores the received values in the *id list* variables.
Request-Response. *id*<*id list*> <*id list*> (*Process*) : waits for a message on the operation whose name is specified with *id*, stores the received values in the first *id list* variables, executes the code block *Process* and sends a response message composed with the values of the second *id list* variables. It is worth noting that, differently from BPEL, in JOLIE the Request-Response operation is specified atomically, where *Process* represents the activities to be executed between the request reception and the response sending.
Notification. *id*@*id*<*id list*> : sends a message on the operation whose name is specified with the first *id* to the corresponding One-Way operation (which is specified in the deployment file), which contains the values of the *id list* variables, to the communication endpoint represented by the second *id*. The second *id* can be a location declared in the locations block or a variable containing a string that can be evaluated as a location. It is worth noting that such a feature implements location mobility. It is possible, indeed, to receive a location which can be exploited afterwards for executing a Notification or a Solicit-Response statement.
Solicit-Response. *id*@*id*<*id list*> <*id list*> : send a message on the operation whose name is specified with the first *id* to the corresponding Request-Response operation, composed with the values of the first *id list* variables, to the communication endpoint represented by the second *id* (which can be, as for the Notification statement, a location or a variable). Once the message is sent, it waits for a response message from the invoked Request-Response and stores the received values in the second *id list* variables.

Basic program flow control statements are provided through the classic `while` and `if-then-else` constructs (which follow the same syntax of the C language), along with the possibility to call a procedure by writing its identifier. Internal parallel processes synchronization is performed by using the `linkIn` and `linkOut` statements, which require an internal synchronization link *id* as a parameter. There are also a *no-op* statement (`nullProcess`) and a `sleep` one; the latter is particularly useful for modeling a timeout when waiting for a communication input or a parallel activity. Moreover, JOLIE provides the possibility to evaluate expressions and make variable assignments.

It is worth noting that, differently from BPEL, JOLIE is able to interact with the executing user, thanks to the `in` and `out` instructions. The former waits for a user console input and stores it in a variable, while the latter displays a message on the screen.

Statement composition JOLIE provides three ways to compose statements: making a sequence, a parallelism or a non-deterministic choice. Every composition can be formed by any number of elements.

Sequences are composed by exploiting the `;` operator, so that `A ; B` executes `A`, waits for it to finish and then executes `B`.

Parallelisms are composed by the | operator. `A | B` executes `A` and `B` in parallel, and waits for the termination of both.

A non-deterministic choice can be expressed among different guarded branches by using the `++` operator. A branch guard can only be an input operation, a `linkIn` statement, an `in` statement or a `sleep` statement, whereas the branch can be any possible process. Let

$$(g_1, p_1), (g_2, p_2), \ldots, (g_{n-1}, p_{n-1}), (g_n, p_n)$$

be branches where g is the branch guard and p the guarded process. The syntax of the non-deterministic choice follows:

$$[g_1]\,p_1 \texttt{++} [g_2]\,p_2 \texttt{++} \ldots \texttt{++} [g_{n-1}]\,p_{n-1} \texttt{++} [g_n]\,p_n$$

The guards are defined within square brackets. When a non-deterministic choice is reached, the interpreter waits for an input on one of its guards. Once an input comes for a guard g_i, the related process p_i is executed and the other branches are deactivated. It is worth noting that the `in`, `linkIn` and `sleep` instructions are useable as branch guards; this is useful to permit user interaction, receive inputs from internal processes or setting timeouts. As a reference, notice that the non-deterministic choice construct follows a behaviour similar to that of the `pick` activity found in BPEL where, differently, it is not possible to specify inputs on internal links.

The statement composers interpretation priority is: `;` | `++`. In the following example, where A, B, C and D are statements, we show how priority works.

```
[req1<a>] A | B ; C ++ [req2<b>] D ; C ; B | D
```

This code fragment contains a non-deterministic choice between two branches. The branches are guarded by two One-Way operations: (`req1<a>` and `req2<b>`). Considering the operator priority, the same code would be explicited as follows.

```
[req1<a>]
  ( A | ( B ; C ) )
++ [req2<b>]
  ( ( D ; C ; B ) | D )
```

2.3. The factorial service

We present now a simple, yet practical, example of how to write a correct behavioural file for the realisation of a service which calculates the factorial of a given number. The code follows:

```
operations {
  RequestResponse:
    calculateFactorialRR< int >< int >
}
variables { i, n, result }
define calcFactorial {
  while( i < n ) {
    i = i + 1; result = result * i
  }
}
main {
  i = 0; result = 1;
  calculateFactorialRR< n >< result >
    ( calcFactorial )
}
```

The orchestrator initializes its variables (`i`, `n` and `result`) and then waits for a request for the operation `calculateFactorialRR`. When a correct request (i.e. a message containing a single `int` value) for the operation is received, JOLIE will call the `calcFactorial` procedure, and then send back to the caller the value stored in `result`.

3. JOLIE deployment language

The deployment file is composed by two main parts: the first one contains *deployment directives*, whereas the second defines the *interfaces* needed for the integration of the orchestrator in the target service oriented architecture. In the following we examine the instructions offered by the deployment language, providing at the end an example of their use.

3.1. Deployment information structure

The JOLIE deployment information structure is represented by the following grammar:

Deployment-information ::=
`state` { *State-mode* }$^?$
`execution` { *Execution-modality* }$^?$
`cset` { *id-list* }$^?$
`locations` { *Location-definitions* }$^?$
`interface` { *Interface-definition* }$^?$

where we exploit the $^?$ notation to show that a block is optional and may be left unspecified. It is worth noting that the first three parts (`state`, `execution` and `cset`) correspond to the service engine layer syntax of SOCK. In the following we explain the structure elements separately.

3.2. Deployment directives

State persistence The `state` instruction indicates how the active sessions access the variables. The programmer can choose between two values:

State-mode ::= `persistent` | `not_persistent`

On the one hand, in a `persistent` state mode the variables are treated as shared among all the sessions; on the other hand, a `not_persistent` state mode makes every session owning its own independent variable state. If the `state` block is left unspecified, JOLIE sets its value to `persistent`.

Execution modality JOLIE supports three possible execution modalities:

Execution-modality ::=
`single` | `sequential` | `concurrent`

The interpreter handles sessions depending on the value of the `execution` block:
`single`: the orchestrator does not create sessions, it just runs the code only one time.
`sequential`: the orchestrator creates sessions sequentially, enqueuing the incoming requests.
`concurrent`: the orchestrator creates sessions concurrently, handling all the requests in parallel.
If the `execution` block is left unspecified, JOLIE sets its value to `single`.

The interpreter generates a session when the first input operation statement, specified in the behavioural file, receives a message. Notice that the input operation can be a One-Way, a Request-Response or a non-deterministic choice which comprehends one or more of them. Consider the following `main` procedure:

```
main {
  out( "Starting..." ); myOneWay< x >; out( x )
}
```

which we suppose is executed with a concurrent execution modality and a not persistent state mode. This code block prints the `Starting...` string, then waits for input messages for the `myOneWay` operation. Whenever an input message for `myOneWay` is received, JOLIE creates a new session, which prints the received value on the screen.

It is worth noting that the `concurrent` modality introduces a problem related to system resources; in JOLIE, every concurrent session is executed by a separate thread. Such a mechanisms offers real concurrency and good scalability, but threads have a cost in terms of system memory allocation. In order to address this issue, JOLIE offers a command line parameter which permits to specify a connection limit: once the number of running sessions reaches the connection limit, JOLIE begins to enqueue the incoming requests. Before JOLIE starts their processing, enqueued requests have to wait for the number of running sessions to decrease. The command line parameter for the specification of connection limit is `-l [number]`; an example of its use follows:

```
jolie -l 1000
```

which tells JOLIE to run at most one thousand sessions at a time. In case the `-l` option is not passed, JOLIE exploits an heuristic approach to decide dynamically if a request should be enqueued or not. The aim of the heuristic choice is to avoid swapping as much as possible, which would result in a seriously slow execution.

Correlation set Sessions often require to be distinguished and accessed only by those invokers which hold some specific references. In other paradigms, such as the Object-oriented one, such references are managed by the underlying framework. Unfortunately, we can't make such an assumption in the service oriented computing model; correlation sets, introduced by BPEL, allow us to address such an issue. An orchestrator may define a set of *correlated variables* through the `cset` instruction (e.g. `cset { a, b, c }`): the specified set becomes the correlation set to use for session referencing. The functioning of correlation sets in SOCK (and thus, in JOLIE) is extensively explained in [GLG+06]. For the sake of clarity, in the following we report an example of their usage, modeling a simplified mechanism for the creation of new gaming sessions of a one player game:

```
main {
  keepRun = 1; createGame<>< id >( newGame );
  while( keepRun ) {
    makeMove< id, move >;
    if ( move == "quit" ) { keepRun = 0 }
    else { playMove }
  }
}
```

which we suppose coupled with the following deployment directives:

```
state { not_persistent }
execution { concurrent }
cset { id }
```

The behavioural code waits for a message for the `createGame` operation. Once the orchestrator receives a message for that operation, it creates a new concurrent session, which calls the `newGame` procedure (responsible for getting a fresh game `id` by interacting with another service) and sends back the identifier of the created game. The orchestrator then enters in a loop which accepts game moves by means of the `makeMove` operation. The user can end the gaming session by issuing the string `quit` as a move. The problem is related to the fact that there could be a lot of sessions in concurrent execution; so, when a message for `makeMove` comes, we need to decide which session should receive it. The correlated variable `id` addresses this issue: once the interpreter reads its value in the message for `makeMove`, it chooses the session which has the same value stored in the same variable. For example, suppose that we created two gaming sessions, session A with `id=2` and session B with `id=4`, and that an invoker sends a message containing the values < 4, "quit" >. Such a message will be routed by the interpreter to session B, because the first value of the message corresponds to the actual value of the `id` variable within its session state.

Locations Locations declared in the behavioural file must have their value specified in the `location` block of the deployment file:
Location-definitions ::=
id = "*URI*" *Location-definitions* | ϵ

where *URI* is to be intended as a standard Uniform Resource Identifier. Currently, JOLIE supports only socket based communications. An example containing location definitions follows:

```
locations {
  myService = "socket://www.adnsname.com:80",
  ipUri = "socket://123.12.13.111:3000"
}
```

3.3. Interface definition

The `interface` block describes the interface offered by the orchestrator, along with the protocols to use for invoking the input operations of other services. This deployment file section is greatly inspired by WSDL, being our first step for a WSDL export tool. Indeed, the non-terminal *Interface-definition* contains elements similar to these found in a WSDL file:

Interface-definition ::=
`operations` { *Operation-defs* }
`inputPortTypes` { *PortType-def** }
`outputPortTypes` { *PortType-def** }
`bindings` { *Binding-def** }
`service` { *Service-def** }

Operations The `operations` block permits to specify interface related operation information, such as the variable names to use in message exchanging (particularly useful for SOAP based transmissions) and the input operation name an output operation has to invoke. The syntax is:

Operation-defs ::=
`OneWay` : *OneWay-def**
| `RequestResponse` : *RequestResponse-def**
| `Notification` : *Notification-def**
| `SolicitResponse` : *SolicitResponse-def**

OneWay-def ::= *id*<*id-list*>
RequestResponse-decl ::= *id*<*id-list*><*id-list*>
Notification-def ::= *id*<*id-list*> = *id*
SolicitResponse-decl ::= *id*<*id-list*><*id-list*> = *id*

where the *id-list*s in the angular brackets represent the bound variable names to use in message exchanging. The output operation definitions require an *id* to be bound through the = operator: that *id* will be used to identify the input operation to invoke on the other service when the behavioural code will call the output operation.

Input port types Input port types are collections of input operations; they are used in the `bindings` and the `service` blocks to define the orchestrator input interface. Thus, each port type requires only a non-empty *id-list* of input operations:

PortType-def ::= *id* : *id-list*
where *id* is the name of the created port type.

Output port types Counterparts of input port types, output port types define collections of output operations; they are used only in the `bindings` block, in order to define the protocol to use when calling an output operation of the bound output port type. Output port types are defined with the same non-terminal of input port types, *PortType-def*.

Bindings A binding creates a port from a port type and defines its communication protocol. The syntax is:
Binding-def ::= *id* : *id* : *id*
where the first *id* is the name of the port to create, the second *id* is the port type to use and the third *id* is a communication protocol. Currently, JOLIE supports fully a proprietary communication protocol (named SODEP[1]) and partially SOAP (only for flat structured messages). Therefore, the programmer can use the `sodep` or `soap` keyword to specify a port protocol.

Services Services represent the input interface of the orchestrator. The syntax follows:
"*URI*" : *id*
where *URI* is a standard Uniform Resource Identifier and *id* is an input port specified in the `bindings` section. Each service entry define an input communication endpoint which other services can use to invoke the relative input port operations.

Deploying the factorial service In the following, we discuss the deployment information to be coupled with the behavioural code presented in 2.3. The code follows:

```
state { not_persistent }
execution { concurrent }
cset {}
interface {
  operations {
  RequestResponse:
    factorialRR = factorialRR< number >< result >
  }
  inputPortTypes { factorialPortType: factorialRR }
  bindings {
    soapFactorialPort: factorialPortType : soap
  }
  service {
    "socket://localhost:2555": soapFactorialPort
  }
}
```

Here, we instruct the interpreter to execute sessions using a not persistent state mode (`state { not_persistent }`) and in a concurrent way (`execution { concurrent }`). We do not need

[1] Simple Operation Data Exchange Protocol

to specify anything in the correlation set, as the service does not need to identify its sessions. Furthermore, in the `operations` block, we specify the names the variables assume in `factorialRR` message exchanging (`number` and `result`). This approach allows the programmer to decouple the behavioural programming from the SOAP message naming details: the variables contained in the `factorialRR` statement of the behavioural file will be automatically renamed to `number` and `result` for message creation and receiving, respectively. Finally, we create an input port (`factorialPortType`) containing our operation, we bind the soap protocol to it (in the `bindings` block) and we expose it on a socket based service, network port 2555 (in the `service` block).

4. JOLIE internal architecture

In this section we report an overview of the internal architecture of JOLIE, along with its connections with the semantic layers of SOCK, which are the *service behaviour layer*, the *service engine layer* and the *services system layer*. A more accurate description of the implementation details, including a graphical representation of the architecture, is available in [MGLZ06].

Code analysis JOLIE offers a library for code analysis, which is the same used by the interpreter itself for parsing its input files and obtain an optimized abstract syntax tree. The library offers the possibility to exploit the `Visitor` object oriented design pattern [Wik], in order to analyze and/or manipulate the parsing result. For example, the JOLIE internal code optimizer and program well-formedness checker (which takes its rules by the SOCK specifications) are implemented by means of this approach.

Object Oriented Interpretation Tree Responsible for the execution of the behavioural code, the Object Oriented Interpretation Tree (OOIT) is a tree composed by small execution units. Each semantic rule specified by the service behaviour layer is implemented by an OOIT execution unit. This approach based on encapsulation makes very simple to update the interpreter semantics w.r.t. new developments of SOCK. The OOIT is produced by JOLIE starting from the optimized abstract syntax tree.

Runtime environment The runtime environment handles the creation of new sessions, the synchronization of processes and the service state. It interacts with all the other components of JOLIE and abstracts the OOIT from session state handling. This component implements rigorously the semantics of the service engine layer.

Communication core The communication core permits to keep the OOIT separated from communication related problematics. This component handles incoming connections and internal message routing to the various sessions, along with the service input interface deployment as specified in the `service` block of the deployment file. Moreover, its modularized design permits to implement easily support for new protocols and communication mediums (such as files and local memory). The communication core implements the semantics of the services system layer, enabling JOLIE to communicate with other services.

5. A business case study

In this section we present a typical business scenario programmed with JOLIE, where there are five participants involved: a customer, a market, a service register, a supplier and a bank. The customer wants to buy a product and asks for its price to the market. The market queries the register in order to obtain a supplier which is able to satisfy the customer and then it requests the supplier for the price. The market forwards the price to the customer, that decides either to buy or not. If it decides to buy, the market requests for the order to the supplier and, concurrently, it asks to the bank to perform the financial transaction. In order to do that, the bank will request both the customer and the supplier for the bank data. At the end, the bank will notify the customer, the supplier and the market for the transaction termination.

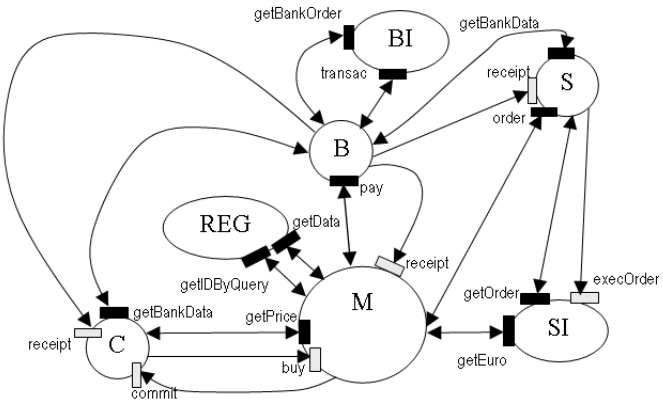

Figure 1. Example architecture

In Fig. 1 we report a graphical representation of the system, where circles represent services, black rectangles represent Request-Response operations, white rectangles represent One-Way operations and arrows represent the interactions among the services. The system is composed by the following services: the *Register Service* REG, the Bank Service composed by the *Bank Information Service* BI and

the *Bank Master Service* B, the Supplier Service composed by the *Supplier Information Service* SI and the *Supplier Master Service* S, the *Market Service* M and the *Customer* C. The Bank Information Service and the Supplier Information Service model services which manage persistent information repositories such as the bank account database and the product list database, respectively.

In the following, we present the code for the market service. For the reader's convenience, each operation statement has been suffixed with its respective type: `OW` for One-Way operations, `RR` for Request-Response operations, `N` for Notification operations and `SR` for Solicit-Response operations. The complete and executable example of the entire service system can be downloaded at [JOLa].

```
main {
  getPriceRR< quantity, clientLoc, product >
  < euro >(
    getIdByQuerySR@register< product >< supId >;
    getDataSR@register< supId >
      < supLoc1, supLoc2 >;
    myLoc = "socket://localhost:2564";
    getEuroSR@supLoc1< product >< price >;
    euro = price * quantity;
  ); ack = "ok";
  [ sleep( 3000 ) ] (
    ack = "timeout";
    buyRR< quantity, clientLoc, product, conf>
      < ack >( nullProcess )
    )
  ++
  [ buyRR< quantity, clientLoc, product, conf >
  < ack >( nullProcess )]
    if( conf=="yes" ) {
      orderSR@supLoc2
        < quantity, clientLoc, product >
        < idorder >;
      paySR@bank
        < idorder, clientLoc,
        supLoc2, myLoc, euro >< bkId >;
      receiptOW< bkId, idtran >;
      commitN@clientLoc<>
    }
}
```

Since the Market Service has to manage different requests from different clients, it is deployed by exploiting a concurrent execution modality and a not persistent state where the correlation set is {`product,quantity,clientLoc,bkId`}. Variable `product` contains the product type, variable `quantity` contains the requested quantity for the given product, variable `clientLoc` contains the location of the client and, finally, variable `bkId` contains the unique identifier for the Bank Master Service session. The behaviour of the Market Service starts with the Request-Response statement `getPriceRR`, which takes as inputs the variables `quantity,clientLoc,product` and returns the product price. Between the request message and the response one, the body of `getPriceRR` performs two invocations to the Register Service (`getIdByQuerySR` and `getDataSR` operations), in order to retrieve the supplier location. The supplier location is exploited for requesting the product price to the Supplier Service (`getEuroSR` operation). Once the `getPriceRR` statement is performed, the Market Service starts a race between an internal timeout and the confirmation message from the customer. Such a race is programmed by means of a non-deterministic choice between the statement *sleep(3000)* and the Request-Response statement `buyRR`. It is worth noting that if the timeout occurs, the `buyRR` operation is still able to receive a message from the customer, but it returns the value `"timeout"` in variable `ack`, which will be tested by the customer application for verifying if the Market Service session has expired. If the customer sends its message before the timeout occurs and confirms to buy, the Market Service invokes the Supplier for initiating the order (`orderSR` operation) and then the Bank Master Service for initiating the financial transaction (`paySR` operation). Finally, it waits for a receipt from the Bank Master Service and then sends a commit to the customer.

6 Related works

JOLIE represents a complementary approach fo service composition w.r.t. BPEL. The main differences can be summarized as follows. JOLIE supports a Java-like syntax that is simpler than the XML one, which needs graphical tools for managing its complexity; we think that textual and graphical programming can be both useful as pseudocode and flow-charts in traditional imperative programming. JOLIE supports decoupling between behaviour and deployment as partially done also by BPEL/WSDL; in JOLIE correlation sets are considered at the level of deployment instead of behaviour and, moreover, we permit the specification of execution modality and state persistence which are not supported by BPEL/WSDL. Finally, JOLIE is built upon the solid formal semantics provided by the SOCK calculus. On the contrary, the official semantics of BPEL is informal; several (and sometimes unrelated) formalizations of BPEL have been given, but usually these formalizations do not cope with the entire very rich BPEL syntax (see [BK06] for an overview).

Relevant orchestration languages based upon a process-calculus formal semantics are inspired by the π-calculus instead of SOCK, see e.g. [CLM05] and [FGK03]. Strong points of these works are the easy manipulation of XML messages and the underlying scalable architecture. The separation between behaviour and deployment, which is a peculiar feature of JOLIE, on the other hand better supports an architecture which is communication technology independent. Another difference is that we more closely reflect the message passing style of Service Oriented Archi-

tectures based on operations and correlation sets, instead of exploiting the typical channel based communication of the π-calculus.

A different approach with respect to the above textual languages is showed in [act, ora, PA03], where service composition is obtained by means of visual programming languages. We think that textual and graphical programming can be both useful in the context of orchestration programming as pseudocode and flow-charts in traditional imperative programming; only the next few years of research on Service Oriented Computing will clarify the relative advantages and disadvantages of the two approaches.

7. Conclusions

We presented JOLIE, Java Orchestration Language Interpreter Engine, and showed a practical example of service composition through its usage.

Future works will cover the introduction of structured data values, in order to manipulate XML messages. In order to improve furthermore our compatibility with the Web Services technology, we plan to exploit the new data handling syntax jointly with the JOLIE deployment language to make possible for the interpreter to understand WSDL files of other services and to publish its own WSDL file. By exploiting the flexibility of the internal communication core, we plan to support SOAP based communications and other IPC (Inter Process Communication) mechanisms. Following recent developments of SOCK, a fault and compensation handling mechanism for JOLIE has been developed and is under testing. Finally, future versions of JOLIE will permit to integrate Java code in an orchestrator workflow. This will permit to use JOLIE even for complex client applications or heavy computational tasks.

References

[act] ActiveBPEL Open Source Engine. [http://www.active-endpoints.com/active-bpel-engine-overview.htm].

[BK06] F. Breugel and M. Koshkina. *Models and verification of BPEL*, 2006. [http://www.cse.yorku.ca/∼franck/research/drafts/tutorial.pdf].

[CLM05] Samuele Carpineti, Cosimo Laneve, and Paolo Milazzo. Bopi - a distributed machine for experimenting web services technologies. In *Fifth International Conference on Application of Concurrency to System Design (ACSD 2005), 6-9 June 2005, St. Malo, France*, pages 202–211. IEEE Computer Society, 2005.

[FGK03] Daniela Florescu, Andreas Grünhagen, and Donald Kossmann. Xl: a platform for web services. In *CIDR*, 2003.

[GLG+06] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: A calculus for service oriented computing. In *Proc. of ICSOC'06*, volume 4294 of *Lecture Notes in Computer Science*, pages 327–338. Springer-Verlag, 2006.

[JOLa] JOLIE. Business case study. [http://jolie.sourceforge.net/files/ecows07/example.zip], 2007.

[JOLb] JOLIE. JOLIE: a Java Orchestration Language Interpreter Engine. [http://jolie.sourceforge.net/], 2006.

[MGLZ06] F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro. JOLIE: a Java Orchestration Language Interpreter Engine. In *Proc. of CoOrg'06*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006. To appear.

[OAS] OASIS. *Web Services Business Process Execution Language Version 2.0, Working Draft*. [http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf].

[ora] Oracle BPEL process manager. [http://www.oracle.com/technology/products/ias/bpel/index.html].

[PA03] Cesare Pautasso and Gustavo Alonso. Visual composition of web services. In *2003 IEEE Symposium on Human Centric Computing Languages and Environments (HCC 2003), 28-31 October 2003, Auckland, New Zealand*, pages 92–99. IEEE Computer Society, 2003.

[Pro] GNU Project. GNU Lesser General Public License. [http://www.gnu.org/copyleft/gpl.html], 2006.

[Tha] S. Thatte. XLANG: Web Services for Business Process Design. Microsoft Corporation, 2001.

[Wik] Wikipedia. Visitor pattern. [http://en.wikipedia.org/wiki/Visitor_pattern], 2007.