Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 181 (2007) 19–33
www.elsevier.com/locate/entcs

JOLIE: a Java Orchestration Language Interpreter Engine

Fabrizio Montesi[1], Claudio Guidi[2], Roberto Lucchi[2]
Gianluigi Zavattaro[2]

[1] *Corso di Scienze dell'Informazione di Cesena, University of Bologna, Italy*
[2] *Department of Computer Science, University of Bologna, Italy*

Abstract

Service oriented computing is an emerging paradigm for programming distributed applications based on services. Services are simple software elements that supply their functionalities by exhibiting their interfaces and that can be invoked by exploiting simple communication primitives. The emerging mechanism exploited in service oriented computing for composing services –in order to provide more complex functionalities– is by means of *orchestrators*. An orchestrator is able to invoke and coordinate other services by exploiting typical workflow patterns such as parallel composition, sequencing and choices. Examples of orchestration languages are XLANG [5] and WS-BPEL [7]. In this paper we present JOLIE, an interpreter and engine for orchestration programs. The main novelties of JOLIE are that it provides an easy to use development environment (because it supports a more programmer friendly C/Java-like syntax instead of an XML-based syntax) and it is based on a solid mathematical underlying model (developed in previous works of the authors [2,3,4]).

Keywords: SOA, coordination, orchestration, Java, service, engine

1 Introduction

Service oriented computing is an emerging paradigm for programming distributed applications based on services. Services are simple software elements that supply their functionalities by exhibiting their interfaces and that can be invoked by exploiting simple communication primitives, the so-called One-Way and Request-Response ones. Services can be composed each other in order to design more complex services by exploiting *orchestrators*. The orchestrators, indeed, are able to invoke and coordinate other services by exploiting typical workflow patterns such as parallel composition, sequencing and choices. Furthermore, composition can be also achieved

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

[1] Email: famontesi@gmail.com
[2] Email: cguidi@cs.unibo.it, lucchi@cs.unibo.it, zavattar@cs.unibo.it

doi:10.1016/j.entcs.2007.01.051

by following a different approach, that is choreography, that allows to design a distributed system in a top view manner [2,8]. The most credited technology that deals with service oriented computing is *Web Services* which aims at guaranteeing interoperability among different platforms and whose specifications are defined by means of the XML language. One of the most important specification is WSDL [10] that defines a language for designing a Web Service interface. An interface allows to access service functionalities by means of operations. An operation represents the basic interaction modality of a service and it can be a One-Way operation, where an invoking message is sent to the service, or a Request-Response one, where an invoking message is sent to the service assuming that a response message will be subsequently sent back from it. Web Services can be composed following both orchestration and choreography approaches. As far as orchestration is concerned here we cite WS-BPEL [7], as far as choreography is concerned we cite WS-CDL [9].

In our previous works [2,3,4] we have analyzed orchestration and choreography as synergic approaches for distributed system design by following a formal approach. Our formal investigation aims at supplying a precise formal framework on which we can develop designing tools for service oriented computing systems where orchestration and choreography languages play a fundamental role. In particular, we have formalized both choreography and orchestration languages by means of two process calculi and we have presented a formal notion of conformance between them based on bisimulation. As it emerges by those works the orchestration represents w.r.t. choreography a refinement step towards the implementation of service oriented applications. Informally, if on the one hand choreography does not produce executable systems, on the other hand the orchestration makes it possible to program each service involved in the application. For the sake of brevity, we do not report the formal definition of the syntax and semantics of our orchestration language (for a closer look to the language we remind to the previous works). We simply report a small example in order to give the flavour of the kind of calculus we have developed. Assume a buyer service requests for the price of a particular kind of good to a seller service by sending a message on a Request-Response operation. Then, it invokes a purchase order by sending a message on a One-Way operation. We can model such a service dialog as follows:

$$Buyer ::= [good := apple; \overline{price}@S(good, price); ...; \overline{apple}@S(250), \mathcal{S}_B]_B$$

$$Seller ::= [price(good, eur, good = apple?eur := 100) \mid apple(n); ..., \mathcal{S}_S]_S$$

The buyer is a service located at site B where the *good* variable is initialized to the value *apple*. $\overline{price}@S(good, price)$ means that the buyer invokes the Request-Response operation *price* at the service located at site S sending the variable *good* and storing the response into the variable *price*. Then, the buyer performs some internal computation (that we do not specify for the sake of brevity). Finally, it performs $\overline{apple}@S(250)$ that represents the invocation of the One-Way operation *apple* to the service located at S in order to initiate a purchase order of 250 apples. The ; is a sequential composition operator which means that all the statements must be executed one after the previous one has completed. The seller is a service located

at S which receives a price request on the Request-Response operation *price*; the good for which the price is requested is stored into the variable *good*. The response to be sent back is taken from the variable *eur*, but before sending the response message the seller verifies if the received good corresponds to *apple*, and in this case it assign the value 100 to the variable *eur*. The operation *apple* works in parallel (exploiting the operator |) and waits for an apple purchase order. We leave unspecified the behaviour of the seller after the reception of the purchase order. The terms $\mathcal{S}_B$ and $\mathcal{S}_S$ represent the internal states of the sites B and S, respectively. A site state is a function that associates to the site variables the corresponding state.

In this paper we focus on orchestration, presenting JOLIE (Java Orchestration Language Interpreter Engine) which we have developed in order to animate orchestration programs written in a language based on the formal orchestration process calculus. The syntax of the JOLIE language is C/Java-like in order to provide a more programmer friendly development environment. Indeed, the typical orchestration languages such as XLANG [5] and WS-BPEL [7] have a less human readable XML-based syntax. The above seller service can be rewritten in the JOLIE language as follows: [3]

```
define priceCalc {
    if ( good == "apple" ) {
        eur = 100
    }
}

main {
    price< good >< eur >( call( priceCalc ) ) || apple<n> ;; ...
}
```

where `good`, `eur` and `n` are variables, `price` is a Request-Response operation and `apple` a One-Way operation. `priceCalc` is a subroutine (similar to C procedures) which can be called by using the statement `call(priceCalc)`.

The peculiar and original characteristic of JOLIE is that it combines a solid mathematical basis provided by the orchestration process calculus discussed above with a programmer friendly development and execution environment based on a C/Java-like (instead of a XML-based) syntax. This contrasts with most of the actual Web Services orchestration languages for which the formal operational semantics has been investigated and (partially) defined after the syntax. This contrasts also with the trend of developments of WS-BPEL for which, only after the definition of the orchestration language, an extension that includes the possibility to exploit and invoke Java programs is currently under development (see e.g. BPELJ [6]).

Moreover, JOLIE is a fundamental step in our research in orchestration and choreography languages because it permits us to experimentally verify whether the theoretical approach taken during the design of the process calculi are actually satisfactory also when the orchestration programs are to be actually run and executed. For instance, we had to add some additional constructs to JOLIE which where not included in the corresponding process calculus. In particular, JOLIE implements also an iterative statement `while` and a timing statement `sleep(msec)` for programming processes that wait for a certain amount of milliseconds. The latter is

[3] For the sake of brevity we present only a fragment of the entire code.

particularly useful to program orchestrators which are not willing to wait indefinitely for a service response that will never arrive due to either a communication or a service fault. It is also worth to underline that JOLIE has been developed by strongly exploiting the encapsulation principle and in a modular way which allows us to be protocol and communication medium independent. Namely, it is simple to extend the engine in order to run orchestrators that exploit different and heterogeneous communication medium such as SOAP, Internet sockets, shared files, etc.

The paper is structured as follows: in Section 2 we present the JOLIE language whereas in Section 3 we show the interpreter internals. In Section 4 we present a case study taken from [7] by using JOLIE and in Section 5 we report conclusions and future works.

2 JOLIE language overview

JOLIE provides a C-like syntax for designing orchestrator services. A C-like syntax makes the language intuitive and easy to learn for a programmer customed to it. In the following we introduce some basics of the JOLIE language, except expression and condition syntaxes which are similar to that of C language.

2.1 Identifiers

An *identifier* (often abbreviated to *id*) is an unambiguous name stored in the orchestrator shared memory which identifies a location, an operation, a variable or a link. An identifier must match the following regular expression:

```
[a-zA-Z]([0-9a-zA-Z])*
```

Some JOLIE statements require that the programmer provides a *list of identifiers*, which is formed by identifiers separated by commas (as "identifier1, identifier2, a, b, c"). In the following, we refer to the list of identifiers by using the name *id list*.

2.2 Program structure

A JOLIE program structure is represented by the following grammar:

program::=
`locations` { *Locations-definition** }
`operations` { *Operations-declaration** }
`variables` { *Variables-declaration* }
`links` { *Links-declaration* }
*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 times repetition. For the sake of clarity the non-terminals *Locations-definition*, *Operations-declaration*, *Variables-declaration*, *Links-declaration* and *Process* are separately explained in the following.

2.2.1 Locations

JOLIE communications are socket based: an orchestrator waits for messages on a network port (the default is 2555 [4]). In order to communicate with another orchestrator it is fundamental to know its hostname (or ip address) and the port it is listening to: these information are stored in a *location*. A location definition joins an identifier to a hostname and a port. The non-terminal follows:

Locations-definition:= *id*="*hostname*:*port*"

where we do not define the *hostname* and the *port* non-terminals which must be intended as a representation of any hostname and any port respectively. In the following we do not define the auto explicative non-terminals which will be represented by using italic characters. In the following we present program fragment which shows a possible location declaration:

```
locations {
    localUri = "localhost:2555",
    googleUri = "www.google.com:80",
    ipUri = "192.168.0.1:2556"
}
```

2.2.2 Operations

The operations represent the way a JOLIE orchestrator exploits for interacting with other orchestrators. We distinguish two types of operations:

- Input operations.
- Output operations.

The former represent the access points an orchestrator offers to communicate with it, whereas the latter are used to invoke input operations of another orchestrator. We distinguish two groups of input operations: One-Way and Request-Response. A One-Way operation simply waits for a message, while a Request-Response operation waits for a message, executes a code block and then sends a response message to the invoker. As far as output operations are concerned they can be a Notification or a Solicit-Response operation. The former is used to invoke a One-Way operation of another orchestrator, sending a message to it, while the latter is used to invoke a Request-Response operation. It is worth noting that a Solicit-Response operation, after sending the request message, is blocked until it receives the response one from the invoked service. The non-terminal follows:

[4] the default port can be overridden by command line

Operations-declaration:= `OneWay:` *id list*
| `RequestResponse:` *id list*
| `Notification:` *id-assign list*
| `SolicitResponse:` *id-assign list*
id-assign:= *id*`=`*id*

By definition, input operations expect a list of identifiers, while the output ones expect a list of pairs *id*=*id* (we have identified such a list by using the notation *id-assign list*). As far as the output operations are concerned we distinguish between the operation name used within the orchestrator and the bound operation name of the invoked one. In a pair *idA*=*idB*, *idA* represents the internal operation name whereas *idB* the bound name of the external one to be invoked. Such a language characteristic allows us to decouple the orchestrator code from the external operation name binding. In the following a program fragment shows an example of operation declaration.

```
operations
{
OneWay:
    ow1
RequestResponse:
    rr1, rr2
Notification:
    n1 = serverOneWay1, n2 = serverOneWay2, n3 = serverOneWay3
SolicitResponse:
    sr1 = serverRequestResponse1
}
```

2.2.3 Variables

JOLIE variables are typeless. Implicit supported types are integers and strings. The variables declaration non-terminal requires only a list of identifiers which represent the shared memory variables. The definition follows:

Variables-declaration:= *id list*

2.2.4 Links

Links are used for internal parallel processes synchronization. As for variables the links declaration non-terminal requires only a list of identifiers where the *id*s will represent internal links used for synchronization purposes.

Links-declaration:= *id list*

2.2.5 Definitions

Definitions allows to define a procedure which will be callable by another one by exploiting the `call` statement. Each definition joins an identifier to a *Process*. Syntactically, a *Process* is a piece of code composed by JOLIE statements. Informally, the process defined within a definition can be viewed as the body of a C function.

2.2.6 Main

The main block allows to define the process which will be run at the start of the program execution. Informally, it is comparable to the main function of a C pro-

gram.

2.3 Statements

This paragraph shows a brief survey of JOLIE statements.

2.3.1 Program control flow statements

- `call(` *id* `)` : calls and executes the procedure which has been defined with the given identifier.
- `if (` *condition* `) {...} else if (` *condition* `) {...} else {...}`: condition statement
- `while(` *condition* `) {...}`: loop statement

2.3.2 Operation statements

- *id*<*id list*> : waits for a message for the OneWay operation declared in the operations block as *id*, and stores its values in the *id list* variables.
- *id*<*id list*> <*id list*> (*Process*) : waits for a message for the RequestResponse operation *id*, stores its values in the first *id list* variables, executes the code block *Process* and sends a response message containing the values of the second *id list* variables.
- *id*@*id*<*id list*> : uses the Notification operation represented by the first *id* to send a message which contains the values of the *id list* variables, to the orchestrator located at 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 allows to implement the location mobility. It is possible, indeed, to receive a location which can be exploited for performing a Notification or a Solicit-Response.
- *id*@*id*<*id list*> <*id list*> : uses the SolicitResponse operation represented by the first *id* to send a message which contains the values of the first *id list* variables, to the orchestrator located by the second *id* (which can be, as for the Notification, a location or a variable). Once the message is sent, it waits for a response message from the invoked Request-Response and stores its values in the second *id list* variables.

2.3.3 Synchronizing statements

- `linkIn(` *id* `)` : linkIn and linkOut are used for parallel processes synchronization and must be always considered together. In particular the linkIn waits for a linkOut trigger on the same internal link identified by *id*. In case there are already one or more linkOut processes triggering for the same internal link, it synchronizes itself with one of them by following a non-deterministic policy.
- `linkOut(` *id* `)` : triggers for a linkIn synchronization on the same internal link identified by *id*. In case there are already one or more linkIn processes waiting for the same internal link, it synchronizes itself with one of them by following a

non-deterministic policy.

2.3.4 Console input/output statements

- `in(` *variable id* `)` : waits for a console user input and stores it in the given variable.
- `out(` *expression* `)` : writes the evaluation of the given expression on the console (note that a variable can be considered as an expression).

2.3.5 Others

- `sleep(` n `)` : makes the current process sleeping for n milliseconds where n is a natural.
- `nullProcess` : no-op statement. [5]

2.4 Statement composers

JOLIE provides three ways to compose statements: sequence, parallelism and non-deterministic choice.

2.4.1 Sequence

Sequences are composed by exploiting the ;; operator. Let $x_1, x_2, \ldots, x_{n-1}, x_n$ be statements. Then, the sequential composition

$$x_1;;x_2;;\ldots;;x_{n-1};;x_n$$

executes x_1 and waits for it to terminate, then executes x_2 and waits for it to terminate and continues with this behaviour until it reaches the end of the sequence.

2.4.2 Parallel

Parallel processes are composed by exploiting the $\|$ operator. The $\|$ operator combines sequences (note a single statement is a sequence of one element). Let $s_1, s_2, \ldots, s_{n-1}, s_n$ be sequences. Then, the parallel composition

$$s_1 \| s_2 \| \ldots \| s_{n-1} \| s_n$$

executes every sequence in parallel. A parallel composition is terminated when all the sequences are terminated.

2.4.3 Non-deterministic choice

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

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

[5] the `nullProcess` statement is usually exploited within the RequestResponse when there is no need to execute anything before sending the response.

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 programmed it makes the interpreter waiting for an input on one of its guards. Once an input has come, the related p process is executed and the other branches are deactivated.

2.4.4 Priority of the composers

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
```

In this code fragment there is a non-deterministic choice between two branches guarded by two One-Way operations (`req1<a>` and `req2<b>`). By considering the operator priority the same code would be explicited as follows.

```
[input1](A || (B ;; C)) ++ [input2] ((D ;; C ;; B) || D)
```

2.5 Example

As a practical example, consider a scenario in which we have an orchestrator which acts as a service provider. The orchestrator declares a Request-Response operation, named `factorialRR`, which has the purpose to receive a number and, as a response, to send its factorial. Moreover, the orchestrator has to interact with a logging server in order to communicate its activity for constructing a statistic of its usage. The following code snippet shows a possible implementation. For the sake of brevity, only the `main` procedure is shown.

```
main
{
    while( 1 ) {
        [ factorialRR< n >< result >( call( calcFactorial ) ) ]
            servedClients = servedClients + 1
        ++
        [ linkIn( logLink ) ]
            notifyActivity@logServerUri< servedClients >;;
            servedClients = 0
    }
    ||
    while( 1 ) {
        sleep( 60000 );; /* 60 seconds */
        linkOut( logLink )
    }
}
```

The main process is composed by two processes in parallel. The former defines a non-deterministic choice between the Request-Response on which the service can be accessed for returning the factorial calculation and the linkIn process defined on the internal link `logLink`. The linkOut process which triggers the internal link `logLink` is defined in the second parallel process which, every 60 seconds, interrupts the service for sending the number of the served clients to the logging service located at `logServerUri`.

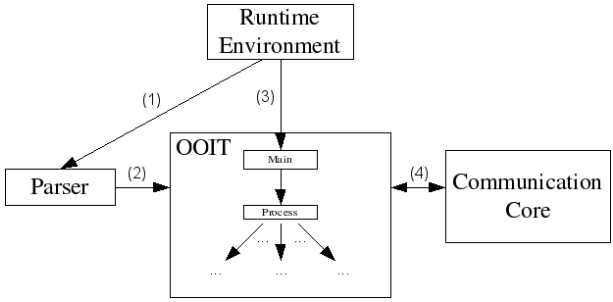

Fig. 1. JOLIE architecture

3 JOLIE interpreter architecture

This section is devoted to describe the architecture of JOLIE.

3.1 Structure overview

Figure 1 describes the JOLIE interpretation algorithm and the parts composing the interpreter. In order to explain how JOLIE works we proceed by describing the main steps of the run-time environment and then its main components: the Parser, the Object Oriented interpretation tree and then the Communication core.

Algorithm 1 JOLIE *interpreter behavior*
Step 1: initialize the communication core.
Step 2: create an instance of the parser.
Step 3: create the Object Oriented Interpretation Tree (OOIT).
Step 4: invoke the **run()** *method of the OOIT's root node (that corresponds to the main).*

We will now examine the various parts composing the interpreter.

3.2 Parser and Object Oriented Interpretation Tree

JOLIE is based on an object oriented infrastructure created during the parsing of the orchestration to be executed, which is realized by a recursive descendant parser. The principle we follow is to create objects as small as possible, which will know –abstracting away from the context– how to execute the simple task they represent. This goal is obtained by exploiting the encapsulation and composition mechanisms. In order to understand how this is realized we first introduce the main components present in the Object Oriented Interpretation Tree: the `Process` class and the *Basic Process* and *Composite Process* concepts. The former is an object class present in the implementation, while the latter are concepts which we will use to distinguish the general behaviour of `Process` objects.

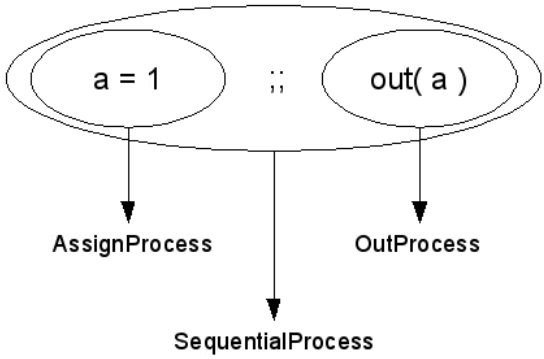

Fig. 2. Objects tree representing `a = 1 ;; out( a )`

3.2.1 The Process class

`Process` is a class representing a generic piece of JOLIE code. `Process` has a **run()** method which performs the activities that the object represents.

3.2.2 Basic Process

A *Basic Process* is a `Process` composed by a single statement of the JOLIE language, like an assignment operation, an output or an input one. The **run()** method in this case performs such a statement.

3.2.3 Composite Process

A *Composite Process* is a `Process` composing other `Process` objects (by running them in parallel, in a sequence or in a non-deterministic choice). The **run()** method executes such composition and, to this end, exploits the **run()** method of the enclosed `Process` objects.

Example 3.1 In order to illustrate how these concepts are used we use the following example:

```
a = 1 ;; out( a )
```

The parser will create three `Process` objects (see Figure 2):

- A `SequentialProcess` (which is as a *Composite Process*) object that encloses the following two processes:
  - An `AssignProcess` (which is a *BasicProcess*) object that assigns the value `1` to `a`.
  - An `OutProcess` (which is a *BasicProcess*) object that prints on the console the value of variable `a`.

When the runtime environment will have to interpret this code block, it will call the **run()** method of the `SequentialProcess` object which will sequentially call the **run()** method of the `AssignProcess` and the `OutProcess` objects it contains. Note that the `SequentialProcess` object knows only that its children are `Process` instances; it simply invokes their **run()** method without knowing anything about their behavior (e.g., they could be themselves *Composite Process* objects).

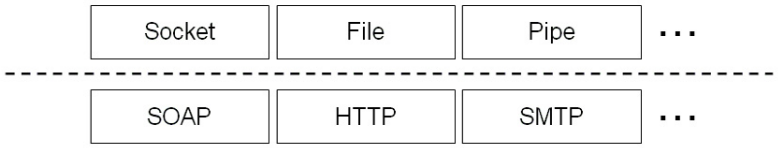

Fig. 3. Communication medium and data protocols

Since this process encapsulation principle is followed in the entire OOIT, starting the execution requires just the call of the **run()** method of the root node (which is the object that contains the `main` process).

3.3 The communication core

The communication core provides an interface for supporting the communication between services that allows us to abstract away from the following aspects:

- The communication medium.
- The communication data protocol.

Figure 3 reports some examples of communication medium and of communication data protocol. For instance the communication medium –which supports the communication– can be a socket, a file or a pipe, while the communication data protocol, which defines how the data should be formatted as well as the interaction modalities that should be used to implement a message exchange, can be (we list the most significant ones in the Internet context) HTTP, SMTP or SOAP.

The communication core supports such abstractions by means of the communication channel `CommChannel` object. The runtime environment exploits the communication channels to send and receive data. Once instantiated, a `CommChannel` object is able to send and receive `CommMessage` (communication message) objects that are composed by:

- The operation name.
- An array of values.

The idea is that each communication channel must be associated to a communication medium and by a communication protocol and that it should be identified by some data that depend on the particular protocols and medium. For instance consider a channel, say c, associated to the SOAP data protocol and to the file "host1@/home/services/op1.ss" communication medium. In order to send a message M on that channel a process must write on the file "host1@/home/services/op1.ss" the SOAP message containing M and, in order to perform an input on that channel, the process must read (and consume the piece of stream it reads) the "host1@/home/services/op1.ss" file by using the SOAP data protocol on the input stream. Although such a interface is designed to support such kind of flexibility on communication medium and data protocol, the current available version of the JOLIE interpreter supports only the socket communication

medium and an internal default data protocol.

4 A purchase order case study

In this Section we present a purchase order case study extracted from the WS-BPEL specifications. More precisely, we present the translation in JOLIE of an example reported in [7]. The aim of this example is to show that JOLIE programs reveals more human readable and manageable than WS-BPEL programs written in XML. For the sake of brevity, we do not report the XML code; the interested reader can find it in [7]. The example models a service for handling a purchase order. The service starts its activity after the reception of a message on the Request-Response operation `sendPurchaseOrder`. Before sending the response message the service executes concurrently three processes defined within the `body` subroutine. One process selects a shipper by invoking the shipping service operation `requestShipping`, another process starts the price calculation by invoking the invoice service operation `InitiatePriceCalculation` and the the third process starts the production scheduling by invoking the operation `requestProductionScheduling` of the production scheduling service. It is worth noting that we abstract away from service locations that are represented by the names `shippingServiceUri`, `InvoiceServiceUri` and `productionSchedulingService`. Furthermore, we remark the use of *linkOut* and *linkIn* statements for synchronizing concurrent processes.

```
locations {
    shUri  = shippingServiceUri,
    inUri = InvoiceServiceUri,
    schUri = productionSchedulingService
}

operations {
OneWay:
    sendSchedule,
    sendInvoice
Notification:
    InPr    = InitiatePriceCalculation,
    SnShPr  = sendShippingPrice,
    rqPrSch = requestProductionScheduling,
    snShSch = sendShippingSchedule

RequestResponse:
    sendPurchaseOrder
SolicitResponse:
    reqShp = requestShipping
}
variables {
    customerInfo, purchaseOrder, IVC, shippingInfo, scheduleInfo
}

links {
    ship-to-invoice, ship-to-scheduling
}

define body {
    reqShp@shUri< customerInfo >< shippingInfo > ;;
    linkOut( ship-to-invoice ) ;; sendSchedule< scheduleInfo > ;;
    linkOut( ship-to-scheduling )

    ||

    InPr@inUri< customerInfo, purchaseOrder > ;;
    linkIn( ship-to-invoice ) ;; SnShPr@inUri< shippingInfo > ;;
    sendInvoice< IVC >
```

```
    ||

    rqPrSch@schUri< customerInfo, purchaseOrder > ;;
    linkIn( ship-to-scheduling ) ;; snShSch@schUri< shippingInfo >
}

main {
    sendPurchaseOrder< customerInfo, purchaseOrder >< IVC >( call( body ) )
}
```

5 Conclusions

JOLIE represents a strict realization of the theoretical orchestration process calculus presented in [3,4]. Along with the possibility to create an orchestrated system (by running multiple instances of the interpreter, on the same computer or on different machines), the internal structure of the interpreter is particularly suitable for future extensions.

Future works will cover the implementation of a new format for locations, which will be aimed to exploit the communication medium independency of the *Communication Core*. By now, every communication uses network sockets. The future format will permit to specify the communication medium in a location definition, giving the possibility to use sockets, files, internal pipes, etc. Moreover, another objective is to make the `operations` block interacting with WSDL definitions. To do so, we will exploit the object oriented internal operations implementation, along with the *Communication Core* data protocol independency. In this way, an orchestrator will be able to use other protocols in order to exchange data with external applications.

JOLIE is also the starting point from which will be developed an implementation for the choreography process calculus in [3], which will join JOLIE to realize a full implementation for the theoretical framework developed in previous work [1,3].

References

[1] Busi, N., R. Gorrieri, C. Guidi, R. Lucchi and G. Zavattaro, *Choreography and Orchestration: a synergic approach for system design*, in: *Proc. of 3rd International Conference on Service Oriented Computing (ICSOC'05)*, LNCS **3826** (2005), pp. 228–240.

[2] Busi, N., R. Gorrieri, C. Guidi, R. Lucchi and G. Zavattaro, *Towards a formal framework for Choreography*, in: *Proc. of 3rd International Workshop on Distributed and Mobile Collaboration (DMC 2005)* (2005).

[3] Busi, N., R. Gorrieri, C. Guidi, R. Lucchi and G. Zavattaro, *Choreography and orchestration conformance for system design*, in: *Proc. of 8th International Conference on Coordination Models and Languages (COORDINATION'06)*, LNCS **to appear**, 2006.

[4] Guidi, C. and R. Lucchi, *Mobility mechanisms in service oriented computing*, in: *Proc. of 8th International Conference on on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, LNCS **to appear**, 2006.

[5] IBM, "XLANG: Web Services for Business Process Design," http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm.

[6] IBM and BEA, "BPELJ: BPEL for Java technology," http://www-128.ibm.com/developerworks/library/specification/ws-bpelj.

[7] OASIS, "Web Services Business Process Execution Language Version 2.0, Working Draft," http://www.oasis-open.org/committees/download.php/10347/wsbpel-specification-draft-120204.htm.

[8] Peltz, C., *Web services orchestration and choreography*, Web Services Journal (2003).

[9] W3C, "Web Services Choreography Description Language Version 1.0," http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217.

[10] W3C, "Web Services Description Language (WSDL) 1.1," http://www.w3.org/TR/wsdl.