A tool for rapid development of WS-BPEL applications[*]

Luca Cesari, Alessandro Lapadula, Rosario Pugliese and Francesco Tiezzi
Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze
cesari.luca@gmail.com, {lapadula,pugliese,tiezzi}@dsi.unifi.it

ABSTRACT

We present B*lite*C, a software tool we have developed for supporting a rapid and easy development of WS-BPEL applications. B*lite*C translates service orchestrations written in B*lite*, a formal language inspired to but simpler than WS-BPEL, into executable WS-BPEL programs. We illustrate our approach by means of an example borrowed from the official specification of WS-BPEL.

Categories and Subject Descriptors

D.2.2 [**Software engineering**]: Design Tools and Techniques—*Computer-aided software engineering*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Syntax Semantics*; D.3.4 [**Programming Languages**]: Processors—*Compilers Parsing*

General Terms

Languages, Design

Keywords

Service-oriented architectures, Web services, Compilers

1. INTRODUCTION

In recent years, there has been an ever increasing acceptance of WS-BPEL [15] as a standard language for orchestration of *web services*, one of the most successful and well-developed implementations of *Service-Oriented Computing* (SOC). However, designing and developing WS-BPEL applications is a difficult and error-prone task. The language has an XML syntax which makes it awkward writing WS-BPEL code by using standard editors. Therefore, many companies (among which e.g. Oracle and Active Endpoints) have equipped their WS-BPEL engines with graphical designers. Such tools are certainly suitable to develop simple business processes, but turn out to be cumbersome and ineffective when dealing with more complex applications. Further difficulties derive from the fact that WS-BPEL is equipped with such intricate features as concurrency, multiple service instances, message correlation, long-running business transactions, termination and compensation handlers. Most of all, WS-BPEL comes without a formal semantics and its specification document, written in 'natural' language, contains a fair number of acknowledged ambiguous features that may give rise to different interpretations. These ambiguities, as shown in [12], have led to engines implementing different semantics and, hence, have undermined portability of WS-BPEL programs across different platforms. Portability is further compromised since the deployment procedure of WS-BPEL programs is not standardised. In fact, to execute a WS-BPEL program, besides the associated WSDL [10] document that describes the program's public interfaces, different engines require different (and not integrable) *process deployment descriptors*, i.e. sets of configuration files that describe how the program should be deployed.

To overcome these difficulties, we have developed B*lite*C, a software tool that accepts as an input a specification written in the lightweight orchestration language B*lite* [12] and returns the corresponding WS-BPEL program together with the associated WSDL and deployment descriptor files.

B*lite* is closely inspired to WS-BPEL. It is the result of a tension between handiness and expressiveness. While the set of WS-BPEL constructs is not intended to be a minimal one, to keep the language manageable, the design of B*lite* only retains the core features of WS-BPEL. It follows that B*lite* is simpler and more compact than WS-BPEL, although it maintains the same descriptive power. Using B*lite* for initially specifying a service orchestration offers some significant advantages. From the one hand, the B*lite* textual notation is certainly more manageable than those, also graphical ones, proposed for WS-BPEL. From the other hand, B*lite* is equipped with a formal operational semantics that clarifies all ambiguous and intricate aspects of WS-BPEL.

B*lite*C further simplifies the programmers work by automatizing the deployment procedure. In fact, the returned files are properly packaged to be immediately executable in a WS-BPEL engine. Currently, these packages are intended to be deployed on ActiveBPEL [2] that, according to [12], is one of the freely available WS-BPEL engines that better complies with the WS-BPEL specification. Anyway, B*lite*C has been designed so that the generation of deployment descriptors for different engines can be easily integrated, and we plan to enable it to produce packages also for two other

[*]This work has been supported by the EU project SENSORIA, IST-2005-016004.

freely available engines, namely Oracle BPEL Process Manager [1] and Apache ODE [4]. Of course, to preserve the semantics of the original B*lite* programs, one has to study the inner implementation of every supported engine and to define a customized translation. Unfortunately, since no engine has a formal description of its behaviour, this study has to be carried out by means of experimental tests and, most of all, no formal proof of semantics preservation can be done. We also plan to integrate in B*lite*C a better support for data manipulation based on XPath.

Related work. The aim of facilitating the development of WS-BPEL applications is shared also by the several graphical editors that permit designing WS-BPEL processes, among which we mention the designers embedded in Oracle BPEL Process Manager [1], Intalio|Designer [6], ActiveVOS Designer [3], and Eclipse BPEL designer [5]. Although their use is quite intuitive, developing large processes by using them can be awkward and annoying compared to the more classic textual approach. Moreover, graphical designers have a significant negative impact on performance, since they usually are plugins of heavy software development environments such as JDeveloper and Eclipse. [13] presents a tool that produces WS-BPEL processes starting from UML-based representations of SOC applications. Due to the use of graphical representations, also this tool suffers from the problems previously mentioned. Furthermore, it generates only non-executable processes (binding and deployment details must be added by editing the generated files). [14] proposes a different approach to develop SOC applications that still relies on a formal language. However, input programs are directly executed in a purposely developed engine, rather than being translated into and deployed as WS-BPEL processes.

2. PROGRAMMING SERVICES IN BLITE

A B*lite program* accepted by B*lite*C is composed of a B*lite* specification and a declarative part. The former focusses on the behavioural aspects of the orchestration, while the latter provides the implementation details (e.g. types, addresses, bindings, ...) that are necessary to deploy and execute the corresponding WS-BPEL program.

2.1 Blite specification

The syntax of B*lite* accepted by B*lite*C is given in Figure 1. Services are *structured activities* built from *basic activities*, i.e. service invocation, service request processing, assignment, empty activity, fault generation and instance forced termination, by exploiting operators for conditional choice, iteration, sequential composition, parallel composition, pick and scope. A scope activity groups a primary activity A together with a fault handling activity A_f and a compensation activity A_c. *Start activities* are structured activities that initially can only execute receive activities. Sequence has higher priority (i.e. bind more tightly) than parallel composition and pick. Moreover, fault and compensation activities may be omitted from a scope construct, in which case they are intended to be `throw` and `empty`, respectively.

Notation `<·>` stands for tuples of objects, e.g. `<x_1,...,x_n>` denotes a tuple of variables (variables in the same tuple must be pairwise distinct). Partner links *pl* can be either of the form `<partner>` or of the form `<partner`$_1$`,partner`$_2$`>`. Indeed, in one-way interactions a partner link indicates a single partner because one of the parties provides all the invoked operations. Instead, in request-response interactions, partner links indicate two partners because the requesting partner must provide a callback operation used by the receiving partner to send notifications. Service partners used for receiving messages must be known at design-time, while the partners used to send messages in reply may be dynamically determined.

Besides asynchronous invocation, WS-BPEL also provides a construct for synchronous invocation of remote services. This construct forces the invoker to wait for an answer by the invoked service, that indeed performs a pair of operations *receive–reply*. In B*lite*, this behaviour is rendered in terms of a pair of activities *invoke–receive* executed by the invoker and a pair of activities *receive–invoke* executed by the invoked service.

Data can be shared among different activities through *shared variables* (ranged over by `x`, `x_1`, ...). The manipulable values are boolean, integer numbers (ranged over by *int*), strings (as usual, written within double inverted commas), partner links, and literals (defined in the declarative part and denoted by putting the symbol `$` in front of the corresponding identifier). *Expressions* combine values and variables by means of boolean, arithmetic, comparision and string operators.

B*lite* specifications are finite compositions of *definitions* (that assign names to B*lite* terms), containing at most one *deployment* definition. A deployment associates a *correlation set*, namely a (possibly empty) set of correlation variables, to a *service*. A service provides a 'top-level' scope (i.e. a scope that cannot be compensated) and offers a choice of alternative receives among multiple start activities.

We refer the interested reader to [12] for a formal account of the B*lite* operational semantics.

2.2 Declarative part

The declarative part of a B*lite* program specifies configuration data necessary to properly translate the B*lite* specification into an executable WS-BPEL program. Notably, B*lite*C requires the user to insert only the strictly necessary data. The declarations must be included within `<?blm` and `?>`, and can occur in any position within a B*lite* program.

A declarative part has the following form:

```
<?blm
  ADDRESSES { myns => " base_for_namespaces ";
              myaddress => " base_for_service_url "; }
    IMPORTS { associations prefix => " url "; }
  VARIABLES { variable and message declarations }
   LITERALS { associations literal_name => [[literal_code]]; }
  PARTNERLINKS { partner link type declarations }
?>
```

where blocks `ADDRESSES` and `VARIABLES` are mandatory, while the other ones can be omitted.

Within the `ADDRESSES` block the user has to specify the base for the namespaces used inside the generated files (after the keyword `myns`) and the base for the address where the new service will be hosted (after the keyword `myaddress`).

To define a service orchestration it is often necessary to import data (e.g. type declarations) from documents (e.g. WSDL files) associated to other services. To this aim, the user can specify the addresses of the documents to be imported within the `IMPORTS` block, by associating to each imported document a namespace prefix that will be used in the subsequent declarations to refer to it. Notably, definitions belonging to standard namespaces (e.g. `http://www.w3.org/2001/XMLSchema`) are automatically imported and, hence, do not require any declaration.

```
    b ::=                                                                       (basic activities)
          inv pl op <x_1,...,x_n>  |  rcv pl op <x_1,...,x_n>                     (invoke, receive)
        | x := e  |  empty  |  throw  |  exit                                     (assign, empty, throw, exit)

   pl ::= <partner>  |  <partner1,partner2>                                     (partner links)

    e ::=                                                                       (expressions)
          e1 | e2  |  e1 & e2  |  ! e  |  TRUE  |  FALSE                          (boolean operators)
        | e1 + e2  |  e1 - e2  |  e1 * e2  |  e1 / e2  |  int                     (arithmetic operators)
        | e1 >= e2  |  e1 <= e2  |  e1 > e2  |  e1 < e2  |  e1 = e2  |  e1 != e2  (comparison operators)
        | e1 . e2  |  "string"                                                    (string operators)
        | x  |  pl  |  $literal_name                                              (variable, partner link, literal)

    a ::=                                                                       (structured activities)
          b  |  if (e) {a1} {a2}  |  while (e) {a}                                (basic, conditional, iteration)
        | seq a1 ;... ; an qes  |  flw a1 |... | an wlf                           (sequence, parallel)
        | [ A @ Af * Ac ]  |  pck rcv pl1 op1 <x_1,...,x_k> ; a1                  (scope, pick)
                                +... + rcv pln opn <x_1,...,x_h> ; an kcp

    r ::=                                                                       (start activities)
          rcv pl op <x_1,...,x_n>  |  seq r; a1 ;... ; an qes  |  flw r1 |... | rn wlf   (receive, sequence, parallel)
        | [ R @ Af * Ac ]  |  pck rcv pl1 op1 <x_1,...,x_k> ; a1                  (scope, pick)
                                +... + rcv pln opn <x_1,...,x_h> ; an kcp

    s ::= [ R @ Af ]          d ::= { S }{x_1,...,x_n}                          (services, deployments)

    A ::= a  |  i             R ::= r  |  i           S ::= s  |  i             (activities/services identifiers)

  def ::= i := a;; def  |  i := r;; def  |  i := s;; def  |  i := d;;           (definitions)
```

Figure 1: Syntax of B*lite*

B*lite* variables are untyped, while WS-BPEL ones must be typed. Therefore, to enable an automated translation, the user has to declare the type of the variables (both local variables and messages) within the `VARIABLES` block. Local variables, that can be used to temporarily store data and manipulate them, are declared by associations of the form `x => XML_Schema_type;` (e.g. `x_shipped => xsd:integer;`). Messages, that are tuples of variables used as either sending source or receiving target, can be declared in two ways:

- by using an imported message type, e.g. in `<x_count,x_id> => bck:number;` the message composed of variables `x_count` and `x_id` is typed as `number`, that is defined in the (WSDL) document identified by the namespace prefix `bck` (defined in the `IMPORTS` block);
- by generating a new message type, e.g. in

  ```
  <x_id,x_c,x_items> => gen:shipOrder,
                        <id,shipComplete,items>,
                        <xsd:int,xsd:int,xsd:int>;
  ```

  message `<x_id,x_c,x_items>` is typed as `shipOrder`, that defines messages composed of three integer parts, `id`, `shipComplete` and `items`. The namespace prefix `gen` indicates that the type must be generated.

In a WS-BPEL program, literals (i.e. constant values) can be directly assigned to variables. Instead, in a B*lite* program, for the sake of readability, literals must be declared within the `LITERALS` block, e.g.

```
reqLit => [[ <weat1:GetCityForecastByZIP xmlns:weat1=
                 "http://ws.cdyne.com/WeatherWS/">
               <weat1:ZIP>10036</weat1:ZIP>
            </weat1:GetCityForecastByZIP> ]];
```

and, then, can be assigned to a variable by using the associated name, e.g. `x_weat := $reqLit;`.

Similarly, also partner links are typed in WS-BPEL and untyped in B*lite*. Therefore, except for the partner links used by the new process to interact with its clients, that are automatically generated and typed by B*lite*C, the type of the other partner links must be defined within the `PARTNERLINKS` block. Each declaration has the following form:

```
PARTNERLINK { TYPE => partner_link_type;
              MY_ROLE partner1 => port_type1;
              PARTNER_ROLE partner2 => port_type2; }
```

where the association for `MY_ROLE` can be omitted whenever the process does not play any role. Moreover, to de-couple the B*lite* operation names from the WS-BPEL ones, associations of the form `(bliteOperation => wsbpelOperation)` may be specified after the definitions of the two roles.

3. BLITEC: FROM BLITE TO WS-BPEL

B*lite*C[1] is developed in Java[2] to guarantee its portability across different platforms, to exploit the well-established libraries for generating parsers and for manipulating XML documents, and because Java is the reference language for the applications designed around WS-BPEL. Besides the standard Java libraries, we have used JDOM [8] for creating and managing XML documents, JavaCC [7] for generating the parsers that validate the input documents, and JJTree[3] for allowing the parsers to build parse trees (already arranged to support the Visitor design pattern [11]).

B*lite*C is composed of five main components:

- *Mapper* parses the declarative part of the input B*lite* program and initializes a map that associates each declared object (e.g. partner link, literal, variable, ...) to its name;
- B*lite parser* analyzes the B*lite* specification within the input program, completes the map created by Mapper and creates the parse tree of the B*lite* specification;
- WS-BPEL and WSDL *generators* use the data produced by the above components to generate a WS-BPEL process and the associated WSDL document;
- *Deployer* generates the deployment descriptor and packages all created documents into a deployable file; it is the only 'engine-dependent' component.

We now provide some insights about the transformation of B*lite* constructs into WS-BPEL activities. Communication activities, invokes and receives, are translated in a different way depending on their arguments and their position in the

[1]B*lite*C is a free software; it can be downloaded from `http://rap.dsi.unifi.it/blite` and redistributed and/or modified under the terms of the GNU GPL.
[2]JRE and JDK version 6.
[3]JJTree is included within JavaCC.

Table 1: Mapping of the receive activity

B*lite*	WS-BPEL

pck ...
 rec *pl* op <x1,...,xn>...
kcp
	<onMessage
	   partnerLink="pl"
	   operation="op"
	   variable="x" />

inv <p,p'> op <y1,...,yn>;
rec <p'> op <x1,...,xn>
	<invoke partnerlink="pl"
	   operation="op"
	   inputVariable="y"
	   outputVariable="x" />

rcv *pl* op <x1,...,xn>	<receive partnerLink="pl"
	         operation="op"
	         variable="x" />

Table 2: Mapping of structured activities

B*lite*	WS-BPEL
if (e) {a_1} {a_2}	<if> <condition> e </condition> a_1 <else> a_2 </else> </if>
while (e) {a}	<while> <condition> e </condition> a </while>
seq a_1 ; ... ; a_n qes	<sequence> a_1 ... a_n </sequence>
flw a_1 | ... | a_n wlf	<flow> a_1 ... a_n </flow>
pck a_1 + ... + a_n kcp	<pick> a_1 ... a_n </pick>

[a @ a_f * a_c]	<scope>
	  <faultHandlers>
	    <catchAll>
	      <sequence>
	         <compensate/> a_f
	      </sequence>
	    </catchAll>
	  </faultHandlers>
	  <compensationHandler>
	    a_c
	  </compensationHandler>
	  a
	</scope>

code. For example, as shown in Table 1, if a receive activity is positioned within a `pck` construct it is translated as an `<onMessage>` activity; if it is positioned after an invoke (in case of a request-response interaction) it is translated as a synchronous `<invoke>`; otherwise, it is simply translated as a `<receive>`. If a receive is a start activity, to allow the process to be instantiated, the `createInstance` attribute must be set to `yes`. Moreover, if some correlation variables are involved, the corresponding correlation set (whose declaration is generated during the translation of the deployment term) must be specified as further argument of the `<receive>` activity. The correlation attributes `initiate` and `pattern` are specified according to the type of the interaction.

The invoke activity is translated similarly; in particular, when it is used in a request-response interaction to send the response, it is translated as a `<reply>` activity. The translation of the remaining basic activities is straightforward. Also the translation of the structured activities does not require significant effort, as shown in Table 2. Finally, a B*lite* service is rendered as a scope, where the compensation handler is removed and the tag `<scope>` is replaced by `<process>`.

4. BLITEC AT WORK

We show an application of B*lite*C to a scenario built upon the shipping service drawn from the official specification of WS-BPEL [15, Sect. 15.1]. The example allows us to illustrate many language features, including correlation sets, shared variables, control flow structures, and fault handling.

The shipping service handles the shipment of orders. From the service point of view, orders are composed of a number of items. The service offers two types of shipments: shipments where the items are held and shipped together and shipments where the items are shipped piecemeal until the order is fulfilled. A skeleton description follows:

```
receive shipOrder
if (shipComplete) then send shipNotice
else
    itemsShipped := 0
    while (itemsShipped < itemsTotal) do
      itemsCount := opaque // non-deterministic assignment
                           // corresponding e.g. to interaction
                           // with a back-end system
      send shipNotice
      itemsShipped = itemsShipped + itemsCount
```

To generate an executable process, we have to replace the opaque assignment with an invocation to the *back-end service*, that in B*lite* is rendered as follows:

```
s_backend ::=
  [ seq rcv <p_backend, x_client> o_num <x_id>;
        rcv <p_human> o_humanInteraction <x_id,x_num>;
        inv <x_client> o_num <x_num,x_id> qes ];;

backend_service ::= {s_backend}{x_id};;
```

Its behaviour is very simple: the process gets instantiated by the shipping service by invoking the operation `o_num`; then, the created instance waits for an integer number (representing the quantity of available items) provided by a human actor along the operation `o_humanInteraction` and concludes by sending the number back to the shipping service. The fact that the invoke activity used for the reply is performed along the same operation of the initial receive indicates that the two activities form a synchronous request-response interaction, hence the invoke will be translated into a `<reply>` activity. The order identifier, stored in `x_id`, is used as a correlation value.

Since the above service does not need to invoke other services, only its address and variables are explicitly declared:

```
<?blm
 ADDRESSES { myns => "http://example/backendService";
   myaddress => "http://XXX:8080/active-bpel/services"; }
 VARIABLES { <x_id> => gen:id,<id>,<xsd:int>;
      <x_id,x_num> => gen:human,<id,num>,<xsd:int,xsd:int>;
      <x_num,x_id> => gen:number,<num,id>,<xsd:int,xsd:int>; }
?>
```

To compile this B*lite* program, we have to save the above code into a file (named, e.g., `backend_service.bl`) and execute the following command `java -jar blite.jar backend_service.bl`. This way, the file `backend_serviceProcess.bpr`, which is a WS-BPEL package directly deployable into ActiveBPEL, is generated. To deploy the file, it is sufficient to move it into the engine's deployment directory `bpr`. Then, to check that the deploy succeeded, we can use the ActiveBPEL's administration console that can be accessed by using any browser at the address `http://XXX:8080/BpelAdmin` (where `XXX` is the server's address where the ActiveBPEL engine is running). By selecting `Deployed Processes` from the menu on the left-hand side, we obtain the list of the deployed processes among which `backend_serviceProcess` should appear. Now, by selecting `Deployed services`, we can retrieve the URLs of the two WSDL files corresponding to the partner links for interacting with the service:

```
http://XXX:8080/active-bpel/services/p_backendService?wsdl

http://XXX:8080/active-bpel/services/p_humanService?wsdl
```

Finally, by using a tool for automatic generation of web service requests (e.g. soapUI [9]), we can invoke the service by sending two SOAP messages: the first message creates an instance for the order identified by `1234`, while the second message indicates that seven items for that order are available for shipping. After the first message is sent, by selecting `Active Processes` from the console menu, we can verify that a back-end service instance has been created and its status is `Running`. Then, after the other message is sent, we get in response the pair of integers `<7,1234>` and the instance status changes to `Completed`.

The *shipping service* in B*lite* is defined as:

```
a_ship ::= seq
            x_shipped := 0;
            while (x_shipped < x_items) {
             seq
              inv <backend,cb_backend> o_num <x_id>;
              rcv <cb_backend> o_num <x_count,x_id>;
              if (x_count <= 0)
                 { throw }
                 { seq
                    inv <x_cust> o_notice <x_id,x_count>;
                    x_shipped := x_shipped + x_count
                   qes }
             qes }
            qes ;;

a_err ::= seq
            x_sorry:="Sorry, the required item is out of stock ";
            inv <x_cust> o_err <x_id,x_sorry>
           qes;;

s_ship ::=
  [ seq
    rcv <p_ship, x_cust> o_req <x_id,x_c,x_items>;
    if (x_c > 0)
       { inv <x_cust> o_notice <x_id,x_items> }
       { [a_ship @ a_err] }
   qes ];;

shipping_service ::= {s_ship}{x_id};;

<?blm
 ADDRESSES { myns => "http://example";
   myaddress =>"http://XXX:8080/active-bpel/services"; }
 IMPORTS { bck => "http://example/backendService/
                            backend_service.wsdl"; }
 VARIABLES {
   <x_id,x_c,x_items> => gen:shipOrder,
                         <id, shipComplete, items>,
                         <xsd:int, xsd:int, xsd:int>;
       <x_id,x_items> => gen:shippingNoticeMsg,
                         <id, items>, <xsd:int,xsd:int>;
       <x_id,x_count> => gen:shippingNoticeMsg;
       <x_id,x_sorry> => gen:shippingErrorMsg,
                         <id, errorMsg>,
                         <xsd:int,xsd:string>;
               <x_id> => bck:id;
       <x_count,x_id> => bck:number;
            x_shipped => xsd:integer; }
 PARTNERLINKS {
    PARTNERLINK { TYPE => bck:clientPLT;
        PARTNER_ROLE backend => bck:p_backendPT; } }
?>
```

Here, after the invocation of the back-end service, the returned number (stored in `x_count`) is checked: a number less than or equal to `0` means that the required item is out of stock and, hence, a fault is raised by means of the `throw` activity. The fault will be caught and handled by the fault handler `a_err`, that will send an error message to the client.

Finally, we report below a (dummy) *client service*:

```
 s_shipClient ::=
   [ seq
      rcv <p_init,y_clt> o_init <y_id,y_c, y_items>;
      y_resp := "ORDER: ". y_id ." BEGIN ";
      inv <ship_srv,cust> o_req <y_id,y_c, y_items>;
      y_shipped := 0;
      while(y_shipped < y_items) {
         pck
           rcv <cust> o_notice <y_id,y_count>;
           seq
             y_shipped := y_shipped + y_count;
             y_resp := y_resp ."NOTICE: sent items=".y_count."; "
           qes
           +
           rcv <cust> o_err <y_id, y_sorry>;
           seq
            y_shipped := y_items;
            y_resp := y_resp ."ERROR: ". y_sorry
           qes
          kcp };
      y_resp := y_resp . " END";
      inv <y_clt> o_init <y_id,y_resp>
     qes ];;
 shipping_client ::= {s_shipClient}{y_id};;

<?blm
  ADDRESSES { myns => "http://example";
     myaddress =>"http://XXX:8080/active-bpel/services"; }
  IMPORTS { shs => "http://example/shipping_service.wsdl"; }
  VARIABLES { <y_id, y_c, y_items> => shs:shipOrder;
              <y_id,y_sorry> => shs:shippingErrorMsg;
              <y_id,y_count> => shs:shippingNoticeMsg;
                   y_shipped => xsd:integer;
               <y_id,y_resp> => gen:response,
                                <id,resp>,
                                <xsd:int,xsd:string>; }
  PARTNERLINKS {
     PARTNERLINK { TYPE => shs:custPLT;
           MY_ROLE cust => shs:x_custPT;
           PARTNER_ROLE ship_srv => shs:p_shipPT; } }
?>
```

This client receives a shipping request and forwards it to the shipping service. Then, it waits all response messages, stores them in a string variable (i.e. `y_resp`) and sends back the string to the invoker.

Since the shipping service requires configuration data provided by the back-end service and, similarly, the client needs data from the shipping service, to successfully compile the above B*lite* programs, we must strictly follow their order of presentation. Once the programs have been deployed, if we invoke the operation `o_init` provided by the client service by sending the request `<1234,0,7>` (i.e. we require 7 items shipped piecemeal with order identifier `1234`) and manually specify that the shipment is divided in two packages of 3 and 4 items, we will get back the string:

```
ORDER: 1234 BEGIN NOTICE: sent items=3; NOTICE: sent items=4; END
```

5. REFERENCES

[1] Oracle BPEL Process Manager 10.1.3, December 2007.

[2] ActiveBPEL 5.0.2, May 2008.

[3] ActiveVOS Designer 5.0.2, June 2009.

[4] Apache ODE 1.3.3, August 2009.

[5] Eclipse BPEL project 0.4.0, May 2009.

[6] Intalio|Designer Community Ed. 6.0.1, August 2009.

[7] JavaCC 4.2, April 2009.

[8] JDOM 1.1, April 2009.

[9] soapUI 2.5.1, February 2009.

[10] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. TR, W3C, 2001.

[11] G. Erich, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[12] A. Lapadula, R. Pugliese, and F. Tiezzi. A formal account of WS-BPEL. *COORDINATION*, *LNCS* 5052, pp. 199–215. Springer, 2008.

[13] P. Mayer, A. Schroeder, and N. Koch. Mdd4soa: Model-driven service orchestration. *EDOC*, pp. 203–212. IEEE, 2008.

[14] F. Montesi, C. Guidi, and G. Zavattaro. Composing Services with JOLIE. *ECOWS*, pp. 13–22. IEEE, 2007.

[15] OASIS WSBPEL TC. Web Services Business Process Execution Language Version 2.0. TR, April 2007.