A tool for rapid development of WS-BPEL applications

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

ABSTRACT

WS-BPEL is imposing itself as a standard for orchestration of web services. However, there are still some well-known difficulties that make programming in WS-BPEL a tricky task. In this paper, 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 readily executable WS-BPEL programs. We illustrate our approach by means of a few practical programming examples.

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*

Keywords

Service-Oriented Computing, Web services, Compilers

1. INTRODUCTION

In recent years, there has been an ever increasing acceptance of WS-BPEL [30] as a standard language for orchestration of *web services*, one of the most successful and well-developed implementations of the *Service-Oriented Computing* (SOC) paradigm. However, designing and developing WS-BPEL applications is a difficult and error-prone task. The language has an XML syntax which makes 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 programming 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 have led to engines implementing different semantics (see [25]) and, hence, have undermined portability of WS-BPEL programs across different platforms. Therefore, many research efforts have been devoted to provide WS-BPEL with a formal semantic (see, e.g., [27, 18, 33, 31, 24, 26, 20, 21]), although most of them do not deal with the complete language. Finally, the deployment procedure of WS-BPEL programs is not standardised, which further compromises portability. In fact, to execute a WS-BPEL program, besides the associated WSDL [17] 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* [25] 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, the design of B*lite*, to keep the language manageable, 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, B*lite* textual notation is certainly more manageable than those, possibly graphical, notations 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. Of course, to preserve such semantics on different WS-BPEL engines, the translation of B*lite* programs into WS-BPEL ones has to be properly targeted to each specific engine.

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

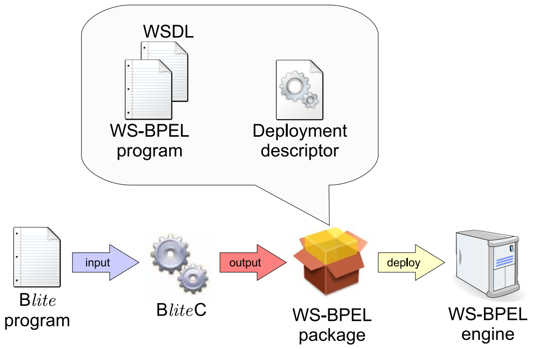


Figure 1: B*lite*C workflow

to be deployed on ActiveBPEL [6] that, according to [25], is one of the freely available WS-BPEL engines that better complies with the WS-BPEL specification. The workflow of use of B*lite*C is graphically depicted in Figure 1.

The rest of the paper is organized as follows. Section 2 summarizes WS-BPEL, while Section 3 introduces the syntax of B*lite* accepted by our tool. Section 4 presents B*lite*C and explains the correspondence between B*lite* constructs and WS-BPEL activities. Section 5 illustrates B*lite*C at work on some practical examples, one of which is borrowed from the official specification of WS-BPEL. Section 6 reviews more closely related work and hints at some future work.

2. AN OVERVIEW OF WS-BPEL

WS-BPEL is essentially a linguistic layer on top of WSDL for describing the structural aspects of web service orchestration. In WS-BPEL, the logic of interaction between a service and its environment is described in terms of structured patterns of communication actions composed by means of control flow constructs that enable the representation of complex structures. Orchestration exploits state information that is maintained through shared variables and managed through message correlation. For the specification of orchestration, WS-BPEL provides many different *basic* and *structured activities*.

The following basic activities are provided: `<receive>` and `<reply>`, to enable web service one-way and request-response operations; `<invoke>`, to invoke web service operations; `<wait>`, to delay execution for some amount of time; `<assign>`, to update the values of variables with new data; `<throw>`, to signal internal faults; `<exit>`, to immediately end a service instance; `<empty>`, to do nothing; `<compensate>` and `<compensateScope>`, to invoke compensation handlers; `<rethrow>`, to propagate faults; `<validate>`, to validate variables; and `<extensionActivity>`, to add new activity types.

The structured activities describe the control flow logic of a business process by composing basic and/or structured activities recursively. The following structured activities are provided: `<sequence>`, to execute activities sequentially; `<if>`, to execute activities conditionally; `<while>` and `<repeatUntil>`, to repetitively execute activities; `<flow>`, to execute activities in parallel; `<pick>`, to execute activities selectively; `<forEach>`, to (sequentially or in parallel) execute multiple activities; and `<scope>`, to associate handlers for exceptional events to a primary activity.

The handlers within a `<scope>` can be of four different kinds: `<faultHandler>`, to provide the activities in response to faults occurring during execution of the primary activity; `<compensationHandler>`, to provide the activities to compensate the successfully executed primary activity; `<terminationHandler>`, to control the forced termination of the primary activity; and `<eventHandler>`, to process message or timeout events occurring during execution of the primary activity. If a fault occurs during execution of a primary activity, the control is transferred to the corresponding fault handler and all currently running activities inside the scope are interrupted immediately without involving any fault/-compensation handling behaviour. If another fault occurs during a fault/compensation handling, then it is re-thrown, possibly, to the immediately enclosing scope. Compensation handlers attempt to reverse the effects of previously successfully completed primary activities (scopes) and have been introduced to support Long-Running (Business) Transactions (LRTs). Compensation can only be invoked from within fault or compensation handlers starting the compensation either of a specific inner (completed) scope, or of all inner completed scopes in the reverse order of completion. The latter alternative is also called the *default* compensation behaviour. Invoking a compensation handler that is unavailable is equivalent to perform an empty activity.

A WS-BPEL program, also called *(business) process*, is a `<process>`, that is a sort of `<scope>` without compensation and termination handlers.

WS-BPEL uses the basic notion of *partner link* to directly model peer-to-peer relationships between services. This relationship is expressed at the WSDL level by specifying the roles played by each of the services in the interaction. However, the information provided by partner links is not enough to deliver messages to a business process. Indeed, since multiple instances of a same service can be simultaneously active because service operations can be independently invoked by several clients, messages need to be delivered not only to the correct partner, but also to the correct instance of the service that the partner provides. To achieve this, WS-BPEL relies on the business data exchanged rather than on specific mechanisms, such as *WS-Addressing* [22] or low-level methods based on SOAP headers. In fact, WS-BPEL exploits *correlation sets*, namely sets of *correlation variables* (called *properties* in WS-BPEL jargon), to declare the parts of a message that can be used to identify an instance. This way, a message can be delivered to the correct instance on the basis of the values associated to the correlation variables, independently of any routing mechanism.

We end this section by showing an auction service described in the official specification of WS-BPEL [30, Sect. 15.4]. This example will allow us to illustrate most of the language features, including correlation sets, shared variables, control flow structures, asynchronous communication and multiple start activities, and, through its implementation in B*lite* presented in Section 5.3, will permit a rough comparison between the two languages.

The auction service collects information from a seller and a buyer about a concluded auction, reports the auction result to an auction registration service, and then communicates the registration result to the seller and the buyer. The auction house process may be instantiated either by receiving the seller information or by receiving the buyer information. Indeed, the process is able of receiving the seller and buyer requests in a statically unpredictable order and in such a way that the first incoming message triggers the creation of a process instance which the subsequent request is delivered to. This requires the two starting receive activities to share a correlation set, which will be initialized with an auction identifier that the seller and the buyer have to provide when sending their requests. The auction house process passes the auction identifier to the auction registration service that, in its turn, returns the identifier in its answer to locate the proper process instance.

We report below the corresponding WS-BPEL program, where to make the reading of the code easier, we have omitted irrelevant details[1] and highlighted the basic activities `receive`, `invoke` and `assign`.

```
<process name="auctionService" ... >
<partnerLinks> ... </partnerLinks>
<variables> ... </variables>
<correlationSets>
  <correlationSet name="auctionIdentification"
                  properties="as:auctionId" />
</correlationSets>
<sequence>
  <flow>
    <receive name="acceptSellerInformation"
             partnerLink="seller"
             portType="as:sellerPT"
             operation="submit"
             variable="sellerData"
             createInstance="yes">
      <correlations>
        <correlation set="auctionIdentification"
                     initiate="join" />
      </correlations>
    </receive>
    <receive name="acceptBuyerInformation"
             partnerLink="buyer"
             portType="as:buyerPT"
             operation="submit"
             variable="buyerData"
             createInstance="yes">
      <correlations>
        <correlation set="auctionIdentification"
                     initiate="join" />
      </correlations>
    </receive>
  </flow>
  <assign>
    <copy>
      <from>
        ... http://example.com/auction/RegistrationService ...
      </from>
      <to partnerLink="auctionRegistrationService" />
    </copy>
    <copy>
      <from partnerLink="auctionRegistrationService"
            endpointReference="myRole" />
      <to>$auctionData.auctionHouseEndpointReference</to>
    </copy>
    <copy>
      <from>$sellerData.auctionId</from>
      <to>$auctionData.auctionId</to>
    </copy>
    <copy>
      <from>1</from>
      <to>$auctionData.amount</to>
    </copy>
  </assign>
  <invoke name="registerAuctionResults"
          partnerLink="auctionRegistrationService"
          portType="as:auctionRegistrationPT"
          operation="process"
          inputVariable="auctionData" />
  <receive name="receiveAuctionRegistrationInformation"
           partnerLink="auctionRegistrationService"
           portType="as:auctionRegistrationAnswerPT"
           operation="answer"
           variable="auctionAnswerData">
    <correlations>
      <correlation set="auctionIdentification" />
    </correlations>
  </receive>
  <flow>
    <sequence>
      <assign>
        <copy>
          <from>$sellerData.endpointReference</from>
          <to partnerLink="seller" />
        </copy>
        <copy>
          <from><literal>Thank you!</literal></from>
          <to>$sellerAnswerData.thankYouText</to>
        </copy>
      </assign>
      <invoke name="respondToSeller"
              partnerLink="seller"
              portType="as:sellerAnswerPT"
              operation="answer"
              inputVariable="sellerAnswerData" />
    </sequence>
    <sequence>
      <assign>
        <copy>
          <from>$buyerData.endpointReference</from>
          <to partnerLink="buyer" />
        </copy>
        <copy>
          <from><literal>Thank you!</literal></from>
          <to>$buyerAnswerData.thankYouText</to>
        </copy>
      </assign>
      <invoke name="respondToBuyer"
              partnerLink="buyer"
              portType="as:buyerAnswerPT"
              operation="answer"
              inputVariable="buyerAnswerData" />
    </sequence>
  </flow>
</sequence>
</process>
```

Notice that the buyer and the seller provide their endpoint references for the auction house process to respond properly in an asynchronous way. For similar reasons, the auction house process provides its own endpoint reference to the auction registration service.

3. 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.

3.1 Blite specification

B*lite* [25] is a prototypical orchestration language whose design is closely inspired to WS-BPEL. To keep the language manageable, the design of B*lite* only retains the core features of WS-BPEL. In fact, some aspects have been intentionally left out, including timeouts, synchronization dependencies within flow activities, event and termination handlers. Moreover, B*lite* only provides a simplified form of fault and compensation handling and only supports unnamed faults and the default compensation mechanisms.

B*lite* provides a formal description of service deployments by only keeping relevant implementation details. Thus, the roles played by service partners in a service interaction are explicitly indicated by *partner links* and *partners*, while such aspects as physical *service binding* (necessary to generate

[1] The fully detailed version of the WS-BPEL process and the associated WSDL document can be found in [30, Sect. 15.4].

the associated WSDL documents and deployment descriptors) are abstracted away and dealt with separately in the declarative part.

The syntax of B*lite* accepted by B*lite*C is given in Figure 2. 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 `<partner1,partner2>`. Indeed, in one-way interactions a partner link indicates a single partner because one of the parties provides all the invoked operations. Instead, in asynchronous 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 activities *receive–reply*. In B*lite*, this behaviour is rendered in terms of a pair of activities *invoke–receive* over the same operation executed by the invoker and a corresponding 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. Operators `set(x,"path")` and `get(x,"path")` can be used respectively in the left and right hand sides of an assignment to act on a specific element (indicated by `path`) of the XML message stored in the variable `x`. Both operators turn out to be quite useful for easily interacting with non-B*lite* services (see Section 5.5).

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 [25] for a formal account of the B*lite* operational semantics.

3.2 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*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_city => xsd:string;`). Messages, i.e. tuples of variables used as either sending source or receiving target, can be declared in two ways:

- by using an imported message type. For example, in `<x_auctionId,x_registrationId> => reg:regResp;` the message composed of variables `x_auctionId` and `x_registrationId` is typed as `regResp`, that is defined in the (WSDL) document identified by the namespace prefix `reg` (defined in the `IMPORTS` block);

```
   b ::=                                                                    (basic activities)
         inv pl op <x_1,...,x_n>  |  rcv pl op <x_1,...,x_n>                   (invoke, receive)
      |  x := e  |  set(x,"path") := e  |  y := get(x,"path")                  (assignments)
      |  empty  |  throw  |  exit                                              (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 (relational operators)
      |  e1 . e2  |  "string"                                                  (string operators)
      |  x  |  pl  |  $literal_name                                            (variable, partner link, literal)

   a ::=                                                                    (structured activities)
         b  |  if (e) {a1} else {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 2: Syntax of B*lite*

- by generating a new message type. For example, in

  ```
  <x_auctionId,x_creditCardNumberB,x_phoneNumber>
    => gen:buyerReq, <id,ccNum,phone>,
       <xsd:int,xsd:string,xsd:string>;
  ```

  message `<x_auctionId,x_creditCardNumberB,x_phoneNumber>` is typed as `buyerReq`, that defines messages composed of one integer and two string parts, `id`, `ccNum` and `phone`, respectively. The namespace prefix `gen` indicates that the type must be generated. If the type of a message part is an element type defined in an XML schema not generated by B*lite*C, the keyword `(El)` must precede the type. For example, in

  ```
  <num,scale,reqWeath>
    => gen:req,
       <init,scale,request>,
       <xsd:integer,xsd:string,(El)wth:GetCityWeatherByZIP>;
  ```

  the element type `GetCityWeatherByZIP` is defined in the block `types` of the WSDL document identified by `wth`.

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 first be declared within the `LITERALS` block, as e.g.

```
litConv => [[<tem:FahrenheitToCelcius
               xmlns:tem="http://webservices.daehosting.com/
                          temperature">
              <tem:nFahrenheit>0</tem:nFahrenheit>
            </tem:FahrenheitToCelcius>]];
```

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

Similarly, also partner links are typed in WS-BPEL and untyped in B*lite*. Therefore, except for the partner links used by the 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.

4. BLITEC: FROM BLITE TO WS-BPEL

In this section we present the architecture of B*lite*C and explain the correspondence between the B*lite* constructs and the WS-BPEL activities.

4.1 BliteC architecture

B*lite*C[2] is developed in Java[3] 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 [12] for creating and managing XML documents, JavaCC [11] for generating the parsers that validate the input documents, and JJTree[4] for allowing the parsers to build parse trees (already arranged to support the Visitor design pattern [19]).

The architecture of B*lite*C is graphically depicted in Figure 3. The tool is composed of five main components:

[2] 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 General Public License.

[3] JRE and JDK version 6.

[4] JJTree is included within JavaCC.

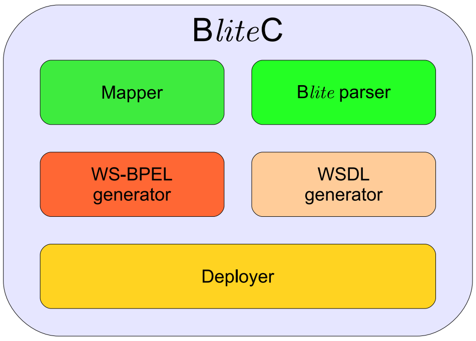


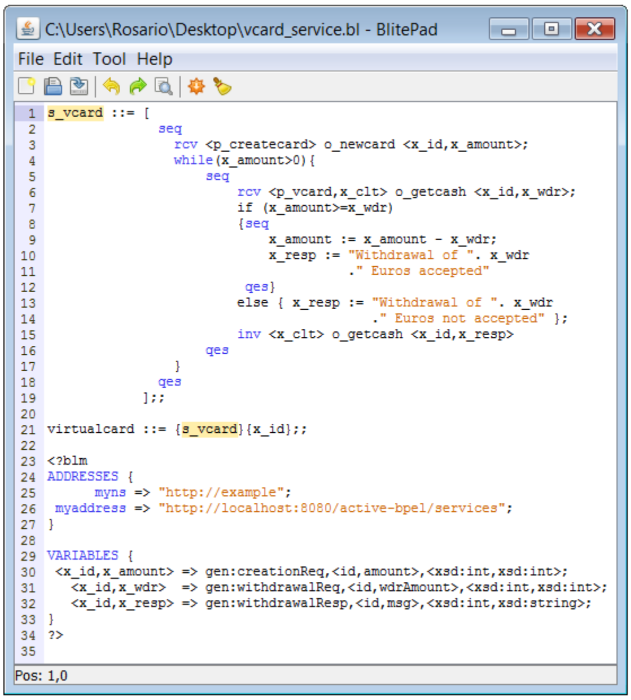
Figure 3: B*lite*C architecture

- *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.

Any text editor can be used to write B*lite* programs. Anyway, to simplify the task, we provide users with a customized version of jEdit[5] [2] equipped with specific features supporting programming in B*lite*, such as syntax highlighting, auto indentation and direct compiling. The files for the customization can be downloaded with the B*lite*C distribution archive. The main advantage of jEdit with respect to more professional development environments is that it is multi-platform and lightweight. We are also implementing a development environment with similar features written in Java. Figure 4 shows a screenshot of our environment. In addition to the functionalities of the customized version of jEdit, our dedicated environment also provides text auto-completion, highlight of search results, local deploy and undeploy.

4.2 From Blite to WS-BPEL

We now provide some insights about the transformation of B*lite* constructs into WS-BPEL activities. Since the same WS-BPEL program might have different behaviours on different engines [25], the translation described here is targeted to a specific engine, i.e. ActiveBPEL. If one want to produce packages intended to be executed by other WS-BPEL engines, the translation has possibly to be properly tailored. Since there is no precise description of the behaviour of the ActiveBPEL engine, it cannot be formally proved that the semantics of the WS-BPEL program resulting from a translation conforms to that of the original B*lite* program. However, since B*lite* is a 'sort of' lightweight variant of WS-BPEL, the translation we define is quite intuitive and direct, which makes us confident that the original semantics is obeyed. This is of course witnessed by all the experiments we have done.



```
s_vcard ::= [
              seq
                rcv <p_createcard> o_newcard <x_id,x_amount>;
                while(x_amount>0){
                    seq
                        rcv <p_vcard,x_clt> o_getcash <x_id,x_wdr>;
                        if (x_amount>=x_wdr)
                        {seq
                            x_amount := x_amount - x_wdr;
                            x_resp := "Withdrawal of ". x_wdr
                                      ." Euros accepted"
                         qes}
                        else { x_resp := "Withdrawal of ". x_wdr
                                         ." Euros not accepted" };
                        inv <x_clt> o_getcash <x_id,x_resp>
                    qes
                }
              qes
            ];;

virtualcard ::= {s_vcard}{x_id};;

<?blm
ADDRESSES {
      myns => "http://example";
 myaddress => "http://localhost:8080/active-bpel/services";
}

VARIABLES {
 <x_id,x_amount> => gen:creationReq,<id,amount>,<xsd:int,xsd:int>;
   <x_id,x_wdr>  => gen:withdrawalReq,<id,wdrAmount>,<xsd:int,xsd:int>;
   <x_id,x_resp> => gen:withdrawalResp,<id,msg>,<xsd:int,xsd:string>;
}
?>
```

Pos: 1,0

Figure 4: B*lite*C development environment

Table 1: Mapping of the receive activity

B*lite*	WS-BPEL

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

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

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

Communication activities, invokes and receives, are translated in a different way depending on their arguments and their position in the code. Therefore, the translation of B*lite* programs proceeds in a top-down fashion and, in doing so, the WS-BPEL generator exploits the information previously collected by the Mapper and the B*lite* parser. 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>`. In addition to the excerpts shown in the table, the translation also settles the following activity attributes. 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 a further argument of the `<receive>` activity. The correlation attributes `initiate` and `pattern` are set according to the type of the interaction.

The invoke activity is translated similarly, as shown in Ta-

[5] jEdit is a programmer's text editor written in Java released as free software with full source code, provided under the terms of the GPL 2.0.

Table 2: Mapping of the invoke activity

B*lite*	WS-BPEL

inv pl op <x1,...,xn>	<invoke partnerLink="pl"
	operation="op"
	variable="x" />

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

rcv <p,p'> op <y1,...,yn> ... inv <p'> op <x1,...,xm>	<receive ... /> ... <reply partnerlink="pl" operation="op" variable="x" />

Table 3: Mapping of assign, empty, throw and exit

B*lite*	WS-BPEL
x := e	<assign> <copy> <from> e </from> <to> $var_x.part_x </to> </copy> </assign>
set(x,"path") := e	<assign> <copy> <from> e </from> <to variable="var_x" part="part_x"> <query> path </query> </to> </copy> </assign>
y := get(x,"path")	<assign> <copy> <from variable="var_x" part="part_x"> <query> path </query> </from> <to variable="var_y" part="part_y" /> </copy> </assign>
empty	<empty />
throw	<throw />
exit	<exit />

ble 2; 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, as shown in Table 3, is straightforward. In particular, an assign activity involving message variables is translated by possibly using XPATH queries and by exploiting the type of the involved variables (defined in the declarative part) to identify the corresponding parts. Also the translation of the structured activities does not require a significant effort, as shown in Table 4. Finally, as shown in Table 5, a B*lite* service is rendered as a scope, where the compensation handler is removed and the tag <scope> is replaced by <process>.

5. BLITEC AT WORK

In this section, we present an application of B*lite*C to some illustrative practical scenarios. The WS-BPEL and WSDL files of the presented services are reported in [16], while all B*lite* programs and the corresponding WS-BPEL packages can be retrieved along with the B*lite*C distribution archive.

5.1 A virtual credit card service

A virtual credit card is a prepaid non-physical credit card devised for safe online shopping. A B*lite* specification for creating and handling a virtual credit card is the following:

```
s_vcard ::=
  [ seq
     rcv <p_createcard> o_newcard <x_id,x_amount>;
     while(x_amount>0){
      seq
       rcv <p_vcard,x_clt> o_getcash <x_id,x_wdr>;
       if (x_amount>=x_wdr)
          { seq
             x_amount := x_amount - x_wdr;
             x_resp := "Withdrawal of ". x_wdr .
                       " Euros accepted"
            qes }
       else { x_resp := "Withdrawal of ". x_wdr .
                        " Euros not accepted"};
       inv <x_clt> o_getcash <x_id,x_resp>
      qes }
    qes ];;

virtualcard ::= {s_vcard}{x_id};;
```

Table 4: Mapping of structured activities

B*lite*	WS-BPEL
if (e) {a_1} else {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>

Table 5: Mapping of service definitions

B*lite*	WS-BPEL
[a @ a_f]	<process> <faultHandlers> <catchAll> <sequence> <compensate/> a_f </sequence> </catchAll> </faultHandlers> a </process>

A new card is created by invoking the operation o_newcard and specifying a card identifier and the initial amount. The created instance allows the card holder to perform withdrawals by repeatedly invoking the request-response operation o_getcash until the card is empty. For each withdrawal request, the money availability is checked and a message, stored in x_resp, is sent back. The fact that the invoke activity used for the reply is performed along the same operation of the second receive indicates that the two activities form a synchronous request-response interaction, hence the invoke will be translated into a <reply> activity. The card 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://virtualcard";
   myaddress => "http://XXX:8080/active-bpel/services";
 }
 VARIABLES {
   <x_id,x_amount> => gen:creationReq,<id,amount>,
                      <xsd:int,xsd:int>;
      <x_id,x_wdr> => gen:withdrawalReq,<id,wdrAmount>,
                      <xsd:int,xsd:int>;
     <x_id,x_resp> => gen:withdrawalResp,<id,msg>,
                      <xsd:int,xsd:string>;
 }
?>
```

To compile this B*lite* program, we have to save the above code into a file (named, e.g., `vcard_service.bl`) and execute the command `java -jar blite.jar vcard_service.bl`. This way, the file `virtualcardProcess.bpr`, which is a WS-BPEL package directly deployable into ActiveBPEL, is generated. Of course, the same actions can be also performed by using the editor or the development environment mentioned in Section 4.1. The WS-BPEL file included in the generated package, where irrelevant details have been omitted, is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<process name="virtualcardProcess" ... >
  <import location="virtualcard.wsdl" ... />
  <partnerLinks>
    <partnerLink name="cltPL" partnerLinkType="mwl:cltPLT"
                 myRole="p_vcard" />
    <partnerLink name="p_createcardPL"
                 partnerLinkType="mwl:p_createcardPLT"
                 myRole="p_createcard" />
  </partnerLinks>
  <variables>
    <variable name="var1" messageType="mwl:withdrawalReq" />
    <variable name="var2" messageType="mwl:withdrawalResp" />
    <variable name="var0" messageType="mwl:creationReq" />
  </variables>
  <correlationSets>
    <correlationSet name="x_idCorr"
                     properties="mwl:x_idProp" />
  </correlationSets>
  <faultHandlers>
    <catchAll>
      <sequence> <compensate /> <empty /> </sequence>
    </catchAll>
  </faultHandlers>
  <sequence>
    <sequence>
      <receive partnerLink="p_createcardPL"
               operation="o_newcard"
               variable="var0"
               createInstance="yes">
        <correlations>
          <correlation set="x_idCorr" initiate="yes" />
        </correlations>
      </receive>
      <assign>
        <copy>
          <from variable="var0" part="id" />
          <to variable="var1" part="id" />
        </copy>
        <copy>
          <from variable="var0" part="id" />
          <to variable="var2" part="id" />
        </copy>
      </assign>
    </sequence>
    <while>
      <condition>$var0.amount &gt; 0</condition>
      <sequence>
        <receive partnerLink="cltPL"
                 operation="o_getcash"
                 variable="var1">
          <correlations>
            <correlation set="x_idCorr" initiate="no" />
          </correlations>
        </receive>
        <if>
          <condition>
            $var0.amount &gt;= $var1.wdrAmount
          </condition>
          <sequence>
            <assign>
              <copy>
                <from>$var0.amount - $var1.wdrAmount</from>
                <to variable="var0" part="amount" />
              </copy>
            </assign>
            <assign>
              <copy>
                <from>
                  concat('Withdrawal of ',string($var1.wdrAmount),
                           ' Euros accepted')
                </from>
                <to variable="var2" part="msg" />
              </copy>
            </assign>
          </sequence>
          <else>
            <assign>
              <copy>
                <from>
                  concat('Withdrawal of ',string($var1.wdrAmount),
                         ' Euros not accepted')
                </from>
                <to variable="var2" part="msg" />
              </copy>
            </assign>
          </else>
        </if>
        <reply operation="o_getcash"
               partnerLink="cltPL"
               variable="var2">
          <correlations>
            <correlation set="x_idCorr" initiate="no" />
          </correlations>
        </reply>
      </sequence>
    </while>
  </sequence>
</process>
```

As expected, the resulting WS-BPEL code is more verbose and intricate than the B*lite* one. However, the performed translation turns out to be quite 'clean', in the sense that each B*lite*C activity has been translated into the corresponding WS-BPEL one without introducing 'junk' code.

To deploy the file `virtualcardProcess.bpr`, 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 (Figure 5) among which `virtualcardProcess` should appear. Now, by selecting `Deployed services`, we can retrieve the URLs of the two WSDL files[6] corresponding to the partner links for interacting with the service:

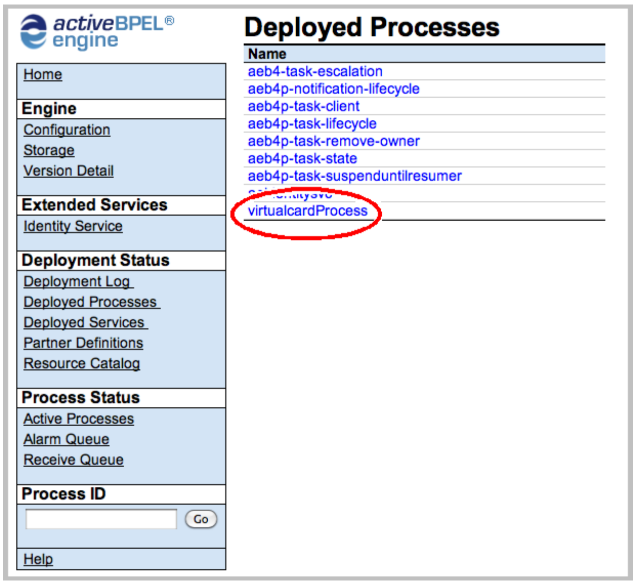
```
http://XXX:8080/active-bpel/services/p_createcardService?wsdl

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

To test the service behaviour, we can use a tool for automatic generation of web service requests, as e.g. soapUI [13], and invoke the service by sending the following SOAP messages:

```
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:vir="http://virtualcard/virtualcard.wsdl">
   <soapenv:Header/>
   <soapenv:Body>
      <vir:x_idEL> 1234 </vir:x_idEL>
      <vir:x_amountEL> 100 </vir:x_amountEL>
   </soapenv:Body>
</soapenv:Envelope>
```

[6] In fact, when provided with a WSDL file, ActiveBPEL produces as many WSDL files as the different partner links.



Deployed Processes

Name
aeb4-task-escalation
aeb4p-notification-lifecycle
aeb4p-task-client
aeb4p-task-lifecycle
aeb4p-task-remove-owner
aeb4p-task-state
aeb4p-task-suspenduntilresumer
aeidentitysvc
virtualcardProcess

Figure 5: List of deployed processes

```
<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:vir="http://virtualcard/virtualcard.wsdl">
   <soapenv:Header/>
   <soapenv:Body>
      <vir:x_idEL> 1234 </vir:x_idEL>
      <vir:x_wdrEL> 50 </vir:x_wdrEL>
   </soapenv:Body>
</soapenv:Envelope>
```

The first message creates a virtual credit card identified by `1234` with `100` Euros as initial amount, while the second message is a request for withdrawing `50` Euros. In response to the second message we get the string `Withdrawal of 50 Euros accepted` and, by selecting `Active Processes` from the console menu, we can verify that the card instance is still running. If we resend the request we obtain the same response, but the instance status changes to `Completed`.

5.2 On asynchronous communication

Frequently, it is assumed that a service request can be processed in a reasonable amount of time, justifying the requirement that the invoker waits for a response related to a synchronous request-response operation. In a business process setting, where communication costs are high or network latency is unpredictable, such assumption usually does not hold and the interactions are better modeled by asynchronous message exchanges. Therefore, an asynchronous messaging approach is considered good practice for web services and service orchestrations in particular.

Although WS-BPEL provides the means for implementing asynchronous communication, it requires programmers to directly deal with endpoint references, as shown by the example in Section 2. Instead, in B*lite* this relevant communication pattern can be easily and transparently implemented.

A B*lite* specification for receiving a request and asynchronously replying with the string `Hello` is the following:

```
s_asyncServer ::= [ seq
                        rcv <server,client> request <x_id>;
                        x := "Hello";
                        inv <client> response <x_id,x>
                      qes ];;
asyncServer ::= {s_asyncServer}{x_id};;
```

Since this service does not need to invoke other services, the declarative part is just as follows:

```
<?blm
 ADDRESSES {
   myns => "http://asyncComm";
   myaddress => "http://XXX:8080/active-bpel/services";
 }
 VARIABLES {
     <x_id> => gen:req,<id>,<xsd:int>;
   <x_id,x> => gen:resp,<id,msg>,<xsd:int,xsd:string>;
 }
?>
```

The corresponding invoker service is rendered in B*lite* as

```
s_asyncClient ::= [ seq
                        rcv <p_init,y_clt> init <y_id>;
                        inv <server,client> request <y_id>;
                        rcv <client> response <y_id,y_msg>;
                        y_resp := "Response: ".y_msg;
                        inv <y_clt> init <y_id,y_resp>
                      qes ];;
asyncClient ::= {s_asyncClient}{y_id};;
```

The process is initialized by invoking the synchronous request-response operation `init`. Then, the created instance asynchronously invokes the partner `server` and, once receives the reply message, appends it at the response and sends the obtained message to the initiator. Notably, in B*lite* synchronous and asynchronous interactions are rendered in a similar way (i.e. as pairs of activities `inv-rcv` and `rcv-inv`); they are distinguished only by the fact that synchronous interactions use the same operation for invoking and receiving.

The declarative part of the asynchronous invoker is:

```
<?blm
 ADDRESSES {
   myns => "http://asyncComm";
   myaddress => "http://XXX:8080/active-bpel/services";
 }
 IMPORTS { asS => "http://asyncComm/asyncServer.wsdl"; }
 VARIABLES {
          <y_id> => asS:req;
    <y_id,y_msg> => asS:resp;
   <y_id,y_resp> => gen:response,<id,resp>,<xsd:int,xsd:string>;
 }
 PARTNERLINKS {
   PARTNERLINK {
       TYPE => asS:serverPLT;
       MY_ROLE client => asS:clientPT;
       PARTNER_ROLE server => asS:serverPT;
   }
 }
?>
```

The types of the messages corresponding to the asynchronous invocation and the related response are imported by the WSDL document of the asynchronous server (identified by the prefix `asS`). Moreover, while the partner link to interact with the initiator partner `client` is automatically generated by B*lite*C, the partner link to interact with `server` must be explicitly declared (the types of the partner link and the involved ports are again imported from the server's WSDL document). Hence, the server must be complied first.

We report here an excerpt of the WS-BPEL program corresponding to `asyncServer`

```
<process name="asyncServerProcess" ... >
  ...
  <partnerLinks>
    <partnerLink name="serverPL" partnerLinkType="mwl:serverPLT"
                  myRole="server"
                  partnerRole="client"
                  initializePartnerRole="no" />
  </partnerLinks>
  <variables> ... </variables>
  <correlationSets>
    <correlationSet name="x_idCorr" properties="mwl:x_idProp" />
  </correlationSets>
  <faultHandlers> ... </faultHandlers>
  <sequence>
    <sequence>
      <receive partnerLink="serverPL"
               operation="request"
               variable="var0"
               createInstance="yes">
        <correlations>
          <correlation set="x_idCorr" initiate="yes" />
        </correlations>
      </receive>
      <assign> ... </assign>
    </sequence>
    <sequence>
      <receive partnerLink="serverPL"
               operation="setClientAddress"
               variable="var2">
        <correlations>
          <correlation set="x_idCorr" initiate="no" />
        </correlations>
      </receive>
      <assign>
        <copy>
          <from variable="var2" part="address" />
          <to partnerLink="serverPL" />
        </copy>
      </assign>
    </sequence>
    <assign> ... </assign>
    <invoke partnerLink="serverPL" operation="response" ... />
  </sequence>
</process>
```

To enable asynchronous communication, the translation automatically puts an additional receive activity (highlighted by a gray background) in the generated WS-BPEL code. This activity will be used by clients to communicate their addresses, which are then assigned to the partner link used for the callback operation. Similarly, in a transparent way, B*lite*C equips the clients invoking this partner link with the symmetric invoke activity.

5.3 An auction service scenario

We show here an application of B*lite*C to a scenario built upon the auction service drawn from the WS-BPEL specification document and already introduced in Section 2.

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

```
s_auction ::=
  [ seq
      flw
        rcv <auctionS,seller> submit
             <x_auctionId,x_creditCardNumberS,x_shippingCost>
        |
        rcv <auctionB,buyer> submit
             <x_auctionId,x_creditCardNumberB,x_phoneNumber>
      wlf;
      x_amount := 1;
      inv <register,auction> process <x_auctionId,x_amount>;
      rcv <auction> regAnswer <x_auctionId,x_registrationId>;
      x_resp := "Thank you!";
      flw
        inv <seller> answer <x_auctionId,x_resp>
        |
        inv <buyer> answer <x_auctionId,x_resp>
      wlf
    qes ];;

auction ::= {s_auction}{x_auctionId};;

<?blm
 ADDRESSES {
       myns => "http://auction";
   myaddress => "http://localhost:8080/active-bpel/services";
 }
 IMPORTS { reg => "http://register/register.wsdl"; }
 VARIABLES {
  <x_auctionId,x_creditCardNumberS,x_shippingCost>
     => gen:sellerReq,<id,ccNum,cost>,
        <xsd:int,xsd:string,xsd:int>;
  <x_auctionId,x_creditCardNumberB,x_phoneNumber>
     => gen:buyerReq,<id,ccNum,phone>,
        <xsd:int,xsd:string,xsd:string>;
  <x_auctionId,x_amount> => reg:regReq;
  <x_auctionId,x_registrationId> => reg:regResp;
  <x_auctionId,x_resp>
     => gen:resp,<id,msg>,<xsd:int,xsd:string>;
 }
 PARTNERLINKS {
   PARTNERLINK {
       TYPE => reg:registerPLT;
       MY_ROLE auction => reg:auctionPT;
       PARTNER_ROLE register => reg:registerPT;
   }
 }
?>
```

Here, differently from the program in Section2, the endpoint references of the registration, buyer and seller services are not explicitly handled by the programmer. Moreover, the only partner link declared is that used for interacting with the registration service.

The *registration service* is rendered in B*lite* as follows:

```
register ::=
  {[ seq
       rcv <register,auction> process <x_id,x_amount>;
       x_registrationId := "someId";
       inv <auction> regAnswer <x_id,x_registrationId>
     qes ]}{x_id};;

<?blm
 ADDRESSES {
       myns => "http://register";
   myaddress => "http://localhost:8080/active-bpel/services";
 }
 VARIABLES {
   <x_id,x_amount>
     => gen:regReq,<id,amount>,<xsd:int,xsd:int>;
   <x_id,x_registrationId>
     => gen:regResp,<id,regId>,<xsd:int,xsd:string>;
 }
?>
```

Its behaviour is very simple: the process gets instantiated by the auction service by invoking the operation `process`; then, the created instance replies with a registration identifier. For the sake of simplicity, we do not model here the generation of a unique identifier (which most likely could be provided by another service).

Finally, we report below the *seller service* (the *buyer service* is defined similarly):

```
v_seller ::=
  [ seq
      rcv <initSeller,initiator> init <x_id,x_cc,x_shipCost>;
      inv <auction,seller> submit <x_id,x_cc,x_shipCost>;
      rcv <seller> answer <x_id, x_resp>;
      inv <initiator> init <x_id,x_resp>
    qes ];;

seller ::= {v_seller}{x_id};;

<?blm ADDRESSES {
   myns => "http://seller";
   myaddress =>"http://localhost:8080/active-bpel/services";
}
IMPORTS { as => "http://auction/auction.wsdl"; }
VARIABLES {
   <x_id,x_cc,x_shipCost> => as:sellerReq;
   <x_id, x_resp> => as:resp;
}
PARTNERLINKS {
   PARTNERLINK {
       TYPE => as:auctionSPLT;
       MY_ROLE seller => as:sellerPT;
       PARTNER_ROLE auction => as:auctionSPT;
   }
}
?>
```

The above specification is very similar to that of the asynchronous client described in Section 5.2.

Once the above programs have been compiled and deployed, regardless of the activation order of the buyer and seller, both services will receive the message `Thank you!` indicating the successful termination of the orchestration.

5.4 An auction service scenario with fault and compensation handling

We now extend the scenario introduced in Section 5.3 with fault and compensation handling. Basically, we allow the buyer and the seller services to cancel the auction, which causes its unregistration. Thus, the auction service becomes:

```
invReg ::=
  seq
    inv <register,auction> process <x_auctionId,x_amount>;
    rcv <auction> regAnswer <x_auctionId,x_registrationId>;
    flw
      inv <seller> answer <x_auctionId,x_resp>
      |
      inv <buyer> answer <x_auctionId,x_resp>
    wlf
  qes;;

compReg ::=
  seq
    x_motivation := "Auction cancelled";
    inv <registerComp> unregister <x_auctionId,x_motivation>
  qes;;

fh ::=
  seq
    x_resp := "The registration has been cancelled";
    flw
      inv <seller> answer <x_auctionId,x_resp>
      |
      inv <buyer> answer <x_auctionId,x_resp>
    wlf
  qes;;

s_auction ::=
  [ seq
      flw
        rcv <auctionS,seller> submit
             <x_auctionId,x_creditCardNumberS,x_shippingCost>
        |
        rcv <auctionB,buyer> submit
             <x_auctionId,x_creditCardNumberB,x_phoneNumber>
      wlf;
      x_amount := 1;
      x_resp := "Thank you!";
      [invReg @ empty * compReg];
      rcv <auctionCanc> cancel <x_auctionId>;
      throw
    qes
    @ fh];;

auction ::= {s_auction}{x_auctionId};;

<?blm
 ADDRESSES {
       myns => "http://auction";
  myaddress => "http://localhost:8080/active-bpel/services";
}
IMPORTS { reg => "http://register/register.wsdl"; }
VARIABLES {
 <x_auctionId,x_creditCardNumberS,x_shippingCost>
     => gen:sellerReq,<id,ccNum,cost>,
         <xsd:int,xsd:string,xsd:int>;
 <x_auctionId,x_creditCardNumberB,x_phoneNumber>
     => gen:buyerReq,<id,ccNum,phone>,
         <xsd:int,xsd:string,xsd:string>;
 <x_auctionId,x_amount> => reg:regReq;
 <x_auctionId,x_registrationId> => reg:regResp;
 <x_auctionId,x_motivation> => reg:unreg;
 <x_auctionId,x_resp>
     => gen:resp,<id,msg>,<xsd:int,xsd:string>;
 <x_auctionId> => gen:cancellation,<id>,<xsd:int>;
}
PARTNERLINKS {
   PARTNERLINK {
       TYPE => reg:registerPLT;
       MY_ROLE auction => reg:auctionPT;
       PARTNER_ROLE register => reg:registerPT;
   }
   PARTNERLINK {
       TYPE => reg:registerCompPLT;
       PARTNER_ROLE registerComp => reg:registerCompPT;
   }
 }
?>
```

Once the auction service receives the two requests, it executes the scope activity `[invReg @ empty * compReg]`: the primary activity `invReg` interacts with the registration service and then replies to buyer and seller, while the activity `compReg` is the corresponding compensation handler that simply invokes the registration service to cancel the registration. After the scope completes, the auction service waits for a cancellation message by either the buyer or the seller. The reception of such a message generates a fault (through the activity `throw`). The fault is handled by the fault handler `fh` that automatically activates the compensation handler `compReg` (we refer to [25] for a complete account of the default compensation mechanism) and sends two messages to buyer and seller to notify that the registration has been cancelled.

5.5 Orchestrating non-Blite services

The previous examples show how B*lite*C can be used to generate complete orchestration scenarios where each service is obtained by a B*lite* specification. However, our tool can be also used to orchestrate B*lite* services together with non-B*lite* ones. To this aim, our specification language provides two simple constructs, `get` and `set`, that permit manipulating XML messages exchanged with (possibly) non-B*lite* services.

Consider a service that receives a US zip code and a preference for the temperature scale, i.e. either the character `c` for Celsius or `f` for Fahrenheit, contacts a first service for getting the current weather conditions and the temperature in degrees Fahrenheit of the city corresponding to the zip code, possibly contacts a second service for converting degrees Fahrenheit into degrees Celsius, and finally sends the obtained weather and temperature information to the invoker. To implement this service we have orchestrated two web services freely provided by CDYNE [1]. The corresponding B*lite* specification is as follows:

```
zipWeatherClient ::=
 {[ seq
     rcv <initWeatherClient,initiator> init <num,scale,reqWeath>;
     inv <weather, cb_weather> reqWeather <reqWeath>;
     rcv <cb_weather> reqWeather <respWeath>;
     x := get(respWeath, "/wth:GetCityWeatherByZIPResponse
                          /wth:GetCityWeatherByZIPResult
                          /wth:Description");
     x_city := get(respWeath, "/wth:GetCityWeatherByZIPResponse
                               /wth:GetCityWeatherByZIPResult
                               /wth:City");
     x_temF := get(respWeath, "/wth:GetCityWeatherByZIPResponse
                               /wth:GetCityWeatherByZIPResult
                               /wth:Temperature");
     x := "The current weather at " . x_city . " is: " . x .
       ", the temperature is: ";
     if (scale == "c")
        {seq
          reqConv := $litConv;
          set(reqConv,"/tem:FahrenheitToCelcius
                   /tem:nFahrenheit") := x_temF;
          inv <converter, cb_converter> reqConversion <reqConv>;
          rcv <cb_converter> reqConversion <respConv>;
          x_temC := get(respConv,"/tem:FahrenheitToCelciusResponse
                                  /tem:FahrenheitToCelciusResult");
          x := x . x_temC ." °C"
         qes}
     else {x := x . x_temF ." °F"};
     inv <initiator> init <num,x>
    qes ]}{num};;

<?blm
  ADDRESSES {
         myns => "http://zipWeather";
    myaddress => "http://localhost:8080/active-bpel/services";
  }
  IMPORTS {
    wth => "http://ws.cdyne.com/WeatherWS/";
    tem => "http://webservices.daehosting.com/temperature";
  }
  VARIABLES {
    <reqWeath> => wth:GetCityWeatherByZIPSoapIn;
    <respWeath> => wth:GetCityWeatherByZIPSoapOut;
    <num,scale,reqWeath>
```

```
       => gen:req, <init,scale,request>,
          <xsd:integer,xsd:string,(El)wth:GetCityWeatherByZIP>;
    <num,x>
       => gen:resp, <init,response>,<xsd:integer,xsd:string>;
    x_city => xsd:string;
    x_temF => xsd:decimal;
    <reqConv> => tem:FahrenheitToCelciusSoapRequest;
    <respConv> => tem:FahrenheitToCelciusSoapResponse;
    x_temC => xsd:string;
  }
  LITERALS {
   litConv => [[<tem:FahrenheitToCelcius
               xmlns:tem="http://webservices.daehosting.com
                          /temperature">
                 <tem:nFahrenheit>0</tem:nFahrenheit>
               </tem:FahrenheitToCelcius>]];
  }
  PARTNERLINKS {
      PARTNERLINK {
         TYPE => gen:weatherPLT;
         PARTNER_ROLE weather => wth:WeatherSoap,
                     (reqWeather =>GetCityWeatherByZIP);
      }
      PARTNERLINK {
         TYPE => gen:converterPLT;
         PARTNER_ROLE
             converter => tem:TemperatureConversionsSoapType,
             (reqConversion =>FahrenheitToCelcius);
      }
  }
?>
```

The `get` construct is used here to extract information from the XML messages received from the two external web services, while the `set` construct is used to insert the received temperature expressed in degrees Fahrenheit into the XML request message for the converter service, whose structure has been previously initialized by means of the literal `litConv`.

Once the program has been compiled and deployed, if we invoke the operation `init` by specifying the zip code `90210` and the scale Celsius, we will get back a string of the form:

```
The current weather at Beverly Hills is: Clear,
the temperature is: 14.4444444 °C
```

6. CONCLUDING REMARKS

We have presented B*lite*C, a software tool for supporting a rapid and easy development of WS-BPEL applications. The tool aims at solving some well-known programming problems of WS-BPEL caused by its XML syntax, lack of a formal semantics, and non-standardization of the deployment procedure. Basically, B*lite*C takes as inputs programs written in B*lite*, a prototypical orchestration language inspired to WS-BPEL but with a simpler syntax and a well-defined operational semantics, and provides as output the corresponding deployable WS-BPEL programs.

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 [5], Intalio|Designer [10], ActiveVOS Designer [7], and Eclipse BPEL designer [9]. Although their use is quite intuitive, developing large applications by using them can be awkward and annoying compared to the more classic textual approach. Indeed, graphical notations turn out to be suitable for beginner WS-BPEL programmers to represent simple business process workflows, but do not allow more expert programmers to exploit commonly used functionalities, such as e.g. copy/cut/paste, and are inappropriate for expressing some (textual) information, such as e.g. correlation sets. Moreover, graphical designers have a significant negative impact on performance during the programming phase (that is, indeed, the phase of the software development process on which we focus on), since they usually are plugins of heavy software development environments such as JDeveloper [3] and Eclipse [4]. Some other works with a similar aim are [28, 32, 14]. The first two present some tools that produce WS-BPEL processes starting, respectively, from UML- and Petri Nets-based representations of SOC applications. Due to the use of graphical representations, also these tools suffer from the problems previously mentioned. Instead, the third one proposes a mapping from a π-calculus based formalism into WS-BPEL. In all three approaches, only non-executable WS-BPEL processes are generated, i.e. the generated code should be thought of as a template code where, besides binding and deployment details, programmers have also to define things such as partner links, variables, port types, correlations sets, etc. by editing the generated files. Another related work is [29], which 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.

Currently, the WS-BPEL packages generated by B*lite*C are intended to be deployed on ActiveBPEL. This is just to demonstrate feasibility of our approach. In fact, 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 other freely available engines, such as Oracle BPEL Process Manager, Apache ODE [8] and Beepell [23]. 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. 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. It is also worth noticing that the semantics of B*lite*, which is in fact quite close to that of ActiveBPEL, could be rather tough to render by some WS-BPEL engines, whose semantics may significantly differ on low-level implementation details (e.g. message queue handling) or may more strictly (or inappropriately) enforce some WS-BPEL constraints. For instance, it may happen that a process instance should receive a message from a partner according to the B*lite* semantics, while instead the message cannot be effectively accepted according to the semantics of the considered engine, due to e.g. some peculiar correlation constraint. In such a case, B*lite*C should identify the potential conflicting receives in the generated WS-BPEL program and, e.g., replace them by a single receive enabling some proper coordination activities.

We also plan to enrich the B*lite*C development environment presented in Section 4.1 with further functionalities, such as deployment/undeployment facilities over remote servers, and debugging tools, such as automatic generation of web interfaces for invoking the created services and log recording based on an embedded dedicated web service. This latter tool could require to extend the syntax of B*lite* accepted by B*lite*C with a construct for printing strings into the log that would be translated into a WS-BPEL one-way interaction.

For what concern the language B*lite*, we also intend to investigate its extension to cover some WS-BPEL constructs that at the time being have been left out, such as timed activities, event and termination handlers, and more sophisticated forms of fault and compensation handling involving named faults and compensation of specified scopes. We do not envisage any major issue in translating such constructs in WS-BPEL code, while their addition to B*lite* would require to significantly revise the formal definition of the operational semantics of the language.

Finally, we intend to develop formal analysis techniques, e.g. based on model checking (as in [14, 15]), for B*lite* specifications. This way, we would be able to specify in B*lite* an orchestration scenario, validate its behaviour by using formal tools, and deploy it as a set of WS-BPEL programs.

7. ACKNOWLEDGMENTS

We thank Alessandro Lapadula for his fundamental contribution to the definition of the language B*lite*. We also thank the anonymous reviewers of SAC'10 and Applied Computing Review for their helpful comments, and people attending the SOAP track at SAC'10 for the fruitful discussions following the presentation of our work. Finally, we thank the chairs of the SOAP track for their invitation to submit to the Applied Computing Review and their support.
This work has been sponsored by the EU project Sensoria, IST-2005-016004.

8. REFERENCES

[1] CDYNE Corporation. `http://www.cdyne.com/`.

[2] jEdit Programmer's Text Editor 4.3. `http://www.jedit.org/`.

[3] Oracle JDeveloper. `http://www.oracle.com/technology/products/jdev`.

[4] The Eclipse project. `http://www.eclipse.org`.

[5] Oracle BPEL Process Manager 10.1.3, December 2007. `http://www.oracle.com/technology/bpel`.

[6] ActiveBPEL 5.0.2, October 2009. `http://sourceforge.net/projects/activebpel502/`.

[7] ActiveVOS Designer 5.0.2, June 2009. `http://www.activevos.com/`.

[8] Apache ODE 1.3.3, August 2009. `http://ode.apache.org`.

[9] Eclipse BPEL project 0.4.0, May 2009. `http://www.eclipse.org/bpel`.

[10] Intalio|Designer Community Edition 6.0.1, August 2009. `http://www.intalio.com/products/bpm/community-edition/designer`.

[11] JavaCC 4.2, April 2009. `https://javacc.dev.java.net`.

[12] JDOM 1.1, April 2009. `http://www.jdom.org`.

[13] soapUI 3.5, March 2010. `http://www.soapui.org`.

[14] F. Abouzaid and J. Mullins. A Calculus for Generation, Verification and Refinement of BPEL Specifications. In *WWV*, volume 200 of *ENTCS*, pages 43–65. Elsevier, 2008.

[15] F. Abouzaid and J. Mullins. Model-checking Web Services Orchestrations using BP-calculus. In *FOCLASA*, volume 255 of *ENTCS*, pages 3–21. Elsevier, 2009.

[16] L. Cesari, R. Pugliese, and F. Tiezzi. A tool for rapid development of WS-BPEL applications. Technical report, Dip. Sistemi e Informatica, Univ.Firenze, 2010.

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

[18] N. Dragoni and M. Mazzara. A Formal Semantics for the WS-BPEL Recovery Framework: The pi-Calculus Way. In *WS-FM*, volume 6194 of *LNCS*, pages 92–109. Springer, 2010.

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

[20] D. Fahland and W. Reisig. ASM-based Semantics for BPEL: The Negative Control Flow. In *Abstract State Machines*, pages 131–152, 2005.

[21] A. Ferrara. Web services: a process algebra approach. In *ICSOC*, pages 242–251. ACM, 2004.

[22] M. Gudgin, M. Hadley, and T. Rogers. Web Services Addressing 1.0 - Core. Technical report, W3C, May 2006. W3C Recommendation.

[23] T. Hallwyl, F. Henglein, and T. Hildebrandt. A standard-driven implementation of WS-BPEL 2.0. In *SAC*, pages 2472–2476. ACM, 2010.

[24] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In *Business Process Management*, volume 3649 of *LNCS*, pages 220–235. Springer, 2005.

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

[26] N. Lohmann. A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In *WS-FM*, volume 4937 of *LNCS*, pages 77–91. Springer, 2008.

[27] R. Lucchi and M. Mazzara. A pi-calculus based semantics for WS-BPEL. *J. Log. Algebr. Program.*, 70(1):96–118, 2007.

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

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

[30] OASIS WSBPEL TC. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, April 2007.

[31] C. Ouyang, E. Verbeek, W. van der Aalst, S. Breutel, M. Dumas, and A. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Sci. Comput. Program.*, 67(2-3):162–198, 2007.

[32] W. M. P. van der Aalst and K. B. Lassen. Translating unstructured workflow processes to readable BPEL: Theory and implementation. *Information & Software Technology*, 50(3):131–159, 2008.

[33] M. Weidlich, G. Decker, and M. Weske. Efficient Analysis of BPEL 2.0 Processes Using π-calculus. In *APSCC*, pages 266–274. IEEE, 2007.