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 Blite, a software tool we have developed for supporting a rapid and easy development of WS-BPEL applications. Blite translates service orchestrations written in Blite, 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 Blite, a software tool that accepts as an input a specification written in the lightweight orchestration language Blite [12] and returns the corresponding WS-BPEL program together with the associated WSDL and deployment descriptor files.

Blite 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 Blite only retains the core features of WS-BPEL. It follows that Blite is simpler and more compact than WS-BPEL, although it maintains the same descriptive power. Using Blite for initially specifying a service orchestration offers some significant advantages. From the one hand, the Blite textual notation is certainly more manageable than those, also graphical ones, proposed for WS-BPEL. From the other hand, Blite is equipped with a formal operational semantics that clarifies all ambiguous and intricate aspects of WS-BPEL.

BliteC 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, BliteC 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 engines.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC’10 March 22-26, 2010, Sierre, Switzerland.
Copyright 2010 ACM 978-1-60558-638-0/10/03 ...$10.00.
freely available engines, namely Oracle BPEL Process Manager [1] and Apache ODE [4]. Of course, to preserve the semantics of the original Blite 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 BliteC 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 Blite program accepted by BliteC is composed of a Blite specification and a declarative part. The former focuses 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 Blite accepted by BliteC 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 <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 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 Blite, this behaviour is rendered in terms of a pair of activities invoke–receive executed by the invoked service.

Data can be shared among different activities through shared variables ( ranged over by x, x1, … ). 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, comparison, and string operators.

Blite specifications are finite compositions of definitions (that assign names to Blite 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 Blite operational semantics.

2.2 Declarative part

The declarative part of a Blite program specifies configuration data necessary to properly translate the Blite specification into an executable WS-BPEL program. Notably, BliteC requires the user to insert only the strictly necessary data. The declarations must be included within <blite> and </blite>, and can occur in any position within a Blite program. A declarative part has the following form:

```xml
<blite
  ADDRESSES { syna => "base_for_namespaces";
               syaddress => "base_for_service_url"; }
  IMPORTS { associations prefix => "url1"; }
  VARIABLES { variable and message declarations }
  LITERALS { associations literal_name => [{Literal_code}]; }
  PARTNERLINKS { partner link type declarations }
</blite>
```

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 syna) and the base for the address where the new service will be hosted (after the keyword syaddress).

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.
Blite 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 and external) 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 or receiving target, can be declared in two ways:

- by using an imported message type, e.g. in
  
  ```xml
  <x_count,x_id> => bck:number;  
  ```

- by generating a new message type, e.g. in
  
  ```xml
  shipComplete
  ```

A WSDL document defines messages composed of three integer parts, i.e., message composed of 

```xml
<x_id,x_c,x_items> => gen:shipOrder, 
```

that are tuples of variables used as either sending or receiving target, can be declared in two ways:

- by using an imported message type, e.g. in
  
  ```xml
  <x_count,x_id> => bck:number;  
  ```

- by generating a new message type, e.g. in
  
  ```xml
  shipComplete
  ```

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

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

```java
PARTNER_LINK { TYPE => partner_link_type;  
MY_ROLE partner => port_type;  
PARTNER_ROLE partner => port_type;  
}
```

where the association for my_role can be omitted whenever the process does not play any role. Moreover, to de-couple the Blite 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

BliteC is developed in Java 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 JJTree3 for allowing the parsers to build parse trees (already arranged to support the Visitor design pattern [11]).

BliteC is composed of five main components:

- Mapper parses the declarative part of the input Blite program and initializes a map that associates each declared object (e.g. partner link, literal, variable, . . . ) to its name;
- Blite parser analyzes the Blite specification within the input program, completes the map created by Mapper and creates the parse tree of the Blite 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 Blite 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

1BliteC 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.
2JRE and JDK, version 6.
3JJTree is included within JavaCC.
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 Bitle service is rendered as a scope, where the compensation handler is removed and the tag <scope> is replaced by <cscope>.

### 4. BLITEC AT WORK

We show an application of BitleC 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:

**Table 1: Mapping of the receive activity**

<table>
<thead>
<tr>
<th>Btle</th>
<th>WS-BPEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>pck  ...</td>
<td>&lt;onMessage partnerLink=&quot;pl&quot; operation=&quot;op&quot; variables=&quot;x&quot; /&gt;</td>
</tr>
<tr>
<td>rec pl op &lt;x1,...,xn&gt;...</td>
<td>&lt;invoke partnerLink=&quot;pl&quot; operation=&quot;op&quot; inputVariable=&quot;y&quot; outputVariable=&quot;z&quot; /&gt;</td>
</tr>
<tr>
<td>kcp</td>
<td>&lt;catchAll&gt;</td>
</tr>
</tbody>
</table>

**Table 2: Mapping of structured activities**

<table>
<thead>
<tr>
<th>Btle</th>
<th>WS-BPEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (e) {a1} {a2}</td>
<td>&lt;if&gt; &lt;condition&gt; e &lt;/condition&gt; &lt;a1&gt; &lt;/a2&gt; &lt;/if&gt;</td>
</tr>
<tr>
<td>while (e) {a}</td>
<td>&lt;while&gt; &lt;condition&gt; e &lt;/condition&gt; &lt;a&gt; &lt;/while&gt;</td>
</tr>
<tr>
<td>seq a1;...;an</td>
<td>&lt;sequence&gt; a1 ... an &lt;/sequence&gt;</td>
</tr>
<tr>
<td>flw a1</td>
<td>&lt;flow&gt; a1 ... an &lt;/flow&gt;</td>
</tr>
<tr>
<td>pck a1;...;an kcp</td>
<td>&lt;pick&gt; a1 ... an &lt;/pick&gt;</td>
</tr>
</tbody>
</table>

To compile this Bitle program, we have to save the above code into a file (named, e.g., backend.service.bl) and execute the following command:

```
$ java -jar bitle.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/Service/ (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
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 Bitte is defined as:

```plaintext
a_ship ::= seq
  x_shipped := 0;
  while (x_shipped < x_items) {
    seq
      inv <backend_ch_backend> o_num <x_id>;
      rcv <ch_backend> o_num <count,x_id>;
      if (x_count < 0) {
        throw
      } else {
        inv <x_cust> o_notice <x_id,x_count>;
        x_shipped := x_shipped + x_count
      }
    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 <ship_c, x_count> o_req <x_id,x_count,x_items>;
  if (x_count > 0) {
    inv <x_cust> o_notice <x_id,x_count>;
    x_shipped := x_shipped + x_count
  }
  qes
];

PARTNERLINK { TYPE => bck:clientPLT; MY_ROLE cust => bck:x_custPT; PARTNER_ROLE ship_srv => bck:p_shipPT; } }
```

This client receives a shipping request and forwards it to the shipping service. Then, it waits until 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 Bitte 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

[5] Eclipse BPEL project 0.4.0, May 2009.