Using formal methods to develop WS-BPEL applications[☆]

Alessandro Lapadula[a], Rosario Pugliese[a,*], Francesco Tiezzi[a]

[a]*Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, Viale Morgagni 65, I-50134, Firenze, Italy*

Abstract

In recent years, there has been an ever increasing acceptance of WS-BPEL as a standard language 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 firstly point out major loose points of the WS-BPEL specification by means of many examples, some of which are also exploited to test and compare the behaviour of three of the most known freely available WS-BPEL engines. We show that, as a matter of fact, these engines implement different semantics, which undermines portability of WS-BPEL programs over different platforms. Then we introduce B*lite*, a prototypical orchestration language equipped with a formal operational semantics, which is closely inspired to but simpler than WS-BPEL. Indeed, B*lite* is designed around some of WS-BPEL distinctive features like partner links, process termination, message correlation, long-running business transactions and compensation handlers. Finally, we present B*lite*C, a software tool we have developed for supporting a rapid and easy development of WS-BPEL applica-

[☆]This work is partially based on two preliminary papers appeared in [29, 17] and has been partially supported by the EU project SENSORIA IST-2005-016004.

[*]Corresponding author

Email addresses: `lapadula@dsi.unifi.it` (Alessandro Lapadula), `pugliese@dsi.unifi.it` (Rosario Pugliese), `tiezzi@dsi.unifi.it` (Francesco Tiezzi)

tions. B*lite*C translates service orchestrations written in B*lite* into executable WS-BPEL programs. We illustrate our approach by means of an example borrowed from the official specification of WS-BPEL.

Key words: Service-oriented architectures, Web services, Formal methods, WS-BPEL, Operational semantics, Compilers

Contents

1. Introduction

Information systems are by now at the very foundations of our society, mainly due to recent considerable advances in the field of Information Technology (IT) and on-line availability of enormous amounts of data. This is having a significant impact especially on the business sector, where organizations depend more and more on functional and flexible IT infrastructures, and need to integrate and adapt their existing systems to enable automating complex and distributed business processes as a whole. The challenges posed by information interchange, software integration, and B2B are addressed by *Service-Oriented Computing* (SOC), an emerging paradigm for distributed and e-business computing that finds its origin in object-oriented and component-based software development. SOC aims at enabling developers to build networks of integrated and collaborative applications, regardless of the platform where the applications run and the programming language used to develop them, through the use of *services*, sort of loosely coupled, reusable software components.

The so-called *Web Services* (WSs) are currently one of the most successful and well-developed implementations of the SOC general paradigm. WSs are software components deployed on the World Wide Web, that can be discovered and exploited both by human clients and other services. A key factor for the success of WSs is the fact that their underlying architecture is the Web, that is nowadays a widespread and extensively used platform suitable to connect different companies and customers. Indeed, independently developed applications can be exposed as services and can be interconnected by exploiting the Web infrastructure with related standards, e.g. HTTP, XML, SOAP, WSDL and UDDI. These standards permit to replace proprietary interfaces and data formats with a standard Web-messaging infrastructure based on XML technologies, thus facilitating automated integration of newly built and legacy applications, both within and across enterprise boundaries.

For instance, the W3C standard WSDL (*Web Services Description Language*, [18]) permits to express WSs public interfaces, i.e. the functionalities offered and required by WSs by means of the signatures of operations and the structure of messages for invoking them and returned by them. WSDL descriptions can then be exploited by client applications to determine the location of a remote web service and the operations it implements, as well as how to access and use each operation.

The above standard technologies are usually sufficient for simple applications integration needs. However, creation of complex B2B applications and automated integration of business processes across enterprises requires managing such features as, e.g., asynchronous interactions, concurrency, workflow coordination, business transactions and exceptions. This raises the need for service composition languages, an additional layer on top of the WSs protocol stack. In this setting, there is an ever increasing acceptance of the OASIS standard WS-BPEL (*Web Services Business Process Execution Language*, [36]), a language expressly designed to define business processes, i.e. software entities capable of orchestrating available WSs by invoking them according to given sets of rules to meet business requirements. It is worth emphasizing that service orchestrations may themselves become services, making composition a recursive operation.

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 precise semantics and its specification document [36], written in 'natural' language, contains a fair number of acknowledged loose points that may give rise to different interpretations. Some of these loose points are due to a extensive use in [36] of the keyword "SHOULD", which indicates recommended requirements that can be for some reason ignored, and leave the difficult task of understanding the full implications of the choice to the implementers. For example, the sentence stating that the "WS-BPEL processor SHOULD throw a conflictingReceive fault" when there exist "indistinguishable" conflicting receive activities (see [36, Section 10.4]) certainly cannot help the implementers, which can be led to implement very different semantics. Similarly, it seems sometimes not appropriate the choice of labelling some implementation details as "out of scope" for the WS-BPEL specification. Examples of "out of scope" indications are the description of the deployment of a WS-BPEL process (see [36, Section 1]) and the handling of an incoming request message that no process instance is able to receive (see [36, Section 9.2]). Other misinterpretations might arise from some ambiguous, and sometimes conflicting, sentences (see also Section 2.2). For example, the relationship between WS-BPEL (multiple) *start activities* and the mechanisms handling race conditions is not fully clarified; moreover, subtle behaviours can arise when implementing activities that cause immediate termination of other activities, if suitable measures for 'protecting' such critical activities, as fault and compensation handlers, are not taken into account. Clearly, different implementation choices can produce very different beaviours for a same WS-BPEL process. The fact that WS-BPEL has become an OASIS standard has not solved all the difficulties of using the language.

In this paper, we firstly point out major loose points of the WS-BPEL specification by means of many examples focussing on key points of the language specification, like message correlation, asynchronous message delivering, multiple start and conflicting receive activities, scheduling of parallel activities, forced termination and eager execution of activities causing termination, and handlers protection. The examples are exploited to test and compare three of the most known freely available WS-BPEL engines, namely ActiveBPEL [9], Apache ODE [10] and Oracle BPEL Process Manager [37]. As a matter of fact, the considered engines implement different semantics and, hence, the portability of WS-BPEL programs across different platforms is considerably undermined. Portability is further compromised since the deployment procedure of WS-BPEL programs is not standardised. Thus, to execute a WS-BPEL program, besides the associated WSDL document, different engines require different (and not integrable) *process deployment descriptors*, i.e. sets of configuration files that describe how the program should be deployed in the engine.

To face these difficulties, we put forward using *formal methods* as a means to build up a framework to precisely describe the behaviour of a SOC application, to state and prove its properties, and to direct attention towards issues that might otherwise be overlooked. Therefore, we define B*lite*, a 'lightweight' orchestration language closely inspired to WS-BPEL. While the set of WS-BPEL constructs is not intended to be a minimal one, to keep the language manageable, the design of B*lite* only retains the core features of WS-BPEL. For example, B*lite* sheds light on the relationship between compensation activities and the control flow of the originating process, and illustrates the mechanisms for service instance creation and identification, and their interplay. It follows that B*lite* is simpler and more compact than WS-BPEL, although it maintains the same descriptive power. Using B*lite* for initially specifying a service orchestration offers some significant advantages. From the one hand, the B*lite* textual notation is certainly more manageable than those, also graphical ones, proposed for WS-BPEL. From the other hand, B*lite* is equipped with a formal operational semantics that clarifies all loose and intricate aspects of WS-BPEL.

To better take advantage of B*lite*, we have developed B*lite*C, a software tool that accepts as an input a specification written in B*lite* and returns the corresponding WS-BPEL program together with the associated WSDL and deployment descriptor files. This way,

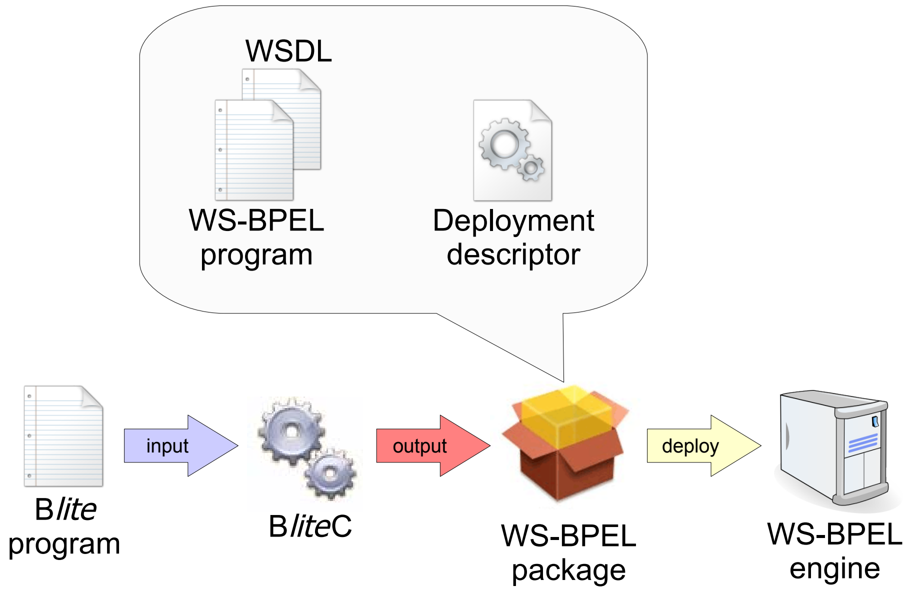


Figure 1: B*lite*C workflow

B*lite*C further simplifies the programmers work by also automatizing the deployment procedure. In fact, the returned files are properly packaged to be immediately executed in the WS-BPEL engine ActiveBPEL that, according to our tests, 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 summarized in Figure 1.

The rest of the paper is organized as follows. Section 2 provides a brief summary of WS-BPEL and, by means of many specific examples, illustrates major loose points of WS-BPEL and the tests carried out with the three previously mentioned WS-BPEL engines. Section 3 presents B*lite*'s syntax and operational semantics. Section 4 introduces B*lite*C. Specifically, it presents the syntax of B*lite* accepted by the tool and illustrates the correspondence between B*lite* constructs and WS-BPEL activities. Section 5 describes an application of B*lite*C to an example borrowed from the official WS-BPEL specification. Finally, Section 6 touches upon more closely related work and directions for future work.

2. Overview of WS-BPEL and experimentation

In this section, we provide an overview of WS-BPEL and present some illustrative examples of WS-BPEL programs used to test and compare the behaviour of three of the most known freely available WS-BPEL engines, namely ActiveBPEL [9], Apache ODE [10] and Oracle BPEL Process Manager [37]. The section ends with an evaluation of the results.

2.1. A glimpse 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 activities that are distinguished between *basic activities* and *structured activities*.

The basic activities provided are: `<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.

Notably, synchronization dependencies among activities, other than by means of control flow constructs, can also be specified through *flow links* to form directed acyclic graphs. A flow link is a conditional transition that connects a ‘source’ activity to a ‘target’ activity. When a source activity completes, the associated *transition condition* is evaluated to determine the status of the *join condition* that acts on the flow link of the target activity. A target activity may only start when all its source activities complete and its join condition evaluates to true.

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 with an example of a shipping service described in the official specification of WS-BPEL [36, Section 15.1]. This example will allow us to illustrate most of the 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. We report below a skeleton description:

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

The corresponding WS-BPEL program follows, 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="shippingService">
...
<correlationSets>
  <correlationSet name="shipOrder" properties="props:shipOrderID" />
</correlationSets>
<sequence>
<receive partnerLink="customer"
         operation="shippingRequest" variable="shipRequest">
  <correlations>
    <correlation set="shipOrder" initiate="yes" />
  </correlations>
</receive>
<if>
  <condition>
    bpel:getVariableProperty('shipRequest','props:shipComplete')
  </condition>
  <sequence>
    <assign>
      <copy>
        <from variable="shipRequest" property="props:shipOrderID" />
        <to variable="shipNotice" property="props:shipOrderID" />
      </copy>
      <copy>
```

[1]The fully detailed version of the WS-BPEL process and the associated WSDL document can be found in [36].

```
      <from variable="shipRequest" property="props:itemsCount" />
      <to variable="shipNotice" property="props:itemsCount" />
    </copy>
  </assign>
  <invoke partnerLink="customer"
          operation="shippingNotice" inputVariable="shipNotice">
    <correlations>
      <correlation set="shipOrder" />
    </correlations>
  </invoke>
</sequence>
<else>
  <sequence>
    <assign>
      <copy>
        <from>0</from>
        <to>$itemsShipped</to>
      </copy>
    </assign>
    <while>
      <condition>
        $itemsShipped
        &lt; bpel:getVariableProperty('shipRequest','props:itemsTotal')
      </condition>
      <sequence>
        <assign>
          <copy>
            <opaqueFrom />
            <to variable="shipNotice" property="props:shipOrderID" />
          </copy>
          <copy>
            <opaqueFrom />
            <to variable="shipNotice" property="props:itemsCount" />
          </copy>
        </assign>
        <invoke partnerLink="customer"
                operation="shippingNotice"> inputVariable="shipNotice">
          <correlations>
            <correlation set="shipOrder" />
          </correlations>
        </invoke>
        <assign>
          <copy>
            <from>
            $itemsShipped
```

```
              + bpel:getVariableProperty('shipNotice','props:itemsCount')
              </from>
              <to>$itemsShipped</to>
            </copy>
          </assign>
        </sequence>
      </while>
    </sequence>
  </else>
</if>
</sequence>
</process>
```

Notice that the above WS-BPEL program is an *abstract* process, i.e. a partially specified process, and cannot be executed, as it uses an 'opaque' activity to model an interaction with a shipping back-end system.

2.2. *An assessment of three* WS-BPEL *engines*

We now present some illustrative examples of WS-BPEL programs and use them to test and compare the behaviour of three of the most known freely available WS-BPEL engines, namely ActiveBPEL [9], Apache ODE [10] and Oracle BPEL Process Manager [37] (the former two are open source projects, whereas the latter is distributed under the Oracle Technology Network Developer License)[2]. For our evaluation, we have taken into account fundamental features of WS-BPEL that remained unchanged since its initial version.

For the sake of readability, in this section WS-BPEL programs are presented by exploiting the graphical notations introduced in Figure 2, rather than the usual verbose textual form. We additionally use the following symbols:

- ⓘ to label an activity that initializes correlated variables;

[2]ActiveBPEL and Oracle BPEL Process Manager are also part of (commercial) tool suites, namely ActiveVOS and Oracle SOA Suite, for designing, developing, testing, deploying and maintaining WS-BPEL applications.

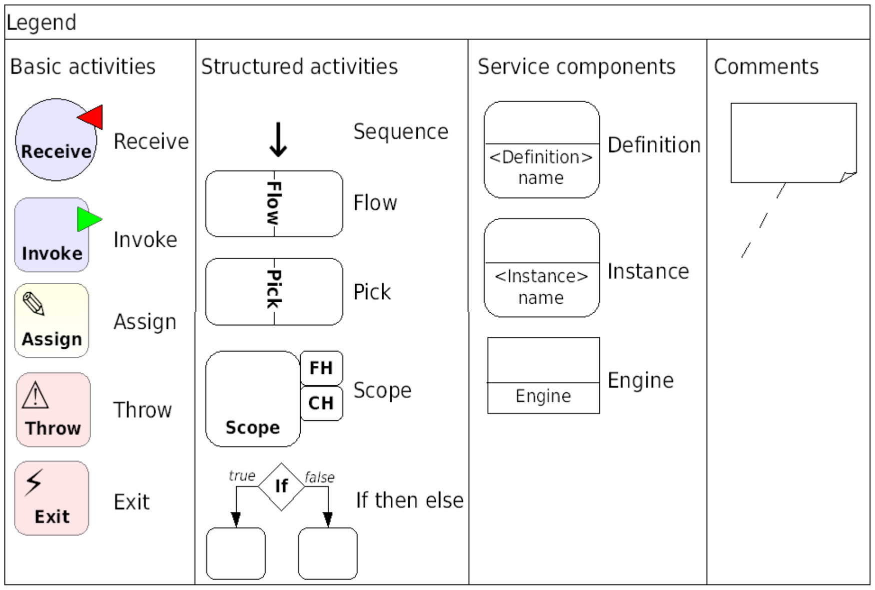


Figure 2: Basic/structured activities and service components used for illustrating the examples

- Ⓤ to label a receive activity that does not use correlated variables;
- © to label an activity that checks correlated variables;
- Ⓘⓒ to label an activity that initializes or checks correlated variables;
- ☎ to label an activity waiting for a message from a partner;
- ✔ to label a completed activity;
- ✽ to label a completed start activity that initiates a new instance of the service;
- ✘ to label a terminated activity due to the execution of `<exit>` or `<throw>` activities.

Example 2.1: Message correlation. A client can request a log-on operation via LogOn, and can request some logging information via RequestLogInfo; this information can be asynchronously obtained by implementing the callback operation SendLogInfo (on the use of asynchronous request-response patterns in service-oriented applications see also

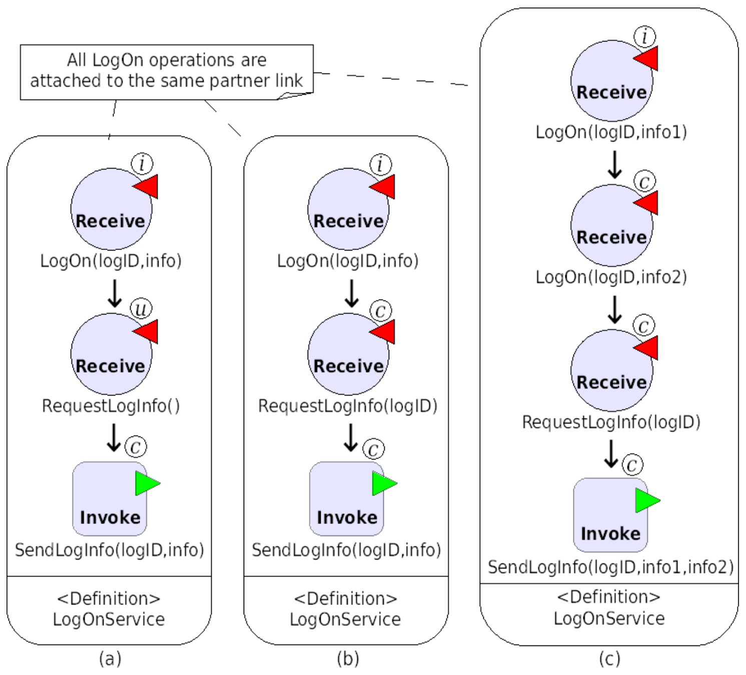


Figure 3: Message correlation

Example 2.2). Correlation variables can be exploited to correlate, by means of their same contents, different service interactions logically forming a same 'session'. For example, consider the simple service LogOnService in Figure 3(a) providing 'log-on' and 'request-log-info' operations. Initially, to request a log-on a client must send its logID with some other data. Then, the service waits for a request from the client to provide some logging information[3]. After that, the service can reply (and terminate) by sending the requested information to the client. Notably, the WS-BPEL process in Figure 3(a) cannot ensure that the service does provide logging information properly. In fact, since the messages for operations LogOn and RequestLogInfo are uncorrelated, if concurrent instances are running

[3]For the sake of simplicity, we assume here that the logging information are simply the data sent by the client through invocation of operation LogOn. In a more realistic scenario, of course, logging information could be internally computed by LogOnService or retrieved from a (possibly external) service.

then, e.g., successive invocations for the same instance can be mixed up and delivered to a wrong instance. This behavior can be prevented by simply correlating consecutive messages by means of some correlation data, e.g. logID, as in the modified service LogOnService of Figure 3(b).

A special case is when the two initial receives are on the same partner and operation, as in Figure 3(c) where LogOnService requires some extra-information from the client, so that it waits for two consecutive log-on requests to let the client logging on the service. This is allowed by the WS-BPEL specification [36, Section 10.4] that, however, does not mention that possible conflicting receives could arise. This situation is illustrated in Figure 4(a), where it is assumed that a client process has performed two log-on requests with data info1 and info2 that, accordingly to the intended semantics of WS-BPEL, should trigger only one instantiation of the service. This is indeed the behaviour of ActiveBPEL and Apache ODE, that exploit the received data to correlate the two consecutive receives and, thus, to prevent creation of a wrong new instance. On the contrary, when executing this example, Oracle BPEL creates two instances, one for each received request as shown in Figure 4(b). An important consequence, and indeed an unexpected side effect, is that the created instances are in conflict and, then, will soon get stuck.

Example 2.2: Asynchronous message delivering. In service-oriented systems communication paradigms are usually asynchronous (mainly for scalability reasons [13]), in the sense that there may be an arbitrary delay between the sending and the receiving of a message, the ordering in which messages are received may differ from that in which they were sent, and a sender cannot determine if and when a sent message will be received. We can guess from [36, Section 10.4], that this is also the case of WS-BPEL. To illustrate, consider the WS-BPEL process in Figure 5(a) representing a client logging on the previous service depicted in Figure 3(b). After the request for some user information is performed by the first

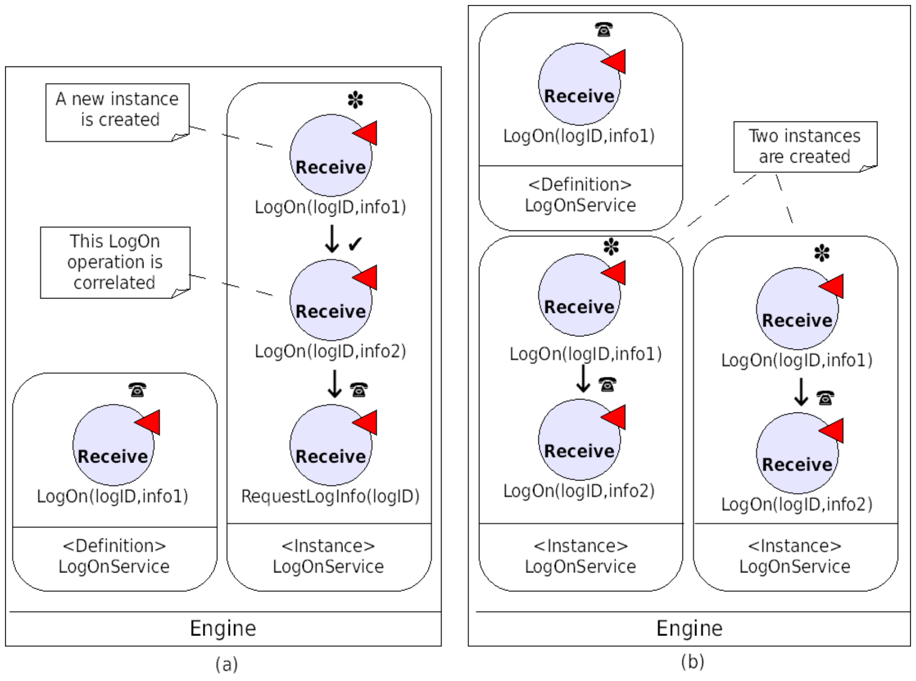


Figure 4: Message correlation: service instantiation

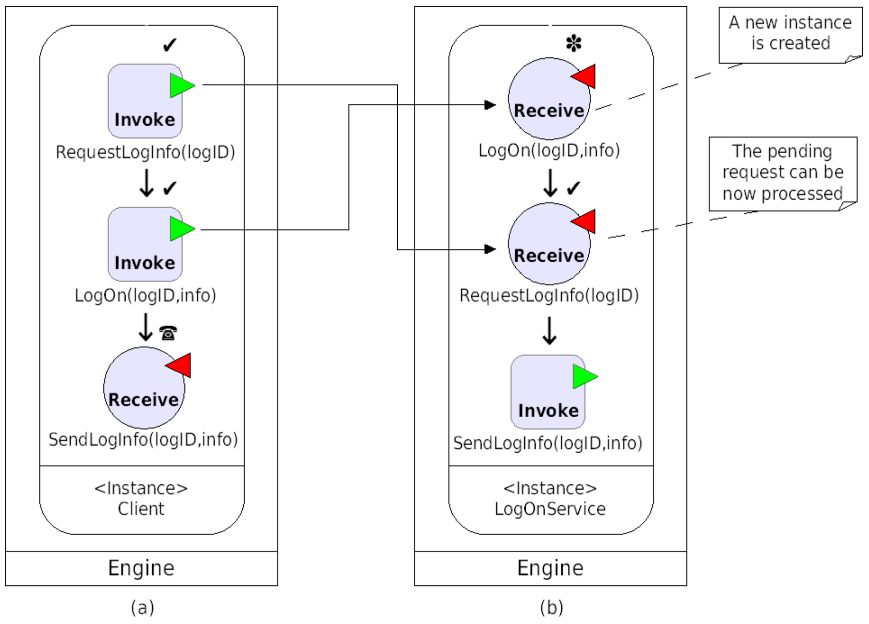


Figure 5: Asynchronous message delivering

invoke activity, a service instance is created as a result of consumption of the request for logging on the service produced by the second invoke activity as depicted in Figure 5(b). Now, the first produced message is not considered expired and, thus, can be consumed by the newly created service instance. All the examined WS-BPEL engines *tacitly* agree with this communication paradigm, although no requirement is explicitly reported in the WS-BPEL specification.

Example 2.3: Multiple start and conflicting receive activities. When defining services, the WS-BPEL specification allows for using multiple start activities [36, Section 10.4]. However, it is not clear how conflicting receive activities enabled at instantiation of such a service must be handled. To explain this point, consider a simple variant of service LogOnService, called MultiLogOnService, that allows two clients to log on the same service instance. Figure 6 illustrates two alternative definitions of MultiLogOnService with the same semantics: the one on the left hand side makes use of activity `<flow>`, while the one on the right hand side uses activity `<pick>`. In both definitions, the service waits for two log-on requests from clients along two different partner links and then, on demand by one of the two clients, provides logging information. After a message from a client, say client1, has been processed, an instance of the service is initiated as illustrated in Figure 7(a) (we only consider the case of the definition in Figure 6(a)). Now, the definition and the instance of the service compete for receiving the same message sent by another client that is correlated to that sent by client1 through the datum logID. In cases like this, the WS-BPEL specification requires the second message to be delivered to the existing instance, thus preventing creation of a new instance. In fact, the instance in Figure 7(a) can only reduce to that of Figure 7(b).

In case of conflicting receives, the WS-BPEL specification document prescribes to raise the standard fault `bpel:conflictingReceive`, which seems to be somehow in

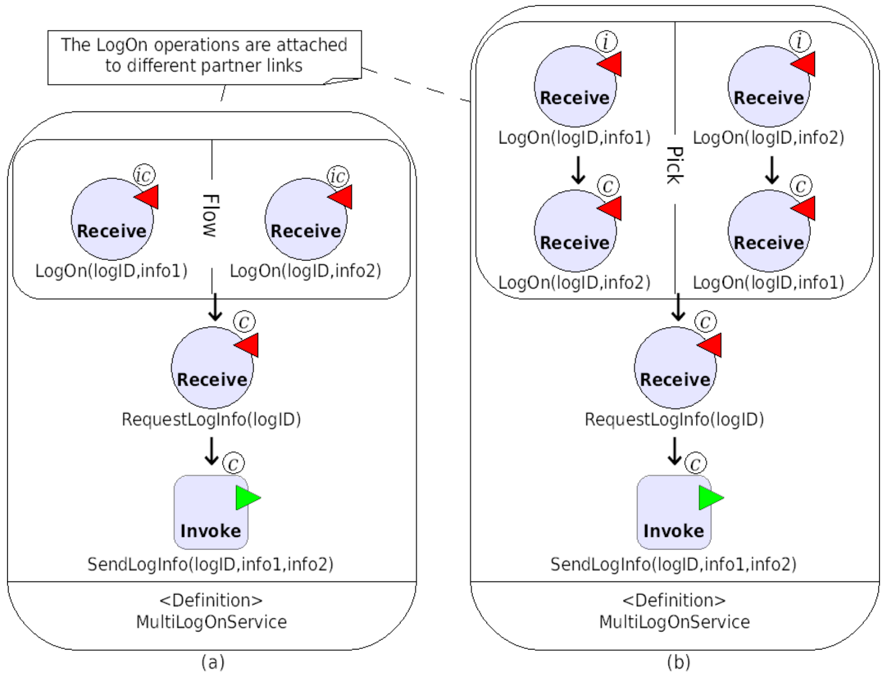


Figure 6: Multiple start activities

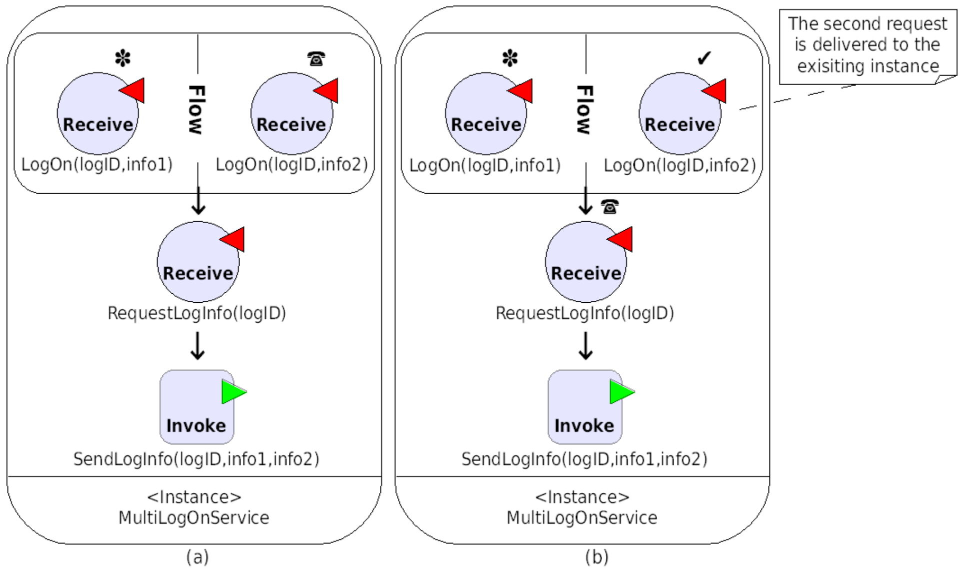


Figure 7: Multiple start activities: service instantiation

contrast with what we have illustrated before. In fact, this situation readily occurs when a service exploits multiple start activities, because of race conditions on incoming messages among the service definition and the created instances. However, in such cases, it does not seem fair to raise a fault because the correlation data contained within each incoming message should be sufficient to decide if the message has to be delivered to a specific instance or to the service definition. This is indeed a tricky question that leads the three engines we have considered to behave differently. Indeed, Oracle BPEL always raises the fault `bpel:conflictingReceive`, ActiveBPEL exploits correlation to enforce creation of only one service instance (just like the example in Figure 7), whereas Apache ODE does not currently support multiple start activities.

Example 2.4: Scheduling of parallel activities. While using the WS-BPEL engines, we have also experimented that they implement the flow activity in a different manner. For example, the expected behaviour of the WS-BPEL process in Figure 8(a) is that the three assignments updating a same shared variable are executed in an unpredictable order that may change in different executions. In fact, only Apache ODE implements this semantics, while the other two engines execute the assignments in an order fixed in advance, that is sequentially from left to right in case of ActiveBPEL (Figure 8(b)) and from right to left in case of Oracle BPEL (Figure 8(c)). As a consequence, we have that the parallel composition implemented by ActiveBPEL and Oracle BPEL is not a commutative operator.

Example 2.5: Forced termination. The WS-BPEL specification [36, Section 12.6] states: "The <`sequence`> and <`flow`> constructs *must* be terminated by terminating their behavior and applying termination to all nested activities currently active within them". This sentence is ambiguous because it is not clear what "nested activities currently active" means in case of termination due to <`exit`> or <`throw`> activities. For example, consider a sequence of two assign activities. In Oracle BPEL, termination prompted by a

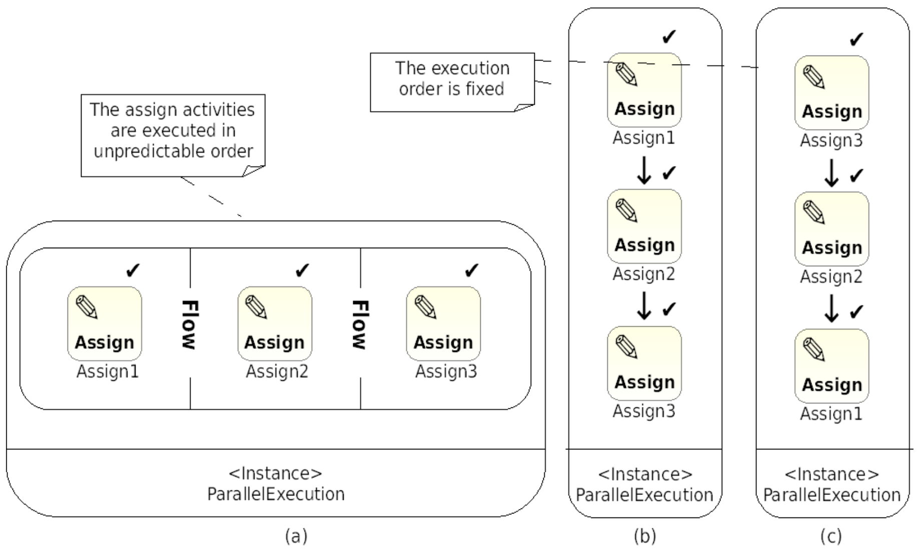
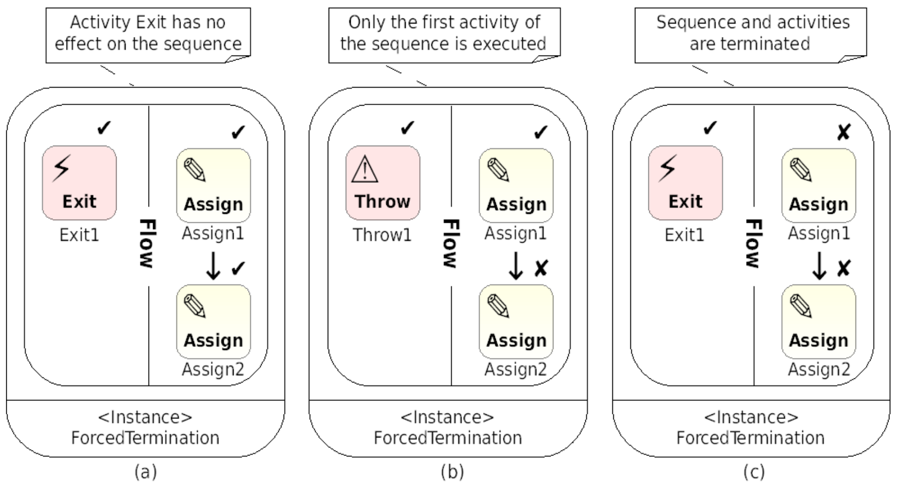


Figure 8: Scheduling of parallel activities

parallel <exit> activity has no effect on the sequence (Figure 9(a)), while termination prompted by a parallel <throw> activity causes execution of only the first assign activity (Figure 9(b)). ActiveBPEL is more compliant to WS-BPEL for which all currently running activities must be terminated as soon as possible (Figure 9(c)) without any fault handling or compensation [36, Section 10.10]. However, differently from what the WS-BPEL specification seems to suggests, ActiveBPEL does not distinguish *short-lived activities* (i.e. sufficiently brief activities that may be allowed to complete) from basic activities and makes them terminate in the same way. Finally, Apache ODE is fully compliant with WS-BPEL, since a termination activity function is applied to the continuation that only retains short-lived activities.

Example 2.6: Eager execution of activities causing termination. As shown in Example 2.5, to be compliant with the WS-BPEL requirement stating that termination activities must end immediately all currently running activities [36, Section 10.10], when defining the semantics of WS-BPEL, execution of activities <throw> and <exit> must have



Activity Exit has no effect on the sequence
Exit
Exit1
Flow
Assign
Assign1
Assign
Assign2
<Instance> ForcedTermination
(a)
Only the first activity of the sequence is executed
Throw
Throw1
Flow
Assign
Assign1
Assign
Assign2
<Instance> ForcedTermination
(b)
Sequence and activities are terminated
Exit
Exit1
Flow
Assign
Assign1
Assign
Assign2
<Instance> ForcedTermination
(c)

Figure 9: Forced termination

higher priority than execution of the remaining ones. Thus, for example, consider again a sequence of two assign activities. By executing a parallel `<throw>` activity, the whole process can only reduce as shown in Figure 10(a). While ActiveBPEL agrees with this requirement, Oracle BPEL and Apache ODE do not implement any prioritized behavior for activities forcing termination. Thus, the above process can also evolve by firstly performing the first `<assign>` activity and then the activity `<throw>`, as shown in Figure 10(b); this way, the first assign activity is not forced to terminate.

Example 2.7: Handlers protection. The structured activity in Figure 11 consists of a process with two inner parallel activities, one of which being a scope whose primary activity is a sequence of a scope and a `<throw>` activity (Throw1), while the other parallel activity is a basic `<throw>` activity (Throw2). Suppose that the innermost scope performs its assignment Assign1 and completes. Then, the associated compensation handler CH (i.e. the activity Assign2) is recorded into the default compensation activities of its enclosing scope. When execution of Throw1 rises a fault, then it is caught by the corresponding fault handler that activates the default compensation that consists of execution of Assign2.

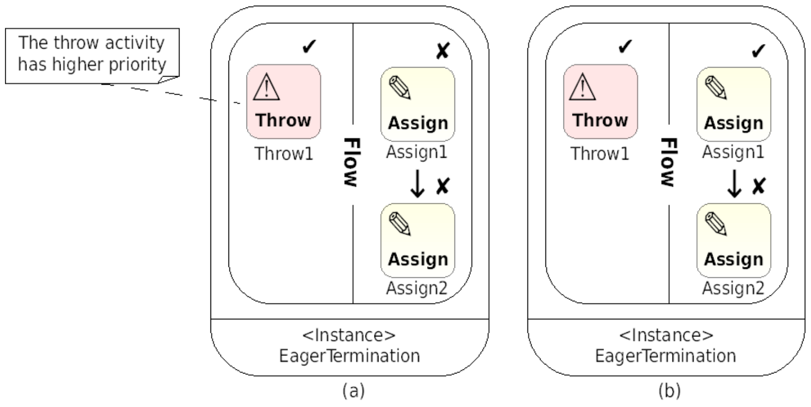


Figure 10: Eager execution of activities causing termination

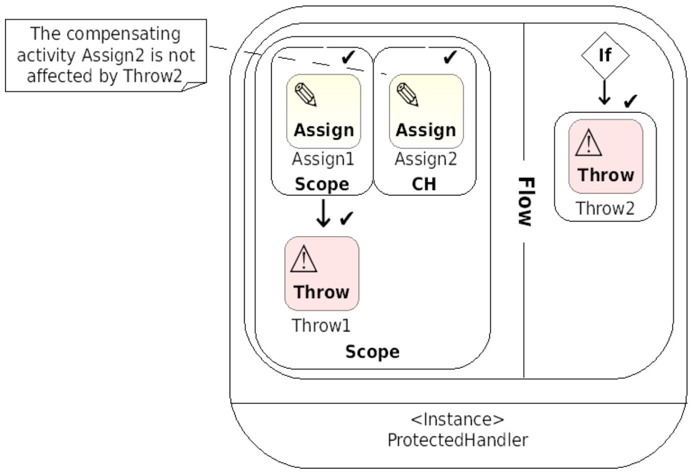


Figure 11: Handlers protection

This activity can be effectively executed since it is appropriately protected from the effect of execution of the parallel activity Throw2.

We end by remarking two aspects of the compensation mechanism prescribed by the WS-BPEL specification [36, Sections 12.5 and 10.10]. Firstly, compensation handlers of faultily terminated scopes should not be installed. Secondly, fault and compensation handlers should not be affected by the activities causing termination. Both aspects are not faithfully implemented in Oracle BPEL, while ActiveBPEL and Apache ODE meet these

	Oracle BPEL	ActiveBPEL	Apache ODE
Message correlation (Ex. 1)	+	+	+
Consecutive conflicting receives (Ex. 1)	−	+	+
Asynchronous message delivering (Ex. 2)	+	+	+
Multiple start (Ex. 3)	−	+	−
Scheduling of parallel activities (Ex. 4)	−	−	+
Short-lived activities (Ex. 5)	+	−	+
Forced Termination (Ex. 5)	−	+	+
Eager execution (Ex. 6)	−	+	−
Handlers protection and installation (Ex. 7)	−	+	+

Table 1: Experiment results on the tested WS-BPEL engines

specific requirements and adhere to the intended WS-BPEL semantics.

Evaluation. The results of our experiments, summarized in Table 1, point out that the engines we have experimented with implement different semantics. From these results it is clear that no engine passes all the selected experiments. In fact, all the three engines only support a subset of the proposed orchestration patterns. Specifically, it is worth noticing the limited support for the multiple start pattern and the eager execution of Oracle BPEL and Apache ODE.

The engines we have used range over different periods of time. In particular, Oracle BPEL is the oldest and best advertised engine, while Apache ODE is a relative newcomer. The fact that, instead, ActiveBPEL seems to be the best established product has led us to choice it for implementing and deploying B*lite* applications.

We believe that the engines' different beaviours we have experimented are a consequence of the lack of a precise semantics for WS-BPEL, that would have disambiguated the intricate and complex features of the language, leaving less room for interpretation by implementers. Therefore our work, and works with similar goals, other than as a guide for developing implementations fully compliant with the intended semantics since the early stages, can also be used for making future versions of existing implementations more compatible.

We end our evaluation with some observations on the procedure to deploy WS-BPEL programs, although the description of the deployment is out of scope of the WS-BPEL specification document [36]. A WS-BPEL process is designed to be a reusable definition that can be deployed in different ways within different scenarios. In these respects the three tested engines pose different requirements. ActiveBPEL provides deployment information (i.e. partner link bindings and address information) in terms of abstract WS-BPEL elements (i.e. partner links and partner roles), while Apache ODE and Oracle BPEL Process Manager use proprietary defined elements to describe a deployment, regardless of whether the same elements are declared at WS-BPEL level. The integration of different deployment documents is then impossible to obtain, which is another factor that reduces the level of portability a programmer might expect.

3. B*lite*: a 'lightweight' variant of WS-BPEL

B*lite* is a prototypical orchestration language whose design is closely inspired to WS-BPEL. It is the result of the tension between handiness and expressiveness which is typical when designing a formalism. In fact, the set of WS-BPEL constructs is not intended to be a minimal one. For example, such constructs as request-response operations and flow links are redundant and can be reasonably expressed in terms of other ones. On the contrary, the design of B*lite* only retains the core, and in our opinion absolutely necessary, features of WS-BPEL, like partners and partner links, message correlation, concurrency, service instance creation/identification, long-running business transactions, termination and compensation handlers. Indeed, to keep the design of the language manageable, we intentionally left out other aspects of WS-BPEL, including timeouts, event and termination handlers, and sophisticated forms of data handling.

B*lite* provides a formal description of service deployments by only retaining relevant implementation details such as partner links and correlation sets. Thus, *partner links* and *partners* explicitly indicate the roles played by service partners in a service interaction, while such aspects as physical *service binding* described in associated WSDL documents are abstracted away. As we will see later on in Section 4.2, such information are dealt with separately in the declarative parts associated to B*lite* specifications, in order to allow B*lite*C to generate the corresponding WSDL documents and deployment descriptors.

3.1. Syntax

The syntax of B*lite* is given in Table 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, pick (with the constraint that $|J| > 1$), parallel composition, 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* r are structured activities that initially can only execute receive activities.

In the sequel, we shall use $\cdot + \cdot$ to abbreviate binary pick. We let sequence have higher priority (i.e. bind more tightly) than parallel composition and pick, i.e. $\mathsf{a}_1 ; \mathsf{a}_2 \,|\, \mathsf{a}_3 ; \mathsf{a}_4$ stands for $(\mathsf{a}_1 ; \mathsf{a}_2) \,|\, (\mathsf{a}_3 ; \mathsf{a}_4)$ and $\mathsf{a}_1 ; \mathsf{a}_2 + \mathsf{a}_3$ stands for $(\mathsf{a}_1 ; \mathsf{a}_2) + \mathsf{a}_3$. Moreover, we adopt the convention that fault and compensation activities may be omitted from a scope construct, in which case they are intended to be throw and empty, respectively.

Data can be shared among different activities through *shared variables* (ranged over by $\mathsf{x}, \mathsf{x}', \ldots$). The set of manipulable values (ranged over by $\mathsf{v}, \mathsf{v}', \ldots$) is left unspecified; however, we assume that it includes the set of *partner names* (ranged over by $\mathsf{p}, \mathsf{q}, \ldots$) and

Basic activities	$\mathsf{b} ::=$	$\mathsf{inv}\,\ell^i\,\mathsf{o}\,\bar{\mathsf{x}} \mid \mathsf{rcv}\,\ell^r\,\mathsf{o}\,\bar{\mathsf{x}} \mid \mathsf{x} := \mathsf{e}$	invoke, receive, assign
		$\mid \mathsf{empty} \mid \mathsf{throw} \mid \mathsf{exit}$	empty, throw, exit
Structured activities	$\mathsf{a} ::=$	$\mathsf{b} \mid \mathsf{if(e)}\{\mathsf{a}_1\}\{\mathsf{a}_2\} \mid \mathsf{while(e)}\,\{\mathsf{a}\}$	basic, conditional, iteration
		$\mid \mathsf{a}_1\,;\mathsf{a}_2 \mid \sum_{j\in J} \mathsf{rcv}\,\ell_j^r\,\mathsf{o}_j\,\bar{\mathsf{x}}_j\,;\mathsf{a}_j$	sequence, pick (with $\mid J\mid > 1$)
		$\mid \mathsf{a}_1 \mid \mathsf{a}_2 \mid [\mathsf{a} \bullet \mathsf{a}_f \star \mathsf{a}_c]$	parallel, scope
Start activities	$\mathsf{r} ::=$	$\mathsf{rcv}\,\ell^r\,\mathsf{o}\,\bar{\mathsf{x}} \mid \sum_{j\in J} \mathsf{rcv}\,\ell_j^r\,\mathsf{o}_j\,\bar{\mathsf{x}}_j\,;\mathsf{a}_j$	receive, pick
		$\mid \mathsf{r}\,;\mathsf{a} \mid \mathsf{r}_1 \mid \mathsf{r}_2 \mid [\mathsf{r} \bullet \mathsf{a}_f \star \mathsf{a}_c]$	sequence, parallel, scope
Services	$\mathsf{s} ::=$	$[\mathsf{r} \bullet \mathsf{a}_f] \mid \mu \vdash \mathsf{a} \mid \mu \vdash \mathsf{a}\,,\,\mathsf{s}$	definition, instance, multiset
Deployments	$\mathsf{d} ::=$	$\{\mathsf{s}\}_\mathsf{c} \mid \mathsf{d}_1 \parallel \mathsf{d}_2$	deployment, composition

Table 2: Syntax of B*lite*

the set of *operation names* (ranged over by $\mathsf{o}, \mathsf{o}', \ldots$). We use u to range over partners and variables and w to range over values and variables. *Expressions* (ranged over by $\mathsf{e}, \mathsf{e}', \ldots$) are left unspecified but contain, at least, values and variables.

Notation $\bar{\cdot}$ stands for tuples of objects, e.g. $\bar{\mathsf{x}}$ is a compact notation for denoting the tuple of variables $\langle \mathsf{x}_1, \ldots, \mathsf{x}_h \rangle$ (with $h \geq 0$). We assume that variables in the same tuple are pairwise distinct. The special notation $\tilde{\cdot}$ stands for tuples of one or two objects, e.g. $\tilde{\mathsf{p}}$ denotes either $\langle \mathsf{p}_1, \mathsf{p}_2 \rangle$ or $\langle \mathsf{p}_1 \rangle$. Tuples can be constructed using a concatenation operator $\cdot : \cdot$, i.e. $\langle \mathsf{p}, \mathsf{u} \rangle : \langle \mathsf{x}_1, \ldots, \mathsf{x}_h \rangle$ returns $\langle \mathsf{p}, \mathsf{u}, \mathsf{x}_1, \ldots, \mathsf{x}_h \rangle$. We will write $Z \triangleq W$ to assign a symbolic name Z to the term W.

Partner links ℓ^r of receive activities can be either $\langle \mathsf{p} \rangle$ or $\langle \mathsf{p}, \mathsf{u} \rangle$, where p is the partner providing the operation and u is a partner or variable used to send messages in reply. 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. Partner links ℓ^i within invoke activities can be either $\langle \mathsf{u} \rangle$ or $\langle \mathsf{u}, \mathsf{p} \rangle$, where u is the partner providing the operation and p is a partner used to receive messages in reply. As before, this latter partner must be statically known, thus it cannot be a variable.

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

Deployments are finite compositions of multisets of service *instances* $\mu \vdash \mathsf{a}$, containing at most one service *definition* $[\mathsf{r} \bullet \mathsf{a}_f]$ and associated to a *correlation set* c, namely a (possibly empty) set of *correlation variables*. A service definition provides a 'top-level' scope (i.e. a scope that cannot be compensated) and offers a choice of alternative receives among multiple start activities. Each service instance $\mu \vdash \mathsf{a}$ has its own (private) state μ. States are (partial) functions mapping variables to values and are written as collections of pairs of the form $\{\mathsf{x} \mapsto \mathsf{v}\}$. The state obtained by updating μ with μ', written as $\mu \circ \mu'$, is inductively defined by: $\mu \circ \mu'(\mathsf{x}) = \mu'(\mathsf{x})$ if $\mathsf{x} \in \mathit{dom}(\mu')$ (where $\mathit{dom}(\mu)$ denotes the domain of μ) and $\mu(\mathsf{x})$ otherwise. The empty state is denoted by $\emptyset$. In the sequel, we will only consider *well-formed* deployments, i.e. compositions where the sets of partners used for handling requests within different deployments are pairwise disjoint. The rationale is that each service definition has its own partner names and all its instances run within the same deployment where the definition resides.

$$\mathsf{a} \,|\, \mathsf{empty} \equiv \mathsf{a} \qquad \mathsf{empty} \,;\, \mathsf{a} \equiv \mathsf{a} \,;\, \mathsf{empty} \equiv \mathsf{a} \qquad \mathsf{stop} \,|\, \mathsf{stop} \equiv \mathsf{stop} \qquad \mathsf{stop} \,;\, \mathsf{a} \equiv \mathsf{stop}$$

$$(\!|(\!|\mathsf{a}|\!)|\!) \equiv (\!|\mathsf{a}|\!) \qquad (\!|\mathsf{sh}|\!) \equiv \mathsf{sh} \qquad (\!|\ll \tilde{\mathsf{p}} : \mathsf{o} : \bar{\mathsf{v}} \gg \,|\, \mathsf{a}|\!) \equiv \ll \tilde{\mathsf{p}} : \mathsf{o} : \bar{\mathsf{v}} \gg \,|\, (\!|\mathsf{a}|\!)$$

$$[\mathsf{a} \bullet \mathsf{a}_f \star \mathsf{a}_c] \equiv [\mathsf{a} \bullet \mathsf{a}_f \star \mathsf{a}_c \vartriangle \mathsf{empty}] \qquad (\ll \tilde{\mathsf{p}} : \mathsf{o} : \bar{\mathsf{v}} \gg \,|\, \mathsf{a}_1) \,;\, \mathsf{a}_2 \equiv \ll \tilde{\mathsf{p}} : \mathsf{o} : \bar{\mathsf{v}} \gg \,|\, (\mathsf{a}_1 \,;\, \mathsf{a}_2)$$

$$[\ll \tilde{\mathsf{p}} : \mathsf{o} : \bar{\mathsf{v}} \gg \,|\, \mathsf{a} \bullet \mathsf{a}_f \star \mathsf{a}_c \vartriangle \mathsf{a}_d] \equiv \ll \tilde{\mathsf{p}} : \mathsf{o} : \bar{\mathsf{v}} \gg \,|\, [\mathsf{a} \bullet \mathsf{a}_f \star \mathsf{a}_c \vartriangle \mathsf{a}_d] \quad \text{if } \neg \mathsf{a} \Downarrow_{\mathsf{throw}}$$

$$\frac{\mathsf{a} \equiv \mathsf{a}' \quad \mathsf{a}_f \equiv \mathsf{a}'_f \quad \mathsf{a}_c \equiv \mathsf{a}'_c \quad \mathsf{a}_d \equiv \mathsf{a}'_d}{[\mathsf{a} \bullet \mathsf{a}_f \star \mathsf{a}_c \vartriangle \mathsf{a}_d] \equiv [\mathsf{a}' \bullet \mathsf{a}'_f \star \mathsf{a}'_c \vartriangle \mathsf{a}'_d]}$$

$$\frac{\mathsf{r} \equiv \mathsf{r}' \qquad \mathsf{a}_f \equiv \mathsf{a}'_f}{\{[\mathsf{r} \bullet \mathsf{a}_f]\,,\, \mathsf{s}\}_{\mathsf{c}} \equiv \{\mathsf{s}\,,\, [\mathsf{r}' \bullet \mathsf{a}'_f]\}_{\mathsf{c}}} \qquad \frac{\mathsf{a} \equiv \mathsf{a}'}{\{\mu \vdash \mathsf{a}\,,\, \mathsf{s}\}_{\mathsf{c}} \equiv \{\mathsf{s}\,,\, \mu \vdash \mathsf{a}'\}_{\mathsf{c}}}$$

$$\mathsf{d}_1 \| \mathsf{d}_2 \equiv \mathsf{d}_2 \| \mathsf{d}_1 \qquad (\mathsf{d}_1 \| \mathsf{d}_2) \| \mathsf{d}_3 \equiv \mathsf{d}_1 \| (\mathsf{d}_2 \| \mathsf{d}_3) \qquad \{\mu \vdash \mathsf{empty}\,,\, \mathsf{s}\}_{\mathsf{c}} \equiv \{\mathsf{s}\}_{\mathsf{c}}$$

$$\{\mu \vdash \mathsf{stop}\,,\, \mathsf{s}\}_{\mathsf{c}} \equiv \{\mathsf{s}\}_{\mathsf{c}} \qquad \{\mu \vdash \mathsf{empty}\}_{\mathsf{c}} \| \mathsf{d} \equiv \mathsf{d} \qquad \{\mu \vdash \mathsf{stop}\}_{\mathsf{c}} \| \mathsf{d} \equiv \mathsf{d}$$

Table 3: Structural congruence for B*lite* activities and deployments

3.2. Operational semantics

The semantics is defined over an enriched syntax that also includes *protected activities* $(\!|\mathsf{a}|\!)$, *unsuccessful termination* stop, *messages* $\ll \tilde{\mathsf{p}} : \mathsf{o} : \bar{\mathsf{v}} \gg$ and *scopes* of the form $[\mathsf{a} \bullet \mathsf{a}_f \star \mathsf{a}_c \vartriangle \mathsf{a}_d]$. The first three 'auxiliary' activities are used to replace, respectively, unsuccessfully completed scopes (with their protected default compensation), compulsorily or faultily terminated services (with stop), and invoke activities (with the message they produced). Instead, such scopes as $[\mathsf{a} \bullet \mathsf{a}_f \star \mathsf{a}_c \vartriangle \mathsf{a}_d]$ are dynamically generated to store in a_d the compensation activities of the immediately enclosed scopes that have successfully completed, together with the order in which they must be executed. In the sequel, empty, exit, throw, stop and messages will be called *short-lived* activities and will be generically indicated by sh.

The operational semantics of B*lite* deployments is defined in terms of a structural congruence and a reduction relation. The *structural congruence*, written $\equiv$, identifies syntactically different terms which intuitively represent the same term. It is defined as the least congruence relation induced by a given set of equational laws. In Table 3, we explicitly show, in the upper part, the laws for empty, stop, protected activities, messages and scopes, and, in the lower part, the laws for services and deployments. Standard laws stating, e.g., that sequence is associative, parallel composition is commutative and associative, are omitted. A few observations on the structural laws are in order. Activity empty acts as the identity element both for sequence and parallel composition. Multiple stop in parallel are equivalent to just one stop, moreover stop disables subsequent activities. The protection operator is idempotent, and short-lived activities are implicitly protected, thus messages can go in/out of the scope of a protection operator. Default compensation is initially empty. Messages do not block subsequent activities and scope completion, except when throw is active in the scope (this is checked by predicate $\cdot \Downarrow_{\mathsf{throw}}$ that will be explained later on). Structural congruence is extended to scopes, instances and deployments in the obvious way. Moreover, the order in which definition and instances occur within a deployment does not matter, and deployment composition is commutative and associative. Instances like $\mu \vdash \mathsf{empty}$ and $\mu \vdash \mathsf{stop}$ are terminated and, thus, can be removed. Similarly, deployments only containing terminated instances are terminated too and can be removed.

The *reduction relation* over deployments, written $\rightarrowtail\!\!\triangleright$, exploits a labelled transition relation over structured activities, written $\xrightarrow{\alpha}\!\!\triangleright$, where α is generated by the grammar:

$$\alpha \ ::= \ \tau \ \mid \ \mathsf{x} \leftarrow \mathsf{v} \ \mid \ !\,\tilde{\mathsf{p}}:\mathsf{o}:\bar{\mathsf{v}} \ \mid \ ?\,\ell^r:\mathsf{o}:\bar{\mathsf{x}} \ \mid \ \boxast \ \mid \ \Rsh \ \mid \ (\mathsf{a})$$

The meaning of labels is as follows: τ indicates message productions, guard evaluations for conditional and iteration or installation/activation of compensations; $\mathsf{x} \leftarrow \mathsf{v}$ indicates assignment of value v to variable x; $!\,\tilde{\mathsf{p}} : \mathsf{o} : \bar{\mathsf{v}}$ and $?\,\ell^r : \mathsf{o} : \bar{\mathsf{x}}$ indicate execution of invoke and receive activities for operation o, where $\tilde{\mathsf{p}}$ and $\bar{\mathsf{v}}$ match with ℓ^r and $\bar{\mathsf{x}}$, respectively;

$$\mu \vdash \mathrm{inv}\ \ell^i\ \mathrm{o}\ \bar{\mathrm{x}} \xrightarrow{\tau} \ll\mu(\ell^i):\mathrm{o}:\mu(\bar{\mathrm{x}})\gg \quad \text{(inv)} \qquad \mathrm{rcv}\ \ell^r\ \mathrm{o}\ \bar{\mathrm{x}} \xrightarrow{?\,\ell^r:\mathrm{o}:\bar{\mathrm{x}}} \mathrm{empty} \quad \text{(rec)}$$

$$\mu \vdash \mathrm{x} := \mathrm{e} \xrightarrow{\mathrm{x} \leftarrow \mu(\mathrm{e})} \mathrm{empty} \quad \text{(asg)} \qquad \mathrm{throw} \xrightarrow{\Rsh} \mathrm{stop} \quad \text{(thr)}$$

$$\mathrm{exit} \xrightarrow{\boxast} \mathrm{stop} \quad \text{(term)} \qquad \ll\tilde{\mathrm{p}}:\mathrm{o}:\bar{\mathrm{v}}\gg \xrightarrow{!\,\tilde{\mathrm{p}}:\mathrm{o}:\bar{\mathrm{v}}} \mathrm{empty} \quad \text{(msg)} \qquad \frac{\mu \vdash \mathrm{a} \xrightarrow{\alpha} \mathrm{a}'}{\mu \vdash \llparenthesis \mathrm{a} \rrparenthesis \xrightarrow{\alpha} \llparenthesis \mathrm{a}' \rrparenthesis} \quad \text{(prot)}$$

$$\frac{\mu \vdash \mathrm{a}_1 \xrightarrow{\alpha} \mathrm{a}_1'}{\mu \vdash \mathrm{a}_1 ; \mathrm{a}_2 \xrightarrow{\alpha} \mathrm{a}_1' ; \mathrm{a}_2} \quad \text{(seq)} \qquad \textstyle\sum_{j\in J} \mathrm{rcv}\ \ell_j^r\ \mathrm{o}_j\ \bar{\mathrm{x}}_j ; \mathrm{a}_j \xrightarrow{?\,\ell_h^r:\mathrm{o}_h:\bar{\mathrm{x}}_h} \mathrm{a}_h \quad (h \in J) \quad \text{(pick)}$$

$$\frac{\mathrm{a} = \begin{cases} \mathrm{a}_1 & \text{if } \mu(\mathrm{e}) = \mathrm{tt} \\ \mathrm{a}_2 & \text{if } \mu(\mathrm{e}) = \mathrm{ff} \end{cases}}{\mu \vdash \mathrm{if(e)\{a_1\}\{a_2\}} \xrightarrow{\tau} \mathrm{a}} \quad \text{(if)} \qquad \frac{\mathrm{a}' = \begin{cases} \mathrm{a} ; \mathrm{while(e)\,\{a\}} & \text{if } \mu(\mathrm{e}) = \mathrm{tt} \\ \mathrm{empty} & \text{if } \mu(\mathrm{e}) = \mathrm{ff} \end{cases}}{\mu \vdash \mathrm{while(e)\,\{a\}} \xrightarrow{\tau} \mathrm{a}'} \quad \text{(while)}$$

$$\frac{\mu \vdash \mathrm{a}_1 \xrightarrow{\alpha} \mathrm{a}_1' \quad \alpha \notin \{\boxast, \Rsh\} \quad \neg(\mathrm{a}_2 \Downarrow_{\mathrm{throw}} \vee \mathrm{a}_2 \Downarrow_{\mathrm{exit}})}{\mu \vdash \mathrm{a}_1 \mid \mathrm{a}_2 \xrightarrow{\alpha} \mathrm{a}_1' \mid \mathrm{a}_2} \quad (\mathrm{par}_1) \qquad \frac{\mathrm{a}_1 \xrightarrow{\alpha} \mathrm{a}_1' \quad \alpha \in \{\boxast, \Rsh\}}{\mathrm{a}_1 \mid \mathrm{a}_2 \xrightarrow{\alpha} \mathrm{a}_1' \mid \mathrm{end}(\mathrm{a}_2)} \quad (\mathrm{par}_2)$$

$$[\mathrm{empty} \bullet \mathrm{a}_f \star \mathrm{a}_c \vartriangle \mathrm{a}_d] \xrightarrow{(\mathrm{a}_c)} \mathrm{empty} \quad (\mathrm{done}_1) \qquad [\mathrm{stop} \bullet \mathrm{a}_f \star \mathrm{a}_c \vartriangle \mathrm{a}_d] \xrightarrow{\tau} \llparenthesis \mathrm{a}_d ; \mathrm{a}_f \rrparenthesis \quad (\mathrm{done}_2)$$

$$\frac{\mu \vdash \mathrm{a} \xrightarrow{\alpha} \mathrm{a}' \quad \alpha \notin \{\Rsh, (\mathrm{a}'')\}}{\mu \vdash [\mathrm{a} \bullet \mathrm{a}_f \star \mathrm{a}_c \vartriangle \mathrm{a}_d] \xrightarrow{\alpha} [\mathrm{a}' \bullet \mathrm{a}_f \star \mathrm{a}_c \vartriangle \mathrm{a}_d]} \quad \text{(exec)}$$

$$\frac{\mathrm{a} \xrightarrow{(\mathrm{a}'')} \mathrm{a}'}{[\mathrm{a} \bullet \mathrm{a}_f \star \mathrm{a}_c \vartriangle \mathrm{a}_d] \xrightarrow{\tau} [\mathrm{a}' \bullet \mathrm{a}_f \star \mathrm{a}_c \vartriangle \mathrm{a}'' ; \mathrm{a}_d]} \quad (\mathrm{done}_3)$$

$$\frac{\mathrm{a} \xrightarrow{\Rsh} \mathrm{a}'}{[\mathrm{a} \bullet \mathrm{a}_f \star \mathrm{a}_c \vartriangle \mathrm{a}_d] \xrightarrow{\tau} [\mathrm{a}' \bullet \mathrm{a}_f \star \mathrm{a}_c \vartriangle \mathrm{a}_d]} \quad \text{(fault)}$$

Table 4: Basic, auxiliary and structured activities

$\boxast$ indicates forced termination of a service instance; $\Rsh$ indicates production of a fault signal from inside a scope; (a) indicates successful completion of a scope that can be compensated by the structured activity a.

The relation $\xrightarrow{\alpha}$ is defined by the rules in Table 4 with respect to a state μ, that is omitted when unnecessary (writing $\mathrm{a} \xrightarrow{\alpha} \mathrm{a}'$ instead of $\mu \vdash \mathrm{a} \xrightarrow{\alpha} \mathrm{a}'$). Before commenting the rules, we introduce the auxiliary functions and predicates they exploit. Specifically, function $\mu(\mathsf{e})$ evaluates expression e with respect to the state μ and returns the computed value. However, $\mu(\cdot)$ cannot be explicitly defined because the exact syntax of expressions is deliberately not specified. Predicates $\mathsf{a}\Downarrow_{\mathsf{exit}}$ and $\mathsf{a}\Downarrow_{\mathsf{throw}}$ check the ability of a of performing exit or throw, respectively. They are defined inductively on the syntax of activities and hold false in all cases but for the following ones

$$\mathsf{exit}\Downarrow_{\mathsf{exit}} \qquad \frac{\mathsf{a}_1\Downarrow_{\mathsf{exit}}}{\mathsf{a}_1\,;\mathsf{a}_2\Downarrow_{\mathsf{exit}}} \qquad \frac{\mathsf{a}\Downarrow_{\mathsf{exit}}}{(\!|\mathsf{a}|\!)\Downarrow_{\mathsf{exit}}} \qquad \frac{\mathsf{a}_1\Downarrow_{\mathsf{exit}}\ \vee \mathsf{a}_2\Downarrow_{\mathsf{exit}}}{\mathsf{a}_1\,|\,\mathsf{a}_2\Downarrow_{\mathsf{exit}}} \qquad \frac{\mathsf{a}\Downarrow_{\mathsf{exit}}}{[\mathsf{a}\bullet\mathsf{a}_f\star\mathsf{a}_c]\Downarrow_{\mathsf{exit}}}$$

$$\mathsf{throw}\Downarrow_{\mathsf{throw}} \qquad \frac{\mathsf{a}_1\Downarrow_{\mathsf{throw}}}{\mathsf{a}_1\,;\mathsf{a}_2\Downarrow_{\mathsf{throw}}} \qquad \frac{\mathsf{a}\Downarrow_{\mathsf{throw}}}{(\!|\mathsf{a}|\!)\Downarrow_{\mathsf{throw}}} \qquad \frac{\mathsf{a}_1\Downarrow_{\mathsf{throw}}\ \vee \mathsf{a}_2\Downarrow_{\mathsf{throw}}}{\mathsf{a}_1\,|\,\mathsf{a}_2\Downarrow_{\mathsf{throw}}} \qquad \frac{\mathsf{a}\Downarrow_{\mathsf{exit}}}{[\mathsf{a}\bullet\mathsf{a}_f\star\mathsf{a}_c\vartriangle\mathsf{a}_d]\Downarrow_{\mathsf{exit}}}$$

$$\frac{\mathsf{a}_1\Downarrow_{\mathsf{exit}}\ \wedge \mathsf{a}_1\equiv\mathsf{a}_2}{\mathsf{a}_2\Downarrow_{\mathsf{exit}}} \qquad \frac{\mathsf{a}_1\Downarrow_{\mathsf{throw}}\ \wedge \mathsf{a}_1\equiv\mathsf{a}_2}{\mathsf{a}_2\Downarrow_{\mathsf{throw}}}$$

Function $\mathsf{end}(\cdot)$, given an activity a, returns the activity obtained by only retaining short-lived and protected activities inside a. It is defined inductively on the syntax of activities, the most significant cases being

$$\mathsf{end}(\mathsf{sh})=\mathsf{sh} \qquad \mathsf{end}((\!|\mathsf{a}|\!))=(\!|\mathsf{a}|\!) \qquad \mathsf{end}([\mathsf{a}\bullet\mathsf{a}_f\star\mathsf{a}_c])=[\mathsf{end}(\mathsf{a})\bullet\mathsf{a}_f\star\mathsf{a}_c]$$

$$\mathsf{end}(\mathsf{a}_1\,;\mathsf{a}_2)=\mathsf{end}(\mathsf{a}_1) \qquad \mathsf{end}([\mathsf{a}\bullet\mathsf{a}_f\star\mathsf{a}_c\vartriangle\mathsf{a}_d])=[\mathsf{end}(\mathsf{a})\bullet\mathsf{a}_f\star\mathsf{a}_c\vartriangle\mathsf{a}_d]$$

where a_1 may not be congruent to empty or to $\ll\tilde{\mathsf{p}}:\mathsf{o}:\bar{\mathsf{v}}\gg$, or to parallel compositions of them. In the remaining cases, $\mathsf{end}(\cdot)$ returns stop, except for parallel composition for which it acts as an homomorphism. Like the two predicates above, function $\mathsf{end}(\cdot)$ is closed under $\equiv$, i.e. $\mathsf{end}(\mathsf{a}_1)=\mathsf{a}$ and $\mathsf{a}_1\equiv\mathsf{a}_2$ imply $\mathsf{end}(\mathsf{a}_2)=\mathsf{a}$.

We now briefly comment on the rules in Table 4. Rules (inv) and (asg) state that invoke and assign activities can proceed only if their arguments are *closed* expressions (i.e. expressions without uninitialized variables) and can be evaluated (i.e. $\mu(\cdot)$ returns a value). By rule (rec), a receive activity offers an invocable operation along a given partner link.

$$\mathsf{match}(\mathsf{c},\mu,\mathsf{v},\mathsf{v}) = \emptyset \qquad \mathsf{match}(\mathsf{c},\mu,\mathsf{x},\mathsf{v}) = \begin{cases} \{\mathsf{x}\mapsto\mathsf{v}\} & \text{if } \mathsf{x}\notin\mathsf{c} \vee (\mathsf{x}\in\mathsf{c} \wedge \mathsf{x}\notin dom(\mu)) \\ \emptyset & \text{if } \mathsf{x}\in\mathsf{c} \wedge \{\mathsf{x}\mapsto\mathsf{v}\}\in\mu \end{cases}$$

$$\mathsf{match}(\mathsf{c},\mu,\langle\rangle,\langle\rangle) = \emptyset \qquad \frac{\mathsf{match}(\mathsf{c},\mu,\mathsf{w}_1,\mathsf{v}_1) = \mu' \quad \mathsf{match}(\mathsf{c},\mu,\bar{\mathsf{w}}_2,\bar{\mathsf{v}}_2) = \mu''}{\mathsf{match}(\mathsf{c},\mu,(\mathsf{w}_1,\bar{\mathsf{w}}_2),(\mathsf{v}_1,\bar{\mathsf{v}}_2)) = \mu'\circ\mu''}$$

$$\frac{|\mathsf{match}(\mathsf{c},\mu,\ell^r\!:\!\mathsf{o}\!:\!\bar{\mathsf{x}},\tilde{\mathsf{p}}\!:\!\mathsf{o}\!:\!\bar{\mathsf{v}})| < n}{\mu \vdash \mathsf{rcv}\,\ell^r\,\mathsf{o}\,\bar{\mathsf{x}}\,;\mathsf{a} \Downarrow^{\mathsf{c},n}_{\tilde{\mathsf{p}}:\mathsf{o}:\bar{\mathsf{v}}}} \qquad \frac{\exists h\in J\,.\ |\mathsf{match}(\mathsf{c},\mu,\ell^r_h\!:\!\mathsf{o}_h\!:\!\bar{\mathsf{x}}_h,\tilde{\mathsf{p}}\!:\!\mathsf{o}\!:\!\bar{\mathsf{v}})| < n}{\mu \vdash \sum_{j\in J}\mathsf{rcv}\,\ell^r_j\,\mathsf{o}_j\,\bar{\mathsf{x}}_j\,;\mathsf{a}_j \Downarrow^{\mathsf{c},n}_{\tilde{\mathsf{p}}:\mathsf{o}:\bar{\mathsf{v}}}}$$

$$\frac{\mu\vdash\mathsf{a}_1\Downarrow^{\mathsf{c},n}_{\tilde{\mathsf{p}}:\mathsf{o}:\bar{\mathsf{v}}}}{\mu\vdash\mathsf{a}_1\,;\mathsf{a}_2\Downarrow^{\mathsf{c},n}_{\tilde{\mathsf{p}}:\mathsf{o}:\bar{\mathsf{v}}}} \qquad \frac{\mu\vdash\mathsf{a}_1\Downarrow^{\mathsf{c},n}_{\tilde{\mathsf{p}}:\mathsf{o}:\bar{\mathsf{v}}} \ \vee\ \mu\vdash\mathsf{a}_2\Downarrow^{\mathsf{c},n}_{\tilde{\mathsf{p}}:\mathsf{o}:\bar{\mathsf{v}}}}{\mu\vdash\mathsf{a}_1\,|\,\mathsf{a}_2\Downarrow^{\mathsf{c},n}_{\tilde{\mathsf{p}}:\mathsf{o}:\bar{\mathsf{v}}}}$$

$$\frac{\mu\vdash\mathsf{a}\Downarrow^{\mathsf{c},n}_{\tilde{\mathsf{p}}:\mathsf{o}:\bar{\mathsf{v}}}}{\mu\vdash(\!|\mathsf{a}|\!)\Downarrow^{\mathsf{c},n}_{\tilde{\mathsf{p}}:\mathsf{o}:\bar{\mathsf{v}}}} \qquad \frac{\mu\vdash\mathsf{a}\Downarrow^{\mathsf{c},n}_{\tilde{\mathsf{p}}:\mathsf{o}:\bar{\mathsf{v}}}}{\mu\vdash[\mathsf{a}\bullet\mathsf{a}_f\star\mathsf{a}_c\vartriangle\mathsf{a}_d]\Downarrow^{\mathsf{c},n}_{\tilde{\mathsf{p}}:\mathsf{o}:\bar{\mathsf{v}}}} \qquad \frac{\mu\vdash\mathsf{a}\Downarrow^{\mathsf{c},n}_{\tilde{\mathsf{p}}:\mathsf{o}:\bar{\mathsf{v}}} \ \vee\ \mathsf{s}\Downarrow^{\mathsf{c},n}_{\tilde{\mathsf{p}}:\mathsf{o}:\bar{\mathsf{v}}}}{\mu\vdash\mathsf{a}\,,\,\mathsf{s}\Downarrow^{\mathsf{c},n}_{\tilde{\mathsf{p}}:\mathsf{o}:\bar{\mathsf{v}}}}$$

Table 5: Matching rules / Is there an active receive along $\tilde{\mathsf{p}}$ and o matching $\bar{\mathsf{v}}$?

Rules (thr) and (term) report production of fault and forced termination signals, respectively. Auxiliary activities behave as expected: a message can always be delivered (rule (msg)) and the protected activity $(\!|\mathsf{a}|\!)$ behaves like a (rule (prot)). Rule (seq) takes care of activities executed sequentially, while rule (pick) permits to choose among alternative receive activities. Rules for conditional choice and iteration ((if) and (while), resp.) are standard. Execution of parallel activities is interleaved (rules (par_1) and (par_2)), except when a terminate/fault activity can be executed (rule (par_2)), in which case all parallel activities must immediately terminate except for short-lived activities and protected fault/compensation handlers. In other words, termination activities throw and exit are executed eagerly.

By rules (done_1) and (done_3), scope completions can be compensated according to the WS-BPEL *default* compensation behaviour (i.e. in the reverse order of completion) by the immediately enclosing scope. Notably, scopes like $[\mathsf{empty}\bullet\mathsf{a}_f\star\mathsf{a}_c\vartriangle\mathsf{a}_d]$ have not completed yet and when a scope completes, the default compensation a_d of inner scopes is not passed to the enclosing scope (rule (done_1)). Rule (exec) permits to perform any action of the primary activity a except for fault emission and scope completion. In particular, inner forced terminations are propagated externally outside the scope. Differently from forced termination, faults arising within a scope are managed internally (rule (fault)), and the corresponding handler is installed when the main activity completes (rule (done_2)). By rule (done_2), default compensation is performed *after* termination of the primary activity and before fault handling. Note that compensation activities do not store any state with them: hence, if the state changes between the compensation being stored and executed, the current state is used.

A few auxiliary functions are also used in the semantics of deployments defined in Table 6. The rules for communication and variables updating ((com), (new) and (var)) need a mechanism for checking if an assignment of some values $\bar{\mathsf{v}}$ to $\bar{\mathsf{w}}$ complies with the constraints imposed by the given correlation set c and state μ and, in case of success, returns a state μ' for the variables in $\bar{\mathsf{w}}$ that records the effect of the assignment. This mechanism is implemented by function $\mathsf{match}(\cdot, \cdot, \cdot, \cdot)$ defined through the rules in the upper part of Table 5. Notice that $\mathsf{match}(\cdot, \cdot, \cdot, \cdot)$ is undefined when $\bar{\mathsf{w}}$ and $\bar{\mathsf{v}}$ have different length or when $\mathsf{x} \in \mathsf{c}$ and $\{\mathsf{x} \mapsto \mathsf{v}'\} \in \mu$ for some $\mathsf{v}' \neq \mathsf{v}$ (since the state $\{\mathsf{x} \mapsto \mathsf{v}\}$ does not comply with c and μ). Rules (com) and (new) also use the auxiliary predicate $\mathsf{s} \Downarrow^{\mathsf{c},n}_{\tilde{\mathsf{p}}:\mathsf{o}:\bar{\mathsf{v}}}$, defined inductively on the syntax of s in the lower part of Table 5, that checks the ability of s of performing a receive on operation o exploiting the partner link $\tilde{\mathsf{p}}$, matching the tuple of values $\bar{\mathsf{v}}$ and generating a state with fewer pairs than n that complies with c and the current state of the activity performing the receive.

Finally, we linger on the rules in Table 6. By rule (com), communication can take place when two service instances perform matching receive and invoke activities complying with the correlation set of the receiving instance. Notice that matching covers both

$$\frac{\mathsf{a}_1 \xrightarrow{?t_1} \mathsf{a}_1' \quad \mathsf{a}_2 \xrightarrow{!t_2} \mathsf{a}_2' \quad \mathsf{match}(\mathsf{c}_1, \mu_1, t_1, t_2) = \mu_1' \quad \neg\,(\mu_1 \vdash \mathsf{a}_1\,,\, \mathsf{s}_1 \Downarrow_{t_2}^{\mathsf{c}_1, |\mu_1'|})}{\{\mu_1 \vdash \mathsf{a}_1\,,\, \mathsf{s}_1\}_{\mathsf{c}_1} \,\|\, \{\mu_2 \vdash \mathsf{a}_2\,,\, \mathsf{s}_2\}_{\mathsf{c}_2} \rightarrowtail \{\mu_1 \circ \mu_1' \vdash \mathsf{a}_1'\,,\, \mathsf{s}_1\}_{\mathsf{c}_1} \,\|\, \{\mu_2 \vdash \mathsf{a}_2'\,,\, \mathsf{s}_2\}_{\mathsf{c}_2}} \; (\mathsf{com})$$

$$\frac{[\mathsf{r} \bullet \mathsf{a}_f \star \mathsf{empty}] \xrightarrow{?t_1} \mathsf{a}_1 \quad \mathsf{a}_2 \xrightarrow{!t_2} \mathsf{a}_2' \quad \mathsf{match}(\mathsf{c}_1, \emptyset, t_1, t_2) = \mu_1 \quad \neg\,(\mathsf{s}_1 \Downarrow_{t_2}^{\mathsf{c}_1, |\mu_1|})}{\{[\mathsf{r} \bullet \mathsf{a}_f]\,,\, \mathsf{s}_1\}_{\mathsf{c}_1} \,\|\, \{\mu_2 \vdash \mathsf{a}_2\,,\, \mathsf{s}_2\}_{\mathsf{c}_2} \rightarrowtail \{\mu_1 \vdash \mathsf{a}_1\,,\, [\mathsf{r} \bullet \mathsf{a}_f]\,,\, \mathsf{s}_1\}_{\mathsf{c}_1} \,\|\, \{\mu_2 \vdash \mathsf{a}_2'\,,\, \mathsf{s}_2\}_{\mathsf{c}_2}} \; (\mathsf{new})$$

$$\frac{\mu \vdash \mathsf{a} \xrightarrow{\mathsf{x} \leftarrow \mathsf{v}} \mathsf{a}' \quad \mathsf{match}(\mathsf{c}, \mu, \mathsf{x}, \mathsf{v}) = \mu'}{\{\mu \vdash \mathsf{a}\,,\, \mathsf{s}\}_{\mathsf{c}} \rightarrowtail \{\mu \circ \mu' \vdash \mathsf{a}'\,,\, \mathsf{s}\}_{\mathsf{c}}} \; (\mathsf{var}) \qquad \frac{\mathsf{d}_1 \rightarrowtail \mathsf{d}_1'}{\mathsf{d}_1 \,\|\, \mathsf{d}_2 \rightarrowtail \mathsf{d}_1' \,\|\, \mathsf{d}_2} \; (\mathsf{part})$$

$$\frac{\mu \vdash \mathsf{a} \xrightarrow{\alpha} \mathsf{a}' \quad \alpha \notin \{?t_1, \,!t_2, \,\mathsf{x} \leftarrow \mathsf{v}\}}{\{\mu \vdash \mathsf{a}\,,\, \mathsf{s}\}_{\mathsf{c}} \rightarrowtail \{\mu \vdash \mathsf{a}'\,,\, \mathsf{s}\}_{\mathsf{c}}} \; (\mathsf{enab}) \qquad \frac{\mathsf{d} \equiv \mathsf{d}_1 \quad \mathsf{d}_1 \rightarrowtail \mathsf{d}_2 \quad \mathsf{d}_2 \equiv \mathsf{d}'}{\mathsf{d} \rightarrowtail \mathsf{d}'} \; (\mathsf{cong})$$

Table 6: Reduction rules for B*lite* deployments (where $t_1 = \ell^r : \mathsf{o} : \bar{\mathsf{x}}$ and $t_2 = \tilde{\mathsf{p}} : \mathsf{o} : \bar{\mathsf{v}}$)

partner link $\tilde{\mathsf{p}}$ and business data $\bar{\mathsf{v}}$. Communication generates a state that updates the state of the receiving instance. If more than one matching receive activity is able to process a given invoke, then only the most defined one (i.e. the receive that generates the 'smaller' state) progresses (predicate $\cdot \Downarrow_{\cdot}^{\cdot}$ serves this purpose). This mechanism permits to correlate messages to service instances and to model the precedence of an existing service instance over a new service instantiation (rule (new)), as it has been shown in Example 2.3 (*Multiple start and conflicting receive activities*) of Section 2.2. In rules (com) and (new), the assumption about *well-formedness* of deployments finds full employment, because it avoids to check every single deployment for possible conflicting receive activities. By rule (new), service instantiation can take place when a service definition and a service instance perform matching receive and invoke activities, respectively. By rule (var), correlation variables cannot be reassigned if the new value does not match with the old one. Moreover, if an assignment takes place, its effect is global to the instance, i.e. the state is updated.

By rule (enab), execution of activities different from communications or assignments can always proceed. If part of a larger deployment evolves, the whole composition evolves accordingly (rule (part)) and, as usual, structural congruent deployments have the same reductions (rule (cong)).

3.3. Examples

We report here the B*lite* specifications of the WS-BPEL processes graphically presented in the examples of Section 2.2.

Message correlation. The example of Figure 3(a) of the two uncorrelated receive activities corresponds to the following B*lite* deployed service definition:

$$\{\, [\, \mathsf{rcv}\,\langle \mathsf{p},\mathsf{q}\rangle\, \mathsf{LogOn}\,\langle \mathsf{x_{logID}}, \mathsf{x_{info}}\rangle \,;\, \mathsf{rcv}\,\langle \mathsf{r}\rangle\, \mathsf{RequestLogInfo}\,\langle\rangle \,; \\ \mathsf{inv}\,\langle \mathsf{q}\rangle\, \mathsf{SendLogInfo}\,\langle \mathsf{x_{logID}}, \mathsf{x_{info}}\rangle \,]\,\}_{\{\mathsf{x_{logID}}\}}$$

By correlating consecutive messages by means of the correlation variable $\mathsf{x_{logID}}$, as in Figure 3(b), we obtain the following modified deployment:

$$\{\, [\, \mathsf{rcv}\,\langle \mathsf{p},\mathsf{q}\rangle\, \mathsf{LogOn}\,\langle \mathsf{x_{logID}}, \mathsf{x_{info}}\rangle \,;\, \mathsf{rcv}\,\langle \mathsf{r}\rangle\, \mathsf{RequestLogInfo}\,\langle \mathsf{x_{logID}}\rangle \,; \\ \mathsf{inv}\,\langle \mathsf{q}\rangle\, \mathsf{SendLogInfo}\,\langle \mathsf{x_{logID}}, \mathsf{x_{info}}\rangle \,]\,\}_{\{\mathsf{x_{logID}}\}}$$

The special case when the two initial receives LogOn are identical (see Figure 3(c)) is expressed in B*lite* as follows:

$$\{\, [\, \mathsf{rcv}\,\langle \mathsf{p},\mathsf{q}\rangle\, \mathsf{LogOn}\,\langle \mathsf{x_{logID}}, \mathsf{x_{info_1}}\rangle \,;\, \mathsf{rcv}\,\langle \mathsf{p},\mathsf{q}\rangle\, \mathsf{LogOn}\,\langle \mathsf{x_{logID}}, \mathsf{x_{info_2}}\rangle \,; \\ \mathsf{rcv}\,\langle \mathsf{r}\rangle\, \mathsf{RequestLogInfo}\,\langle \mathsf{x_{logID}}\rangle \,;\, \mathsf{inv}\,\langle \mathsf{q}\rangle\, \mathsf{SendLogInfo}\,\langle \mathsf{x_{logID}}, \mathsf{x_{info_1}}, \mathsf{x_{info_2}}\rangle \,]\,\}_{\{\mathsf{x_{logID}}\}}$$

To illustrate, consider the following composition that also includes the deployment of two client process instances:

$$\{\, [\, \mathsf{rcv}\,\langle \mathsf{p},\mathsf{q}\rangle\, \mathsf{LogOn}\,\langle \mathsf{x_{logID}}, \mathsf{x_{info_1}}\rangle \,;\, \mathsf{rcv}\,\langle \mathsf{p},\mathsf{q}\rangle\, \mathsf{LogOn}\,\langle \mathsf{x_{logID}}, \mathsf{x_{info_2}}\rangle \,; \\ \mathsf{rcv}\,\langle \mathsf{r}\rangle\, \mathsf{RequestLogInfo}\,\langle \mathsf{x_{logID}}\rangle \,;\, \mathsf{inv}\,\langle \mathsf{q}\rangle\, \mathsf{SendLogInfo}\,\langle \mathsf{x_{logID}}, \mathsf{x_{info_1}}, \mathsf{x_{info_2}}\rangle \,]\,\}_{\{\mathsf{x_{logID}}\}} \\ \|\, \{\, \{\mathsf{x} \mapsto \mathsf{id}, \mathsf{y} \mapsto \mathsf{d_1}\} \vdash \mathsf{inv}\,\langle \mathsf{p},\mathsf{q}\rangle\, \mathsf{LogOn}\,\langle \mathsf{x},\mathsf{y}\rangle \,;\, \mathsf{inv}\,\langle \mathsf{r}\rangle\, \mathsf{RequestLogInfo}\,\langle \mathsf{x}\rangle \,; \\ \mathsf{rcv}\,\langle \mathsf{q}\rangle\, \mathsf{SendLogInfo}\,\langle \mathsf{x},\mathsf{z},\mathsf{k}\rangle \,\}_{\{\mathsf{x}\}} \\ \|\, \{\, \{\mathsf{x} \mapsto \mathsf{id}, \mathsf{y} \mapsto \mathsf{d_2}\} \vdash \mathsf{inv}\,\langle \mathsf{p},\mathsf{q}\rangle\, \mathsf{LogOn}\,\langle \mathsf{x},\mathsf{y}\rangle \,\}_{\{\mathsf{x}\}}$$

According to the semantics of B*lite*, the client processes trigger only one instantiation of the service. Thus, the only possible evolution, after four reduction steps, leads to

$$\begin{array}{l}\{\,[\,\mathsf{rcv}\,\langle \mathsf{p},\mathsf{q}\rangle\,\mathsf{LogOn}\,\langle \mathsf{x}_{\mathsf{logID}},\mathsf{x}_{\mathsf{info}_1}\rangle\,;\mathsf{rcv}\,\langle \mathsf{p},\mathsf{q}\rangle\,\mathsf{LogOn}\,\langle \mathsf{x}_{\mathsf{logID}},\mathsf{x}_{\mathsf{info}_2}\rangle\,;\\ \quad \mathsf{rcv}\,\langle \mathsf{r}\rangle\,\mathsf{RequestLogInfo}\,\langle \mathsf{x}_{\mathsf{logID}}\rangle\,;\mathsf{inv}\,\langle \mathsf{q}\rangle\,\mathsf{SendLogInfo}\,\langle \mathsf{x}_{\mathsf{logID}},\mathsf{x}_{\mathsf{info}_1},\mathsf{x}_{\mathsf{info}_2}\rangle\,]\ ,\\ \quad \{\,\mathsf{x}_{\mathsf{logID}}\mapsto \mathsf{id},\mathsf{x}_{\mathsf{data}_1}\mapsto \mathsf{d}_1,\mathsf{x}_{\mathsf{data}_2}\mapsto \mathsf{d}_2\,\}\vdash [\mathsf{empty}]\,\}_{\{\mathsf{x}_{\mathsf{logID}}\}}\end{array}$$

Asynchronous message delivering. To illustrate the example in Figure 5, consider the following B*lite* term:

$$\begin{array}{l}\{\,[\,\mathsf{rcv}\,\langle \mathsf{p},\mathsf{q}\rangle\,\mathsf{LogOn}\,\langle \mathsf{x}_{\mathsf{logID}},\mathsf{x}_{\mathsf{info}}\rangle\,;\mathsf{rcv}\,\langle \mathsf{r}\rangle\,\mathsf{RequestLogInfo}\,\langle \mathsf{x}_{\mathsf{logID}}\rangle\,;\\ \quad \mathsf{inv}\,\langle \mathsf{q}\rangle\,\mathsf{SendLogInfo}\,\langle \mathsf{x}_{\mathsf{logID}},\mathsf{x}_{\mathsf{info}}\rangle\,]\,\}_{\{\mathsf{x}_{\mathsf{logID}}\}}\\ \parallel \{\,\{\mathsf{x}\mapsto \mathsf{id},\mathsf{y}\mapsto \mathsf{d}_1\}\vdash \mathsf{inv}\,\langle \mathsf{r}\rangle\,\mathsf{RequestLogInfo}\,\langle \mathsf{x}\rangle\,;\\ \quad \mathsf{inv}\,\langle \mathsf{p},\mathsf{q}\rangle\,\mathsf{LogOn}\,\langle \mathsf{x},\mathsf{y}\rangle\,;\mathsf{rcv}\,\langle \mathsf{q}\rangle\,\mathsf{SendLogInfo}\,\langle \mathsf{x},\mathsf{z}\rangle\,\}_{\{\mathsf{x}\}}\end{array}$$

After message $\ll\langle \mathsf{r}\rangle : \mathsf{RequestLogInfo} : \langle \mathsf{id}\rangle \gg$ is produced by the first invoke activity, a service instance is created as a result of consumption of the message produced by the second invoke activity. That is, we have

$$\begin{array}{l}\{\,[\,\mathsf{rcv}\,\langle \mathsf{p},\mathsf{q}\rangle\,\mathsf{LogOn}\,\langle \mathsf{x}_{\mathsf{logID}},\mathsf{x}_{\mathsf{info}}\rangle\,;\mathsf{rcv}\,\langle \mathsf{r}\rangle\,\mathsf{RequestLogInfo}\,\langle \mathsf{x}_{\mathsf{logID}}\rangle\,;\\ \quad \mathsf{inv}\,\langle \mathsf{q}\rangle\,\mathsf{SendLogInfo}\,\langle \mathsf{x}_{\mathsf{logID}},\mathsf{x}_{\mathsf{info}}\rangle\,]\,\}_{\{\mathsf{x}_{\mathsf{logID}}\}}\\ \parallel \{\,\{\mathsf{x}\mapsto \mathsf{id},\mathsf{y}\mapsto \mathsf{d}_1\}\vdash \ll\langle \mathsf{r}\rangle:\mathsf{RequestLogInfo}:\langle \mathsf{id}\rangle\gg\ \mid\\ \quad \mathsf{inv}\,\langle \mathsf{p},\mathsf{q}\rangle\,\mathsf{LogOn}\,\langle \mathsf{x},\mathsf{y}\rangle\,;\mathsf{rcv}\,\langle \mathsf{q}\rangle\,\mathsf{SendLogInfo}\,\langle \mathsf{x},\mathsf{z}\rangle\,\}_{\{\mathsf{x}\}}\\ \rightarrowtail\ \rightarrowtail\\ \{\,[\,\mathsf{rcv}\,\langle \mathsf{p},\mathsf{q}\rangle\,\mathsf{LogOn}\,\langle \mathsf{x}_{\mathsf{logID}},\mathsf{x}_{\mathsf{info}}\rangle\,;\mathsf{rcv}\,\langle \mathsf{r}\rangle\,\mathsf{RequestLogInfo}\,\langle \mathsf{x}_{\mathsf{logID}}\rangle\,;\\ \quad \mathsf{inv}\,\langle \mathsf{q}\rangle\,\mathsf{SendLogInfo}\,\langle \mathsf{x}_{\mathsf{logID}},\mathsf{x}_{\mathsf{info}}\rangle\,]\ ,\\ \quad \{\,\mathsf{x}_{\mathsf{logID}}\mapsto \mathsf{id},\mathsf{x}_{\mathsf{info}}\mapsto \mathsf{d}\,\}\vdash [\,\mathsf{rcv}\,\langle \mathsf{r}\rangle\,\mathsf{RequestLogInfo}\,\langle \mathsf{x}_{\mathsf{logID}}\rangle\,;\\ \quad\quad \mathsf{inv}\,\langle \mathsf{q}\rangle\,\mathsf{SendLogInfo}\,\langle \mathsf{x}_{\mathsf{logID}},\mathsf{x}_{\mathsf{info}}\rangle\,]\,\}_{\{\mathsf{x}_{\mathsf{logID}}\}}\\ \parallel \{\,\{\mathsf{x}\mapsto \mathsf{id},\mathsf{y}\mapsto \mathsf{d}_1\}\vdash \ll\langle \mathsf{r}\rangle:\mathsf{RequestLogInfo}:\langle \mathsf{id}\rangle\gg\ \mid \mathsf{rcv}\,\langle \mathsf{q}\rangle\,\mathsf{SendLogInfo}\,\langle \mathsf{x},\mathsf{z}\rangle\,\}_{\{\mathsf{x}\}}\end{array}$$

Now, the first produced message can be consumed by the newly created service instance and we get

$$\begin{aligned}&\{\,[\,\mathsf{rcv}\,\langle p,q\rangle\,\mathsf{LogOn}\,\langle x_{\mathsf{logID}}, x_{\mathsf{info}}\rangle\,;\mathsf{rcv}\,\langle r\rangle\,\mathsf{RequestLogInfo}\,\langle x_{\mathsf{logID}}\rangle\,;\\&\quad \mathsf{inv}\,\langle q\rangle\,\mathsf{SendLogInfo}\,\langle x_{\mathsf{logID}}, x_{\mathsf{info}}\rangle\,]\;,\\&\quad \{\,x_{\mathsf{logID}} \mapsto \mathsf{id}, x_{\mathsf{info}} \mapsto \mathsf{d}\,\} \vdash [\,\mathsf{inv}\,\langle q\rangle\,\mathsf{SendLogInfo}\,\langle x_{\mathsf{logID}}, x_{\mathsf{info}}\rangle\,]\,\}_{\{x_{\mathsf{logID}}\}}\\&\;\|\,\{\,\{x \mapsto \mathsf{id}, y \mapsto \mathsf{d}_1\} \vdash \mathsf{rcv}\,\langle q\rangle\,\mathsf{SendLogInfo}\,\langle x, z\rangle\,\}_{\{x\}}\end{aligned}$$

that can further reduce to

$$\begin{aligned}&\{\,[\,\mathsf{rcv}\,\langle p,q\rangle\,\mathsf{LogOn}\,\langle x_{\mathsf{logID}}, x_{\mathsf{info}}\rangle\,;\mathsf{rcv}\,\langle r\rangle\,\mathsf{RequestLogInfo}\,\langle x_{\mathsf{logID}}\rangle\,;\\&\quad \mathsf{inv}\,\langle q\rangle\,\mathsf{SendLogInfo}\,\langle x_{\mathsf{logID}}, x_{\mathsf{info}}\rangle\,]\;,\;\{\,x_{\mathsf{logID}} \mapsto \mathsf{id}, x_{\mathsf{info}} \mapsto \mathsf{d}\,\} \vdash [\,\mathsf{empty}\,]\,\}_{\{x_{\mathsf{logID}}\}}\\&\;\|\,\{\,\{x \mapsto \mathsf{id}, y \mapsto \mathsf{d}_1, z \mapsto \mathsf{d}\} \vdash \mathsf{empty}\,\}_{\{x\}}\end{aligned}$$

Multiple start and conflicting receive activities. A deployment corresponding to the multiple start activities in Figure 6(a) is:

$$\begin{aligned}&\{\,[\;(\mathsf{rcv}\,\langle p_1,q\rangle\,\mathsf{LogOn}\,\langle x_{\mathsf{logID}}, x_{\mathsf{info}_1}\rangle\,|\,\mathsf{rcv}\,\langle p_2,q\rangle\,\mathsf{LogOn}\,\langle x_{\mathsf{logID}}, x_{\mathsf{info}_2}\rangle\,)\,;\\&\quad \mathsf{rcv}\,\langle r\rangle\,\mathsf{RequestLogInfo}\,\langle x_{\mathsf{logID}}\rangle\,;\mathsf{inv}\,\langle q\rangle\,\mathsf{SendLogInfo}\,\langle x_{\mathsf{logID}}, x_{\mathsf{info}_1}, x_{\mathsf{info}_2}\rangle\,]\,\}_{\{x_{\mathsf{logID}}\}}\end{aligned}$$

Now, consider the following composed deployment with two client processes:

$$\begin{aligned}&\{\,[\;(\mathsf{rcv}\,\langle p_1,q\rangle\,\mathsf{LogOn}\,\langle x_{\mathsf{logID}}, x_{\mathsf{info}_1}\rangle\,|\,\mathsf{rcv}\,\langle p_2,q\rangle\,\mathsf{LogOn}\,\langle x_{\mathsf{logID}}, x_{\mathsf{info}_2}\rangle\,)\,;\\&\quad \mathsf{rcv}\,\langle r\rangle\,\mathsf{RequestLogInfo}\,\langle x_{\mathsf{logID}}\rangle\,;\mathsf{inv}\,\langle q\rangle\,\mathsf{SendLogInfo}\,\langle x_{\mathsf{logID}}, x_{\mathsf{info}_1}, x_{\mathsf{info}_2}\rangle\,]\,\}_{\{x_{\mathsf{logID}}\}}\\&\;\|\,\{\,\{x \mapsto \mathsf{id}, y \mapsto \mathsf{d}_1\} \vdash \mathsf{inv}\,\langle p_1,q\rangle\,\mathsf{LogOn}\,\langle x,y\rangle\,;\mathsf{inv}\,\langle r\rangle\,\mathsf{RequestLogInfo}\,\langle x\rangle\,;\\&\quad \mathsf{rcv}\,\langle q\rangle\,\mathsf{SendLogInfo}\,\langle x,z,k\rangle\,\}\{x\}\\&\;\|\,\{\,\{x \mapsto \mathsf{id}, y \mapsto \mathsf{d}_2\} \vdash \mathsf{inv}\,\langle p_2,q\rangle\,\mathsf{LogOn}\,\langle x,y\rangle\,\}\{x\}\end{aligned}$$

After message $\ll\langle p_1\rangle:\mathsf{LogOn}:\langle \mathsf{id},\mathsf{d}_1\rangle\gg$, produced by invocation $\mathsf{inv}\,\langle p_1,q\rangle\,\mathsf{LogOn}\,\langle x,y\rangle$, has been processed by $\mathsf{rcv}\,\langle p_1,q\rangle\,\mathsf{LogOn}\,\langle x_{\mathsf{logID}}, x_{\mathsf{data}_1}\rangle$, the overall composition becomes

$$\begin{aligned}&\{\,[\;(\mathsf{rcv}\,\langle p_1,q\rangle\,\mathsf{LogOn}\,\langle x_{\mathsf{logID}}, x_{\mathsf{info}_1}\rangle\,|\,\mathsf{rcv}\,\langle p_2,q\rangle\,\mathsf{LogOn}\,\langle x_{\mathsf{logID}}, x_{\mathsf{info}_2}\rangle\,)\,;\\&\quad \mathsf{rcv}\,\langle r\rangle\,\mathsf{RequestLogInfo}\,\langle x_{\mathsf{logID}}\rangle\,;\mathsf{inv}\,\langle q\rangle\,\mathsf{SendLogInfo}\,\langle x_{\mathsf{logID}}, x_{\mathsf{info}_1}, x_{\mathsf{info}_2}\rangle\,,\\&\quad \{\,x_{\mathsf{logID}} \mapsto \mathsf{id}, x_{\mathsf{info}_1} \mapsto \mathsf{d}_1\,\} \vdash [\,\mathsf{rcv}\,\langle p_2,q\rangle\,\mathsf{LogOn}\,\langle x_{\mathsf{logID}}, x_{\mathsf{info}_2}\rangle\,;\\&\qquad \mathsf{rcv}\,\langle r\rangle\,\mathsf{RequestLogInfo}\,\langle x_{\mathsf{logID}}\rangle\,;\mathsf{inv}\,\langle q\rangle\,\mathsf{SendLogInfo}\,\langle x_{\mathsf{logID}}, x_{\mathsf{info}_1}, x_{\mathsf{info}_2}\rangle\,]\,\}_{\{x_{\mathsf{logID}}\}}\\&\;\|\,\{\,\{x \mapsto \mathsf{id}, y \mapsto \mathsf{d}_1\} \vdash \mathsf{inv}\,\langle r\rangle\,\mathsf{RequestLogInfo}\,\langle x\rangle\,;\mathsf{rcv}\,\langle q\rangle\,\mathsf{SendLogInfo}\,\langle x,z,k\rangle\,\}\{x\}\\&\;\|\,\{\,\{x \mapsto \mathsf{id}, y \mapsto \mathsf{d}_2\} \vdash \mathsf{inv}\,\langle p_2,q\rangle\,\mathsf{LogOn}\,\langle x,y\rangle\,\}\{x\}\end{aligned}$$

Now, the definition and the instance of the service compete for receiving the same message sent by the invoke activity $\mathsf{inv}\,\langle \mathsf{p}_2, \mathsf{q}\rangle\, \mathsf{LogOn}\,\langle \mathsf{x}, \mathsf{y}\rangle$. In cases like this, the WS-BPEL specification requires that the invocation is only delivered to the existing instance, which prevents creation of a new instance. In fact, in B*lite* the above term can only reduce to

$$\begin{array}{l}
\{\, [\ (\mathsf{rcv}\,\langle \mathsf{p}_1, \mathsf{q}\rangle\, \mathsf{LogOn}\,\langle \mathsf{x}_{\mathsf{logID}}, \mathsf{x}_{\mathsf{info}_1}\rangle \,|\, \mathsf{rcv}\,\langle \mathsf{p}_2, \mathsf{q}\rangle\, \mathsf{LogOn}\,\langle \mathsf{x}_{\mathsf{logID}}, \mathsf{x}_{\mathsf{info}_2}\rangle\,)\,; \\
\quad \mathsf{rcv}\,\langle \mathsf{r}\rangle\, \mathsf{RequestLogInfo}\,\langle \mathsf{x}_{\mathsf{logID}}\rangle\,; \mathsf{inv}\,\langle \mathsf{q}\rangle\, \mathsf{SendLogInfo}\,\langle \mathsf{x}_{\mathsf{logID}}, \mathsf{x}_{\mathsf{info}_1}, \mathsf{x}_{\mathsf{info}_2}\rangle\,, \\
\quad \{\, \mathsf{x}_{\mathsf{logID}} \mapsto \mathsf{id}, \mathsf{x}_{\mathsf{info}_1} \mapsto \mathsf{d}_1, \mathsf{x}_{\mathsf{info}_2} \mapsto \mathsf{d}_2 \,\} \vdash [\, \mathsf{rcv}\,\langle \mathsf{r}\rangle\, \mathsf{RequestLogInfo}\,\langle \mathsf{x}_{\mathsf{logID}}\rangle\,; \\
\qquad \mathsf{inv}\,\langle \mathsf{q}\rangle\, \mathsf{SendLogInfo}\,\langle \mathsf{x}_{\mathsf{logID}}, \mathsf{x}_{\mathsf{info}_1}, \mathsf{x}_{\mathsf{info}_2}\rangle\,]\,\}_{\{\mathsf{x}_{\mathsf{logID}}\}} \\
\|\, \{\, \{\mathsf{x} \mapsto \mathsf{id}, \mathsf{y} \mapsto \mathsf{d}_1\} \vdash \mathsf{inv}\,\langle \mathsf{r}\rangle\, \mathsf{RequestLogInfo}\,\langle \mathsf{x}\rangle\,; \mathsf{rcv}\,\langle \mathsf{q}\rangle\, \mathsf{SendLogInfo}\,\langle \mathsf{x}, \mathsf{z}, \mathsf{k}\rangle\,\}\{\mathsf{x}\} \\
\|\, \{\, \{\mathsf{x} \mapsto \mathsf{id}, \mathsf{y} \mapsto \mathsf{d}_2\} \vdash \mathsf{empty}\,\}\{\mathsf{x}\}
\end{array}$$

Scheduling of parallel activities. The example of Figure 8(a) in B*lite* can be rendered by the following term:

$$\mathsf{x}_1 := \mathsf{v}_1 \mid \mathsf{x}_2 := \mathsf{v}_2 \mid \mathsf{x}_3 := \mathsf{v}_3$$

The semantics of B*lite* prescribes that the three assignments can be executed in an unpredictable order that may change in different executions.

Forced termination. The example of Figure 9(a) in B*lite* can be rendered by the following term:

$$\texttt{<exit>} \mid \mathsf{x}_1 := \mathsf{v}_1\, ;\, \mathsf{x}_2 := \mathsf{v}_2$$

while that of Figure 9(b) can be rendered by the following term:

$$\texttt{<throw>} \mid \mathsf{x}_1 := \mathsf{v}_1\, ;\, \mathsf{x}_2 := \mathsf{v}_2$$

Eager execution of activities causing termination. The B*lite* terms are the same as the example before. To illustrate that B*lite* activities throw and exit have higher priority than the remaining ones, consider the following structured activity:

$$\mathsf{a} \triangleq \mathsf{throw} \mid \mathsf{sh}_1\ ;\ \mathsf{sh}_2 \mid \mathsf{rcv}\,\langle \mathsf{p}\rangle\, \mathsf{o}\,\langle \mathsf{x}\rangle\ ;\ \mathsf{a}'$$

In B*lite*, by executing activity throw, this term can only reduce to:

$$\mathsf{stop} \mid \mathsf{end}(\mathsf{sh}_1\ ;\ \mathsf{sh}_2) \mid \mathsf{end}(\mathsf{rcv}\,\langle \mathsf{p}\rangle\,\mathsf{o}\,\langle \mathsf{x}\rangle\ ;\ \mathsf{a}') \ \equiv\ \mathsf{stop} \mid \mathsf{sh}_1$$

While ActiveBPEL agrees with this requirement, Oracle BPEL and Apache ODE do not implement any prioritized behavior for termination activities. Thus, for example, the above term a can evolve by firstly performing the short-lived activity sh_1 and then activity throw; this way, activity sh_2 is not terminated.

Handlers protection. The following B*lite* term corresponds to the example illustrated in Figure 11:

$$\begin{aligned}\mathsf{a}\ \triangleq\ & [\,(\,[\,[\,\mathsf{x} := \mathsf{b}_1 \bullet \mathsf{throw} \star \mathsf{x} := \mathsf{b}_2\,]\,;\mathsf{throw} \bullet \mathsf{throw} \star \mathsf{empty}\,]\\ & \quad \mid \mathsf{if(x)\{throw\}\{empty\}}\,) \bullet \mathsf{empty}\ \,]\end{aligned}$$

Now, consider a deployment containing a service instance $\emptyset \vdash \mathsf{a}$. A possible computation is the following one

$$\begin{aligned}
\{\emptyset \vdash \mathsf{a}\}_\emptyset \ \xrightarrow{(1)}\ & \{\{\mathsf{x} \mapsto \mathsf{b}_1\} \vdash [\,(\,[\,[\,\mathsf{empty} \bullet \mathsf{throw} \star \mathsf{x} := \mathsf{b}_2\,]\,;\mathsf{throw} \bullet \mathsf{throw} \star \mathsf{empty}\,]\\
& \quad \mid \mathsf{if(x)\{throw\}\{empty\}}\,) \bullet \mathsf{empty}\ \,]\}_\emptyset\\
\xrightarrow{(2)}\ & \{\{\mathsf{x} \mapsto \mathsf{b}_1\} \vdash [\,(\,[\,\mathsf{empty}\,;\mathsf{throw} \bullet \mathsf{throw} \star \mathsf{empty} \vartriangle (\mathsf{x} := \mathsf{b}_2\,;\mathsf{empty})]\\
& \quad \mid \mathsf{if(x)\{throw\}\{empty\}}\,) \bullet \mathsf{empty}\ \,]\}_\emptyset\\
\xrightarrow{(3)}\ & \{\{\mathsf{x} \mapsto \mathsf{b}_1\} \vdash [\,(\,[\,\mathsf{throw} \bullet \mathsf{throw} \star \mathsf{empty} \vartriangle (\mathsf{x} := \mathsf{b}_2\,;\mathsf{empty})]\\
& \quad \mid \mathsf{if(x)\{throw\}\{empty\}}\,) \bullet \mathsf{empty}\ \,]\}_\emptyset\\
\xrightarrow{(4)}\ & \{\{\mathsf{x} \mapsto \mathsf{b}_1\} \vdash [\,(\,[\,\mathsf{stop} \bullet \mathsf{throw} \star \mathsf{empty} \vartriangle (\mathsf{x} := \mathsf{b}_2\,;\mathsf{empty})]\\
& \quad \mid \mathsf{if(x)\{throw\}\{empty\}}\,) \bullet \mathsf{empty}\ \,]\}_\emptyset\\
\xrightarrow{(5)}\ & \{\{\mathsf{x} \mapsto \mathsf{b}_1\} \vdash [\,(\,(\!|\mathsf{x} := \mathsf{b}_2\,;\mathsf{empty}|\!) \mid \mathsf{if(x)\{throw\}\{empty\}}\,) \bullet \mathsf{empty}\ \,]\}_\emptyset\\
\xrightarrow{(6)}\ & \{\{\mathsf{x} \mapsto \mathsf{b}_1\} \vdash [\,(\,(\!|\mathsf{x} := \mathsf{b}_2\,;\mathsf{empty}|\!) \mid \mathsf{throw}\,) \bullet \mathsf{empty}\ \,]\}_\emptyset\\
\xrightarrow{(7)}\ & \{\{\mathsf{x} \mapsto \mathsf{b}_1\} \vdash [\,(\,\mathsf{end}((\!|\mathsf{x} := \mathsf{b}_2\,;\mathsf{empty}|\!)) \mid \mathsf{stop}\,) \bullet \mathsf{empty}\ \,]\}_\emptyset\\
\equiv\ & \{\{\mathsf{x} \mapsto \mathsf{b}_1\} \vdash [\,(\,(\!|\mathsf{x} := \mathsf{b}_2\,;\mathsf{empty}|\!) \mid \mathsf{stop}\,) \bullet \mathsf{empty}\ \,]\}_\emptyset\\
\xrightarrow{(8)}\ & \{\{\mathsf{x} \mapsto \mathsf{b}_2\} \vdash [\,(\,(\!|\mathsf{empty}|\!) \mid \mathsf{stop}\,) \bullet \mathsf{empty}\ \,]\}_\emptyset
\end{aligned}$$

where the reductions are labelled by numbers indicating the corresponding steps. When the scope of the first assign activity completes, the compensation handler (i.e. the second assign activity) is inserted into the default compensation activities of its enclosing scope (1-2). When execution of the next throw activity rises a fault, then the fault is caught by the corresponding fault handler (3-4) that activates the default compensation $\mathsf{a}_c\,;\mathsf{throw}$. This activity is protected, by using the auxiliary operator $\llparenthesis\cdot\rrparenthesis$, from the effect of the forced termination triggered by the parallel scope throw activity (5-8).

4. B*lite*C: a tool for rapid development of WS-BPEL applications

In this section, we present B*lite*C[4], a software tool supporting the development of WS-BPEL applications by automatically translating B*lite* specifications into executable WS-BPEL programs. B*lite*C is developed in Java[5] 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 [7] for creating and managing XML documents, JavaCC [6] for generating the parsers that validate the input documents, and JJTree[6] 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 12. The tool is composed of five main components:

- *Mapper* parses the declarative part (see Section 4.1) of the input B*lite* program and

[4]B*lite*C is a free software; it can be downloaded from `http://rap.dsi.unifi.it/blite` and re-distributed and/or modified under the terms of the GNU General Public License as published by the Free Software Foundation.

[5]JRE and JDK version 6.

[6]JJTree is included with JavaCC.

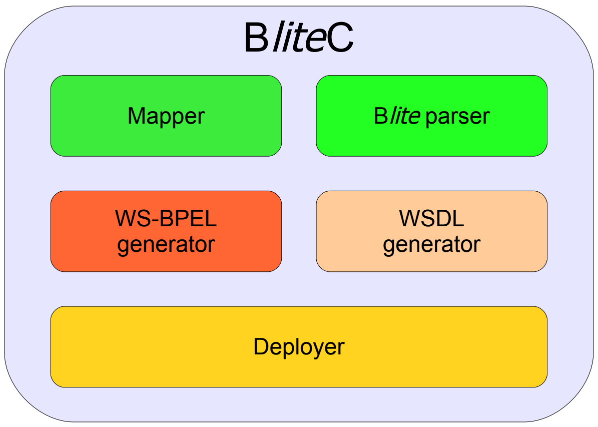


Figure 12: B*lite*C architecture

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.

In the rest of the section, we present the syntax of B*lite* accepted by the tool and explain the correspondence between the B*lite* constructs and the WS-BPEL activities. An example application of B*lite*C will be illustrated in the next section.

4.1. Specifying service orchestrations in BliteC

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.

The syntax of B*lite* accepted by B*lite*C is given in Table 7. With respect to the syntax reported in Table 2, for the sake of practicality, sequence and parallel composition are defined as n-ary operators. Moreover, expressions are explicitly defined as combination of values and variables by means of boolean, arithmetic, comparision and string operators, where 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). B*lite* specifications are finite compositions of *definitions* (that assign names to B*lite* terms), containing at most one *deployment* definition. Notably, the symbol $\triangleq$ used in Section 3 is replaced by the symbol `:=`.

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 ]];
  }
```

b ::=		(basic activities)
	inv pl op <x_1,...,x_n> | rcv pl op <x_1,...,x_n>	(invoke, receive)
|	x := e | empty | throw | exit	(assign, empty, throw, exit)
pl ::=	<partner> | <partner$_1$,partner$_2$>	(partner links)
e ::=		(expressions)
	e_1 | e_2 | e_1 & e_2 | ! e | TRUE | FALSE	(boolean operators)
|	$e_1 + e_2$ | $e_1 - e_2$ | $e_1 * e_2$ | e_1 / e_2 | int	(arithmetic operators)
|	$e_1 >= e_2$ | $e_1 <= e_2$ | $e_1 > e_2$ | $e_1 < e_2$	(comparison operators)
|	$e_1 = e_2$ | e_1 != e_2	
|	e_1 . e_2 | "string"	(string operators)
|	x | pl | $literal_name	(variable, partner link, literal)
a ::=		(structured activities)
	b | if (e) {a_1} {a_2} | while (e) {a}	(basic, conditional, iteration)
|	seq a_1 ; ... ; a_n qes | flw a_1 | ... | a_n wlf	(sequence, parallel)
|	[A @ A_f * A_c]	(scope)
|	pck rcv pl_1 op$_1$ <x_1,...,x_k> ; a_1	(pick)
	+ ... + rcv pl_n op$_n$ <x_1,...,x_h> ; a_n kcp	
r ::=		(start activities)
	rcv pl op <x_1,...,x_n> | seq r; a_1 ; ... ; a_n qes	(receive, sequence)
|	flw r_1 | ... | r_n wlf | [R @ A_f * A_c]	(parallel, scope)
|	pck rcv pl_1 op$_1$ <x_1,...,x_k> ; a_1	(pick)
	+ ... + rcv pl_n op$_n$ <x_1,...,x_h> ; a_n kcp	
s ::=	[R @ A_f] 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	(definitions)
|	i := d;;	

Table 7: Syntax of B*lite* accepted by B*lite*C

```
    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_shipped => xsd:integer;`). Messages, that are tuples of variables used as either sending source or receiving target, can be declared in two ways:

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

- by generating a new message type, e.g. in the following declaration

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

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

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

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

and, then, can be assigned to a B*lite* variable (in the specification part) by using the associated name, e.g. `x_weat := $reqLit;`.

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

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

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

4.2. From Blite to WS-BPEL

We provide here some insights about the transformation of B*lite* constructs into WS-BPEL activities. Since our tests point out that the same WS-BPEL program might have different behaviours on different engines, 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.

Communication activities, invokes and receives, are translated in a different way depending on their arguments and their position in the code. For example, as shown in Table 8, 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

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" />`

Table 8: Mapping of the receive activity

`<receive>` activity. The correlation attributes `initiate` and `pattern` are specified according to the type of the interaction.

The invoke activity is translated similarly, as shown in Table 9; 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 10, is straightforward. In particular, an assign activity involving message variables is translated by exploiting the type of such variables (defined in the declarative part) to identify the involved parts. Also the translation of the structured activities does not require significant effort, as shown in Table 11. Finally, in Table 12 a B*lite* service is rendered as a scope, where the compensation handler is removed and the tag `<scope>` is replaced by `<process>`.

We conclude this section by showing a sample application of B*lite*C to one of the examples introduced in Section 2.2 and formalized as a B*lite* term in Section 3.3.

The B*lite*C program informally illustrated in Figure 3(b) is as follows:

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 9: Mapping of the invoke activity

```
s_example1 ::= [ seq
                  rcv <p,q> LogOn <x_logID,x_info>;
                  rcv <r> RequestLogInfo <x_logID>;
                  inv <q> SendLogInfo <x_logID,x_info>
                 qes ];;

example1 ::= {s_example1}{x_logID};;

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

  VARIABLES {
    <x_logID,x_info> => gen:msg1,<id,info>,<xsd:int,xsd:string>;
    <x_logID> => gen:msg2,<id>,<xsd:int>;
  }
?>
```

The corresponding WS-BPEL program generated by B*lite*C, where irrelevant details have been omitted, is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<process name="example1Process" ... >
  <import location="example1.wsdl" ... />
  <partnerLinks>
    <partnerLink name="pPL"
```

B*lite*	WS-BPEL

`x :=` *e*	`<assign>`
	  `<copy>`
	    `<from>` *e* `</from>`
	    `<to> $var_x.part_x </to>`
	  `</copy>`
	`</assign>`

`empty`	`<empty />`
`throw`	`<throw />`
`exit`	`<exit />`

Table 10: Mapping of assign, empty, throw and exit activities

```
              partnerLinkType="mwl:pPLT"
              myRole="p"
              partnerRole="q"
              initializePartnerRole="no" />
  <partnerLink name="rPL"
              partnerLinkType="mwl:rPLT"
              myRole="r" />
</partnerLinks>
<variables>
  <variable name="var0" messageType="mwl:msg1" />
  <variable name="var1" messageType="mwl:msg2" />
  <variable name="var2" messageType="mwl:clientAddressP" />
</variables>
<correlationSets>
  <correlationSet name="x_logIDCorr"
                 properties="mwl:x_logIDProp" />
</correlationSets>
<faultHandlers>
  <catchAll>
    <sequence>
      <compensate /><empty />
    </sequence>
  </catchAll>
</faultHandlers>
<sequence>
  <receive partnerLink="pPL"
          operation="LogOn"
          variable="var0"
          createInstance="yes">
    <correlations>
      <correlation set="x_logIDCorr" initiate="yes" />
    </correlations>
```

B*lite*	WS-BPEL
if (e) {a_1} {a_2}	<if> <condition> e </condition> a_1 <else> a_2 </else> </if>
while (e) {a}	<while> <condition> e </condition> a </while>
seq a_1 ; ... ; a_n qes	<sequence> a_1 ... a_n </sequence>
flw a_1 | ... | a_n wlf	<flow> a_1 ... a_n </flow>
pck a_1 + ... + a_n kcp	<pick> a_1 ... a_n </pick>
[a @ a_f * a_c]	<scope> <faultHandlers> <catchAll> <sequence> <compensate/> a_f </sequence> </catchAll> </faultHandlers> <compensationHandler> a_c </compensationHandler> a </scope>

Table 11: Mapping of structured activities

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

Table 12: Mapping of service definitions

```
  </receive>
  <sequence>
    <receive partnerLink="pPL"
             operation="setClientAddress"
             variable="var2">
      <correlations>
        <correlation set="x_logIDCorr" initiate="no" />
      </correlations>
    </receive>
    <assign>
      <copy>
        <from variable="var2" part="address" />
        <to partnerLink="pPL" />
      </copy>
    </assign>
  </sequence>
  <receive partnerLink="rPL"
           operation="RequestLogInfo"
           variable="var1">
    <correlations>
      <correlation set="x_logIDCorr" initiate="no" />
    </correlations>
  </receive>
  <invoke operation="SendLogInfo"
          inputVariable="var0"
          partnerLink="pPL">
    <correlations>
      <correlation set="x_logIDCorr"
                   pattern="request"
                   initiate="no" />
    </correlations>
  </invoke>
</sequence>
```

```
</process>
```

As expected, the activities of the B*lite*C program have been translated into the corresponding WS-BPEL ones. Notably, to enable asynchronous communication, the translation automatically puts an additional receive activity 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 such partner link with the symmetric invoke activity.

The generation of WSDL documents and file descriptors used to deploy WS-BPEL programs on ActiveBPEL will be shown in the next section.

5. B*lite*C at work

We show an application of B*lite*C to a scenario built upon the shipping service drawn from the WS-BPEL specification document and already introduced in Section 2.1.

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

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

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

Its behaviour is very simple: the process gets instantiated by the shipping service by invoking the operation `o_num`; then, the created instance waits for an integer number (representing the quantity of available items) provided by a human actor along the operation `o_humanInteraction` and concludes by sending the number back to the shipping service.

The fact that the invoke activity used for the reply is performed along the same operation of the initial receive indicates that the two activities form a synchronous request-response interaction, hence the invoke will be translated into a `<reply>` activity. The order identifier, stored in `x_id`, is used as a correlation value.

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

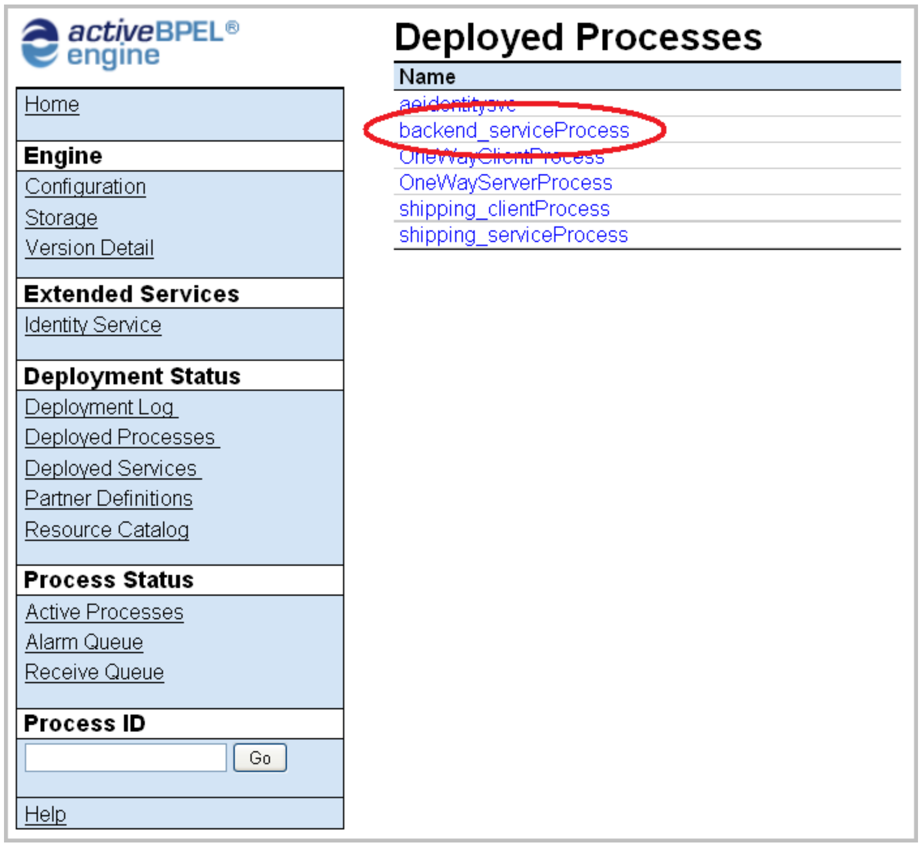
```
<?blm
 ADDRESSES {
   myns => "http://example/backendService";
   myaddress => "http://XXX:8080/active-bpel/services";
 }

 VARIABLES {
         <x_id> => gen:id,<id>,<xsd:int>;
   <x_id,x_num> => gen:human,<id,num>,<xsd:int,xsd:int>;
   <x_num,x_id> => gen:number,<num,id>,<xsd:int,xsd:int>;
 }
?>
```

To compile this B*lite* program, we have to save the above code into a file (named, e.g., `backend_service.bl`) and execute the following command:

```
java -jar blite.jar backend_service.bl
```

This way, the file `backend_serviceProcess.bpr`, which is a WS-BPEL package directly deployable into ActiveBPEL, is generated (the included WS-BPEL and WSDL files are reported in the Appendix). To deploy the file, it is sufficient to move it into the engine's deployment directory `bpr`. Then, to check that the deploy succeeded, we can use the ActiveBPEL's administration console that can be accessed by using any browser at the address `http://XXX:8080/BpelAdmin` (where `XXX` is the server's address where the ActiveBPEL engine is running). By selecting `Deployed Processes` from the menu on the left-hand side, we obtain the list of the deployed processes (Figure 13) among which



Deployed Processes

Name
aeidentitysvc
backend_serviceProcess
OneWayClientProcess
OneWayServerProcess
shipping_clientProcess
shipping_serviceProcess

Figure 13: List of deployed processes

`backend_serviceProcess` must appear. Now, by selecting `Deployed services`, we can retrieve the URLs of the two WSDL files corresponding to the partner links for interacting with the service:

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

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

Finally, by using a tool for automatic generation of web service requests (e.g. soapUI [8]), we can invoke the service by sending the following SOAP messages:

```
<soapenv:Envelope ... >
   <soapenv:Header/>
   <soapenv:Body>
```

```
      <bac:x_idEL> 1234 </bac:x_idEL>
   </soapenv:Body>
</soapenv:Envelope>

<soapenv:Envelope ... >
   <soapenv:Header/>
   <soapenv:Body>
      <bac:x_idEL> 1234 </bac:x_idEL>
      <bac:x_numEL> 7 </bac:x_numEL>
   </soapenv:Body>
</soapenv:Envelope>
```

The first message creates an instance for the order identified by `1234`, while the second message indicates that seven items for that order are available for shipping. After the first message is sent, by selecting `Active Processes` from the console menu, we can verify that a back-end service instance has been created and its status is `Running`. Then, after the other message is sent, we get in response the pair of integers `<7,1234>` and the instance status changes to `Completed`.

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

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

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

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

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

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

 IMPORTS {
   bck => "http://example/backendService/
                  backend_service.wsdl";
 }

 VARIABLES {
   <x_id,x_c,x_items> => gen:shipOrder,
                        <id, shipComplete, items>,
                        <xsd:int, xsd:int, xsd:int>;
       <x_id,x_items> => gen:shippingNoticeMsg,
                        <id, items>,
                        <xsd:int,xsd:int>;
       <x_id,x_count> => gen:shippingNoticeMsg;
       <x_id,x_sorry> => gen:shippingErrorMsg,
                        <id, errorMsg>,
                        <xsd:int,xsd:string>;
               <x_id> => bck:id;
       <x_count,x_id> => bck:number;
            x_shipped => xsd:integer;
 }

 PARTNERLINKS {
    PARTNERLINK {
        TYPE => bck:clientPLT;
        PARTNER_ROLE backend => bck:p_backendPT;
    }
 }
```

```
?>
```

Here, differently from the specification in Section 2.1, after the invocation of the back-end service, the returned number (stored in `x_count`) is checked: a number less than or equal to `0` means that the required item is out of stock and, hence, a fault is raised by means of the `throw` activity. The fault will be caught and handled by the fault handler `a_err`, that will send an error message to the client.

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

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

 shipping_client ::= {s_shipClient}{y_id};;

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

  IMPORTS {
     shs => "http://example/shipping_service.wsdl";
  }

  VARIABLES {
    <y_id, y_c, y_items> => shs:shipOrder;
           <y_id,y_sorry> => shs:shippingErrorMsg;
           <y_id,y_count> => shs:shippingNoticeMsg;
                y_shipped => xsd:integer;
            <y_id,y_resp> => gen:response,
                             <id,resp>,
                             <xsd:int,xsd:string>;
  }

  PARTNERLINKS {
     PARTNERLINK {
         TYPE => shs:custPLT;
         MY_ROLE cust => shs:x_custPT;
         PARTNER_ROLE ship_srv => shs:p_shipPT;
     }
  }
 ?>
```

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

Since the shipping service requires configuration data provided by the back-end service and, similarly, the client needs data from the shipping service, to successfully compile the above B*lite* programs (whose corresponding WS-BPEL and WSDL files are reported in the Appendix), 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
```

6. Concluding remarks

In this paper we have discussed the problem of equipping WS-BPEL with a precise semantics for supporting a rapid and easy development of WS-BPEL applications.

As a first contribution, we have tested and compared, by means of several illustrative examples, the behaviour of three of the most known freely available WS-BPEL engines. The results of our experiments demonstrate that the many loose points in the WS-BPEL specification document have led to engines implementing different semantics and, hence, have undermined portability of WS-BPEL programs over different platforms.

As a second contribution, we have introduced B*lite*, a prototypical orchestration language inspired to WS-BPEL but with a simpler syntax and a well-defined operational semantics. B*lite*'s formal semantics can help making some loose aspects of the WS-BPEL specification more precise and can be exploited to drive implementations of future WS-BPEL engines. Our study can also contribute to the many discussions on compensation and correlation which have been reported by the WS-BPEL Technical Committee [35] (see, e.g., discussions related to issues 66, 207 and 271).

Several precise semantics of WS-BPEL were proposed in the literature (for an overview see [38]). Many of these efforts aim at formalizing a *complete* semantics for WS-BPEL using Petri nets [38, 30], but do not cover such dynamical aspects as service instantiation and message correlation. Other works [21, 24] using process calculi focus instead on small and relatively simple subsets of WS-BPEL. Another bunch of related works [25, 32] formalize the semantics of WS-BPEL by encoding parts of the language into more foundational orchestration languages. Our work differs for the number of features that are simultaneously modelled and for the fact that dynamical aspects are fully taken into account. Recently, a very general and flexible framework for error recovery has been introduced in [23]; this framework extends [24] with dynamic compensation, modelling in particular the dependency between fault handling and the request-response communication pattern.

Some other relevant related works are [15, 14, 12]. In the first two, the authors propose a formal approach to model compensation in transactional calculi and present a detailed comparison with [16]. The third is an extension of asynchronous π-calculus with long-running (scoped) transactions. The language has a scope construct which plays a role similar to the scope activity presented in our semantics, but it is not aimed at capturing the order in which compensations should be activated. On the contrary, the semantics we propose 'faithfully' capture the intended semantics of WS-BPEL, thus for example compensations are activated in the reverse order w.r.t. the order of completion of the original scopes.

As a third contribution, we have developed B*lite*C, a software tool aiming at solving some well-known programming problems of WS-BPEL caused by its XML syntax, lack of a precise semantics, and non-standardization of the deployment procedure. B*lite*C takes as an input programs written in B*lite* and provides as an output the corresponding WS-BPEL processes deployable on the engine ActiveBPEL.

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 [37], Intalio—Designer [5], ActiveVOS Designer [3], and Eclipse BPEL designer [4]. 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 [1] and Eclipse [2]. [31] 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). [34] proposes a different approach to develop SOC applications that still relies on a formal language. However, input programs are directly executed within a purposely developed engine, rather than being translated into and deployed as WS-BPEL processes.

Currently, 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 two other freely available engines, namely Oracle BPEL Process Manager and Apache ODE [10]. Of course, to preserve the semantics of the original B*lite* programs, one has to study the inner implementation of every supported engine and to define a customized translation. Unfortunately, since no engine has a formal description of its behaviour, this study has to be carried out by means of experimental tests and, most of all, no formal proof of semantics preservation can be done. We also plan to integrate in B*lite*C a better support for data manipulation based on XPath.

Our long term programme is to provide a framework for the design and verification of WS-BPEL applications that supports analysis of service orchestration. We believe that our approach can enable tailoring proof techniques and analytical tools typical of process calculi to the needs of WS-BPEL applications. Indeed, on the one hand, alike other technological standards, WS-BPEL does not provide support for sound engineering methodologies to application development and analysis. On the other hand, it has been shown that type systems, model checking and (bi)simulation analysis provide adequate tools to address topics relevant to the web services technology [33, 40]. In the end, this 'proof technology' can pave the way for the development of (semi-)automatic property validation tools (as e.g. in [20]).

As a step in this direction, in [26] we have defined an encoding from B*lite* to COWS [27], a recently proposed calculus for orchestration of web services, and we have formalized the properties enjoyed by the encoding. By relying on these results, we plan to devise methods to analyze B*lite* specifications (and the corresponding WS-BPEL applications) by exploiting the analytical tools already developed for COWS, such as the type system introduced in [28] to check confidentiality properties, the stochastic extension defined in [39] to enable quantitative reasoning on service behaviours, the static analysis introduced in [11] to establish properties of the flow of information between services, and the logical verification environment presented in [20] to express and check functional properties of services.

Acknowledgments. We thank Luca Cesari for having contributed with his master thesis to the development of the software tool B*lite*C.

References

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

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

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

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

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

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

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

[8] soapUI 2.5.1, February 2009. `http://www.soapui.org`.

[9] Active Endpoints. ActiveBPEL 5.0.2, May 2008. `http://www.active-endpoints.com`.

[10] Apache Software Foundation. Apache ODE 1.2, July 2008. Available at `http://ode.apache.org`.

[11] J. Bauer, F. Nielson, H. Nielson, and H. Pilegaard. Relational analysis of correlation. In M. Alpuente and G. Vidal, editors, *The 15th International Static Analysis Symposium (SAS'08)*, volume 5079 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2008.

[12] L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long-running transactions. In *Proceeding of FMOODS*, volume 2884 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2003.

[13] A. Brown, S. Johnston, and K. Kelly. Using service-oriented architecture and component-based development to build web service applications. Technical report, Rational Software Corp., 2002.

[14] R. Bruni, M. Butler, C. Ferreira, C. Hoare, H. Melgratti, and U. Montanari. Comparing two approaches to compensable flow composition. In *Proceeding of CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 383–397. Springer, 2005.

[15] R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *Proceeding of POPL*, pages 209–220. ACM, 2005.

[16] M. Butler and C. Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In *Proceeding of COORDINATION*, volume 2949 of *Lecture Notes in Computer Science*, pages 87–104. Springer, 2004.

[17] L. Cesari, A. Lapadula, R. Pugliese, and F. Tiezzi. A tool for rapid development of WS-BPEL applications. In *SAC*. ACM, 2010. To appear.

[18] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. Technical report, W3C, 2001.
Available at `http://www.w3.org/TR/wsdl/`.

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

[20] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A model checking approach for verifying COWS specifications. In *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 2008.

[21] P. Geguang, Z. Xiangpeng, W. Shuling, and Q. Zongyan. Semantics of BPEL4WS-like fault and compensation handling. In *Proceeding of FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 350–365. Springer, 2005.

[22] M. Gudgin, M. Hadley, and T. Rogers. Web Services Addressing 1.0 - Core. Technical report, W3C, 2006.
Available at `http://www.w3.org/TR/ws-addr-core`.

[23] C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro. On the interplay between fault handling and request-response service invocations. In J. Billington, Z. Duan, and M. Koutny, editors, *Proc. of Int. Conf. on Application of Concurrency to System Design (ACSD'08)*, pages pages 190–199. IEEE CS Press, 2008.

[24] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: a calculus for service oriented computing. In *Proceeding of ICSOC*, volume 4294 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2006.

[25] C. Laneve and G. Zavattaro. Foundations of web transactions. In *Proceeding of FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2005.

[26] A. Lapadula. *A Formal Account of Web Services Orchestration*. PhD Thesis in Computer Science, Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, 2008.
Available at `http://rap.dsi.unifi.it/cows`.

[27] A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2007.

[28] A. Lapadula, R. Pugliese, and F. Tiezzi. Regulating data exchange in service oriented applications. In *FSEN*, volume 4767 of *Lecture Notes in Computer Science*, pages 223–239. Springer, 2007.

[29] A. Lapadula, R. Pugliese, and F. Tiezzi. A formal account of WS-BPEL. In *COORDINATION*, volume 5052 of *Lecture Notes in Computer Science*, pages 199–215. Springer, 2008.

[30] N. Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0. In *Proceeding of WSFM*, volume 4937 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2008.

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

[32] M. Mazzara and R. Lucchi. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):96–118, 2006.

[33] L. Meredith and S. Bjorg. Contracts and types. *Communication of the ACM*, 46(10):41–47, 2003.

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

[35] OASIS WSBPEL TC. WS-BPEL issues list.
Available at `http://www.oasis-open.org/committees/download.php/20228/WS_BPEL_issues_list.html`.

[36] OASIS WSBPEL TC. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, April 2007.
Available at `http://docs.oasis-open.org/wsbpel/2.0/OS/`.

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

[38] C. Ouyang, W. van der Aalst, S. Breutel, M. Dumas, A. ter Hofstede, and H. Verbeek. Formal semantics and analysis of control flow in WS-BPEL (revised version). Technical report, BPM Center Report, 2005.
Available at `http://www.BPMcenter.org`.

[39] D. Prandi and P. Quaglia. Stochastic COWS. In *ICSOC*, volume 4749 of *Lecture Notes in Computer Science*, pages 245–256. Springer, 2007.

[40] F. van Breugel and M. Koshkina. Models and verification of BPEL. Available at `http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf`, 2006.

APPENDIX

WS-BPEL and WSDL files generated by BliteC

We report here the WS-BPEL and WSDL files generated by B*lite*C for the shipping service scenario.

Back-end service (WS-BPEL file)

```
<?xml version="1.0" encoding="UTF-8"?>
<!--This file is generated by Blite - Original
    Checksum:eecf51f7b7042467042123053c43c5f3-->
<process xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
        xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:mwl="http://example/backendService/backend_service.wsdl"
        suppressJoinFailure="yes"
        name="backend_serviceProcess"
        targetNamespace="http://example/backendService/backend_service.bpel">
  <import location="backend_service.wsdl"
         namespace="http://example/backendService/backend_service.wsdl"
         importType="http://schemas.xmlsoap.org/wsdl/"/>
  <partnerLinks>
    <partnerLink name="p_humanPL"
                 partnerLinkType="mwl:p_humanPLT"
                 myRole="p_human" />
    <partnerLink name="clientPL"
                 partnerLinkType="mwl:clientPLT"
                 myRole="p_backend" />
  </partnerLinks>
  <variables>
    <variable name="var2" messageType="mwl:number" />
    <variable name="var1" messageType="mwl:human" />
    <variable name="var0" messageType="mwl:id" />
  </variables>
  <correlationSets>
    <correlationSet name="x_idCorr" properties="mwl:x_idProp" />
  </correlationSets>
  <faultHandlers>
    <catchAll>
      <sequence>
        <compensate /><empty />
      </sequence>
    </catchAll>
  </faultHandlers>
  <sequence>
    <receive partnerLink="clientPL"
```

```
            operation="o_num"
            variable="var0"
            createInstance="yes">
      <correlations>
        <correlation set="x_idCorr" initiate="yes" />
      </correlations>
    </receive>
    <sequence>
      <receive partnerLink="p_humanPL"
              operation="o_humanInteraction"
              variable="var1">
        <correlations>
          <correlation set="x_idCorr" initiate="no" />
        </correlations>
      </receive>
      <assign>
        <copy>
          <from variable="var1" part="num" />
          <to variable="var2" part="num" />
        </copy>
      </assign>
    </sequence>
    <reply operation="o_num"
          partnerLink="clientPL"
          variable="var2">
      <correlations>
        <correlation set="x_idCorr" initiate="no" />
      </correlations>
    </reply>
  </sequence>
</process>
```

Back-end service (WSDL file)

```
<?xml version="1.0" encoding="UTF-8"?>
<!--This file is generated by Blite - Original
     Checksum:bb68ade28c12cf759b2cff007f360974-->
<wsdl:definitions
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
     xmlns:tns="http://example/backendService/backend_service.wsdl"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
     xmlns:prop="http://docs.oasis-open.org/wsbpel/2.0/varprop"
     targetNamespace="http://example/backendService/backend_service.wsdl">
  <wsdl:types>
    <xsd:schema
        targetNamespace="http://example/backendService/backend_service.wsdl">
     <xsd:element name="x_idEL" type="xsd:int" />
     <xsd:element name="x_numEL" type="xsd:int" />
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="id">
    <wsdl:part name="id" element="tns:x_idEL" />
  </wsdl:message>
  <wsdl:message name="human">
    <wsdl:part name="id" element="tns:x_idEL" />
    <wsdl:part name="num" element="tns:x_numEL" />
  </wsdl:message>
  <wsdl:message name="number">
    <wsdl:part name="num" element="tns:x_numEL" />
    <wsdl:part name="id" element="tns:x_idEL" />
  </wsdl:message>
  <wsdl:portType name="p_backendPT">
    <wsdl:operation name="o_num">
     <wsdl:input message="tns:id" />
     <wsdl:output message="tns:number" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:portType name="p_humanPT">
    <wsdl:operation name="o_humanInteraction">
     <wsdl:input message="tns:human" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="p_backendBinding" type="tns:p_backendPT">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="o_num">
     <soap:operation
          soapAction="http://example/backendService/backend_service.wsdl/o_num"
          style="document" />
     <wsdl:input>
       <soap:body use="literal" />
     </wsdl:input>
     <wsdl:output>
       <soap:body use="literal" />
     </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:binding name="p_humanBinding"
               type="tns:p_humanPT">
    <soap:binding style="document"
     transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="o_humanInteraction">
     <soap:operation
          soapAction="http://example/backendService/backend_service.wsdl/o_humanInteraction"
          style="document" />
     <wsdl:input>
       <soap:body use="literal" />
       </wsdl:input>
     </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="p_backendService">
     <wsdl:port name="p_backendPort" binding="tns:p_backendBinding">
       <soap:address
            location="http://XXX:8080/active-bpel/services/p_backendService" />
     </wsdl:port>
    </wsdl:service>
    <wsdl:service name="p_humanService">
     <wsdl:port name="p_humanPort" binding="tns:p_humanBinding">
       <soap:address
            location="http://XXX:8080/active-bpel/services/p_humanService" />
     </wsdl:port>
    </wsdl:service>
    <plnk:partnerLinkType name="clientPLT">
     <plnk:role name="p_backend" portType="tns:p_backendPT" />
    </plnk:partnerLinkType>
    <plnk:partnerLinkType name="p_humanPLT">
     <plnk:role name="p_human" portType="tns:p_humanPT" />
    </plnk:partnerLinkType>
    <prop:property name="x_idProp" type="xsd:int" />
    <prop:propertyAlias propertyName="tns:x_idProp" messageType="tns:number" part="id" />
    <prop:propertyAlias propertyName="tns:x_idProp" messageType="tns:human" part="id" />
    <prop:propertyAlias propertyName="tns:x_idProp" messageType="tns:id" part="id" />
  </wsdl:definitions>
```

Shipping service (WS-BPEL file)

```
<?xml version="1.0" encoding="UTF-8"?>
<!--This file is generated by Blite - Original
   Checksum:8700cc88bc91b3c79506530830303 0ff-->
<process xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
        xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:bck="http://example/backendService/backend_service.wsdl"
        xmlns:mwl="http://example/shipping_service.wsdl"
        suppressJoinFailure="yes"
        name="shipping_serviceProcess"
        targetNamespace="http://example/shipping_service.bpel">
  <import location="shipping_service.wsdl"
         namespace="http://example/shipping_service.wsdl"
         importType="http://schemas.xmlsoap.org/wsdl/" />
  <import location="backend_service.wsdl"
         namespace="http://example/backendService/backend_service.wsdl"
         importType="http://schemas.xmlsoap.org/wsdl/" />
  <partnerLinks>
    <partnerLink name="backend"
```

```
              partnerLinkType="bck:clientPLT"
              partnerRole="p_backend" />
  <partnerLink name="custPL"
              partnerLinkType="mwl:custPLT"
              myRole="p_ship"
              partnerRole="x_cust"
              initializePartnerRole="no" />
</partnerLinks>
<variables>
  <variable name="var0" messageType="mwl:shipOrder" />
  <variable name="var3" messageType="mwl:shippingErrorMsg" />
  <variable name="var7" messageType="mwl:clientAddressP_SHIP" />
  <variable name="var5" messageType="bck:number" />
  <variable name="var4" messageType="bck:id" />
  <variable name="var1" messageType="mwl:shippingNoticeMsg" />
  <variable name="var2" messageType="mwl:shippingNoticeMsg" />
  <variable name="var6" type="xsd:integer" />
</variables>
<correlationSets>
  <correlationSet name="x_idCorr" properties="mwl:x_idProp" />
</correlationSets>
<faultHandlers>
  <catchAll>
    <sequence>
      <compensate /><empty />
    </sequence>
  </catchAll>
</faultHandlers>
<sequence>
  <sequence>
    <receive partnerLink="custPL"
            operation="o_req"
            variable="var0"
            createInstance="yes">
      <correlations>
        <correlation set="x_idCorr" initiate="yes" />
      </correlations>
    </receive>
    <assign>
      <copy>
        <from variable="var0" part="items" />
        <to variable="var1" part="items" />
      </copy>
    </assign>
  </sequence>
  <sequence>
    <receive partnerLink="custPL"
            operation="setClientAddress"
            variable="var7">
      <correlations>
      <correlation set="x_idCorr" initiate="no" />
    </correlations>
  </receive>
  <assign>
    <copy>
      <from variable="var7" part="address" />
      <to partnerLink="custPL" />
    </copy>
  </assign>
</sequence>
<if>
  <condition>$var0.shipComplete &gt; 0</condition>
  <invoke operation="o_notice"
          inputVariable="var1"
          partnerLink="custPL">
    <correlations>
      <correlation set="x_idCorr"
                   pattern="request"
                   initiate="no" />
    </correlations>
  </invoke>
  <else>
    <scope>
      <faultHandlers>
        <catchAll>
          <sequence>
            <compensate/>
            <sequence>
              <assign>
                <copy>
                  <from>'Sorry, the required item
                   is out of stock '</from>
                  <to>$var3.errorMsg</to>
                </copy>
              </assign>
              <invoke operation="o_err"
                      inputVariable="var3"
                      partnerLink="custPL">
                <correlations>
                  <correlation set="x_idCorr"
                               pattern="request"
                               initiate="no" />
                </correlations>
              </invoke>
            </sequence>
          </sequence>
        </catchAll>
      </faultHandlers>
      <compensationHandler>
        <empty />
```

```
</compensationHandler>
<sequence>
  <assign>
    <copy>
      <from>0</from>
      <to variable="var6" />
    </copy>
  </assign>
  <while>
    <condition>
       $var6 &lt; $var0.items
    </condition>
    <sequence>
      <sequence>
        <invoke operation="o_num"
                inputVariable="var4"
                partnerLink="backend"
                outputVariable="var5">
          <correlations>
            <correlation set="x_idCorr"
                 pattern="request-response"
                 initiate="no" />
          </correlations>
        </invoke>
        <assign>
          <copy>
            <from variable="var5" part="num" />
            <to variable="var2" part="items" />
          </copy>
        </assign>
      </sequence>
      <if>
        <condition>
           $var2.items &lt;= 0
        </condition>
        <throw xmlns:err="http://example/error"
          faultName="err:
              shipping_serviceError"/>
        <else>
          <sequence>
            <invoke operation="o_notice"
                    inputVariable="var2"
                    partnerLink="custPL">
              <correlations>
                <correlation set="x_idCorr"
                             pattern="request"
                             initiate="no" />
              </correlations>
            </invoke>
            <assign>
```

```
                    <copy>
                     <from>$var6 + $var2.items</from>
                     <to variable="var6" />
                    </copy>
                  </assign>
                </sequence>
              </else>
            </if>
          </sequence>
        </while>
      </sequence>
    </scope>
  </else>
  </if>
 </sequence>
</process>
```

Shipping service (WSDL file)

```
<?xml version="1.0" encoding="UTF-8"?>
<!--This file is generated by Blite - Original
   Checksum:49db04e980ed5b3a93862d9a82529b5f-->
<wsdl:definitions
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
     xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
     xmlns:bck="http://example/backendService/backend_service.wsdl"
     xmlns:tns="http://example/shipping_service.wsdl"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
     xmlns:prop="http://docs.oasis-open.org/wsbpel/2.0/varprop"
     targetNamespace="http://example/shipping_service.wsdl">
  <wsdl:import namespace="http://example/backendService/backend_service.wsdl"
             location="backend_service.wsdl" />
  <wsdl:types>
    <xsd:schema targetNamespace="http://example/shipping_service.wsdl">
      <xsd:import
          namespace="http://schemas.xmlsoap.org/ws/2003/03/addressing"
          schemaLocation="../schema/ws-addressing.xsd" />
      <xsd:element name="x_countEL" type="xsd:int" />
      <xsd:element name="x_idEL" type="xsd:int" />
      <xsd:element name="x_itemsEL" type="xsd:int" />
      <xsd:element name="x_sorryEL" type="xsd:string" />
      <xsd:element name="x_cEL" type="xsd:int" />
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="shippingNoticeMsg">
  <wsdl:part name="id" element="tns:x_idEL" />
  <wsdl:part name="items" element="tns:x_itemsEL" />
</wsdl:message>
<wsdl:message name="clientAddressP_SHIP">
  <wsdl:part name="corrID" element="tns:x_idEL" />
  <wsdl:part name="address" element="wsa:EndpointReference" />
</wsdl:message>
<wsdl:message name="shippingErrorMsg">
  <wsdl:part name="id" element="tns:x_idEL" />
  <wsdl:part name="errorMsg" element="tns:x_sorryEL" />
</wsdl:message>
<wsdl:message name="shipOrder">
  <wsdl:part name="id" element="tns:x_idEL" />
  <wsdl:part name="shipComplete" element="tns:x_cEL" />
  <wsdl:part name="items" element="tns:x_itemsEL" />
</wsdl:message>
<wsdl:portType name="x_custPT">
  <wsdl:operation name="o_notice">
    <wsdl:input message="tns:shippingNoticeMsg" />
  </wsdl:operation>
  <wsdl:operation name="o_err">
    <wsdl:input message="tns:shippingErrorMsg" />
  </wsdl:operation>
</wsdl:portType>
<wsdl:portType name="p_shipPT">
  <wsdl:operation name="o_req">
    <wsdl:input message="tns:shipOrder" />
  </wsdl:operation>
  <wsdl:operation name="setClientAddress">
    <wsdl:input message="tns:clientAddressP_SHIP" />
  </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="p_shipBinding" type="tns:p_shipPT">
  <soap:binding style="document"
             transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="o_req">
    <soap:operation
         soapAction="http://example/shipping_service.wsdl/o_req"
         style="document" />
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
  </wsdl:operation>
  <wsdl:operation name="setClientAddress">
    <soap:operation
         soapAction="http://example/shipping_service.wsdl/setClientAddress"
         style="document" />
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="p_shipService">
    <wsdl:port name="p_shipPort" binding="tns:p_shipBinding">
      <soap:address
            location="http://XXX:8080/active-bpel/services/p_shipService" />
    </wsdl:port>
  </wsdl:service>
  <plnk:partnerLinkType name="custPLT">
    <plnk:role name="p_ship" portType="tns:p_shipPT" />
    <plnk:role name="x_cust" portType="tns:x_custPT" />
  </plnk:partnerLinkType>
  <prop:property name="x_idProp" type="xsd:int" />
  <prop:propertyAlias propertyName="tns:x_idProp"
        messageType="tns:clientAddressP_SHIP"
        part="corrID" />
  <prop:propertyAlias propertyName="tns:x_idProp"
        messageType="bck:number" part="id" />
  <prop:propertyAlias propertyName="tns:x_idProp"
        messageType="bck:id" part="id" />
  <prop:propertyAlias propertyName="tns:x_idProp"
        messageType="tns:shippingErrorMsg" part="id" />
  <prop:propertyAlias propertyName="tns:x_idProp"
        messageType="tns:shippingNoticeMsg" part="id" />
  <prop:propertyAlias propertyName="tns:x_idProp"
        messageType="tns:shippingNoticeMsg" part="id" />
  <prop:propertyAlias propertyName="tns:x_idProp"
        messageType="tns:shipOrder" part="id" />
</wsdl:definitions>
```

Shipping client (WS-BPEL file)

```
<?xml version="1.0" encoding="UTF-8"?>
<!--This file is generated by Blite - Original
   Checksum:d181ce2939dcdbe8a7208a7d4445a98a-->
<process xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
        xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:mwl="http://example/shipping_client.wsdl"
        xmlns:shs="http://example/shipping_service.wsdl"
        suppressJoinFailure="yes"
        name="shipping_clientProcess"
        targetNamespace="http://example/shipping_client.bpel">
  <import location="shipping_service.wsdl"
          namespace="http://example/shipping_service.wsdl"
          importType="http://schemas.xmlsoap.org/wsdl/"/>
  <import location="shipping_client.wsdl"
          namespace="http://example/shipping_client.wsdl"
        importType="http://schemas.xmlsoap.org/wsdl/"/>
<partnerLinks>
  <partnerLink name="cltPL"
              partnerLinkType="mwl:cltPLT"
              myRole="p_init" />
  <partnerLink name="srv"
              partnerLinkType="shs:custPLT"
              myRole="x_cust"
              partnerRole="p_ship" />
</partnerLinks>
<variables>
  <variable name="var1" messageType="shs:shippingErrorMsg" />
  <variable name="var3" type="xsd:integer" />
  <variable name="var4" messageType="mwl:response" />
  <variable name="var0" messageType="shs:shipOrder" />
  <variable name="var5" messageType="shs:clientAddressP_SHIP" />
  <variable name="var2" messageType="shs:shippingNoticeMsg" />
</variables>
<correlationSets>
  <correlationSet name="y_idCorr" properties="mwl:y_idProp" />
</correlationSets>
<faultHandlers>
  <catchAll>
    <sequence>
      <compensate /><empty />
    </sequence>
  </catchAll>
</faultHandlers>
<sequence>
  <receive partnerLink="cltPL"
          operation="o_init"
          variable="var0"
          createInstance="yes">
    <correlations>
      <correlation set="y_idCorr" initiate="yes" />
    </correlations>
  </receive>
  <assign>
    <copy>
      <from>concat('ORDER: ', string($var0.id), ' BEGIN ')</from>
      <to>$var4.resp</to>
    </copy>
  </assign>
  <invoke operation="o_req"
         inputVariable="var0"
         partnerLink="srv">
    <correlations>
      <correlation set="y_idCorr"
                  pattern="request"
                  initiate="no" />
  </correlations>
</invoke>
<sequence>
  <assign>
    <copy>
      <from>
        <literal>
          <wsa:EndpointReference
               xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
               xmlns:dyn="http://example/shipping_service.wsdl">
            <wsa:Address>
               http://XXX:8080/active-bpel/services/x_custService
            </wsa:Address>
            <wsa:PortType>dyn:x_custPT</wsa:PortType>
            <wsa:ServiceName PortName="x_custServicePort">
                    dyn:x_custService
            </wsa:ServiceName>
          </wsa:EndpointReference>
        </literal>
      </from>
      <to>$var5.address</to>
    </copy>
  </assign>
  <invoke operation="setClientAddress"
          inputVariable="var5"
          partnerLink="srv">
    <correlations>
      <correlation set="y_idCorr"
                   pattern="request"
                   initiate="no" />
    </correlations>
  </invoke>
</sequence>
<assign>
  <copy>
    <from>0</from>
    <to variable="var3" />
  </copy>
</assign>
<while>
  <condition>$var3 &lt; $var0.items</condition>
  <pick>
    <onMessage partnerLink="srv"
               operation="o_notice"
               variable="var2">
      <correlations>
        <correlation set="y_idCorr" initiate="no" />
      </correlations>
      <sequence>
        <assign>
        <copy>
          <from>$var3 + $var2.items</from>
          <to variable="var3" />
        </copy>
      </assign>
      <assign>
        <copy>
          <from>concat(string($var4.resp),
                       'NOTICE: sent items=',
                       string($var2.items),'; ')
          </from>
          <to>$var4.resp</to>
        </copy>
      </assign>
    </sequence>
  </onMessage>
  <onMessage partnerLink="srv"
             operation="o_err"
             variable="var1">
    <correlations>
      <correlation set="y_idCorr" initiate="no" />
    </correlations>
    <sequence>
      <assign>
        <copy>
          <from>$var0.items</from>
          <to variable="var3" />
        </copy>
      </assign>
      <assign>
        <copy>
          <from>concat(string($var4.resp),
                       'ERROR: ',
                       string($var1.errorMsg))
          </from>
          <to>$var4.resp</to>
        </copy>
      </assign>
    </sequence>
  </onMessage>
</pick>
</while>
<assign>
  <copy>
    <from>concat(string($var4.resp),' END')</from>
    <to>$var4.resp</to>
  </copy>
</assign>
<reply operation="o_init"
       partnerLink="cltPL"
```

```
          variable="var4">
      <correlations>
        <correlation set="y_idCorr" initiate="no" />
      </correlations>
    </reply>
  </sequence>
</process>
```

Shipping client (WSDL file)

```
<?xml version="1.0" encoding="UTF-8"?>
<!--This file is generated by Blite - Original
     Checksum:a36628ae5bab1af4addba03185416741-->
<wsdl:definitions
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:shs="http://example/shipping_service.wsdl"
      xmlns:tns="http://example/shipping_client.wsdl"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
      xmlns:prop="http://docs.oasis-open.org/wsbpel/2.0/varprop"
      targetNamespace="http://example/shipping_client.wsdl">
  <wsdl:import
        namespace="http://example/shipping_service.wsdl"
        location="shipping_service.wsdl" />
  <wsdl:types>
    <xsd:schema targetNamespace="http://example/shipping_client.wsdl">
      <xsd:element name="y_respEL" type="xsd:string" />
      <xsd:element name="y_idEL" type="xsd:int" />
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="response">
    <wsdl:part name="id" element="tns:y_idEL" />
    <wsdl:part name="resp" element="tns:y_respEL" />
  </wsdl:message>
  <wsdl:portType name="p_initPT">
    <wsdl:operation name="o_init">
      <wsdl:input message="shs:shipOrder" />
      <wsdl:output message="tns:response" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="x_custBinding" type="shs:x_custPT">
    <soap:binding style="document"
              transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="o_notice">
      <soap:operation
            soapAction="http://example/shipping_service.wsdl/o_notice"
            style="document" />
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
  </wsdl:operation>
  <wsdl:operation name="o_err">
    <soap:operation
         soapAction="http://example/shipping_service.wsdl/o_err"
         style="document" />
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
  </wsdl:operation>
</wsdl:binding>
<wsdl:binding name="p_initBinding" type="tns:p_initPT">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="o_init">
    <soap:operation
         soapAction="http://example/shipping_client.wsdl/o_init"
         style="document" />
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:service name="x_custService">
  <wsdl:port name="x_custPort" binding="tns:x_custBinding">
    <soap:address
         location="http://XXX:8080/active-bpel/services/x_custService" />
  </wsdl:port>
</wsdl:service>
<wsdl:service name="p_initService">
  <wsdl:port name="p_initPort" binding="tns:p_initBinding">
    <soap:address
         location="http://XXX:8080/active-bpel/services/p_initService" />
  </wsdl:port>
</wsdl:service>
<plnk:partnerLinkType name="cltPLT">
  <plnk:role name="p_init" portType="tns:p_initPT" />
</plnk:partnerLinkType>
<prop:property name="y_idProp" type="xsd:int" />
<prop:propertyAlias propertyName="tns:y_idProp"
      messageType="shs:clientAddressP_SHIP"
      part="corrID" />
<prop:propertyAlias propertyName="tns:y_idProp"
      messageType="tns:response" part="id" />
<prop:propertyAlias propertyName="tns:y_idProp"
```

```
        messageType="shs:shippingNoticeMsg" part="id" />
  <prop:propertyAlias propertyName="tns:y_idProp"
        messageType="shs:shippingErrorMsg" part="id" />
  <prop:propertyAlias propertyName="tns:y_idProp"
        messageType="shs:shipOrder" part="id" />
</wsdl:definitions>
```