

# Using formal methods to develop WS-BPEL applications<sup>☆</sup>

Alessandro Lapadula<sup>a</sup>, Rosario Pugliese<sup>a,\*</sup>, Francesco Tiezzi<sup>a</sup>

<sup>a</sup>*Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze,  
Viale Morgagni 65, I-50134, Firenze, Italy*

---

## Abstract

In recent years, WS-BPEL has become a *de facto* 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 *Blite*, a prototypical orchestration language equipped with a formal operational semantics, which is closely inspired by, but simpler than, WS-BPEL. Indeed, *Blite* 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 *BliteC*, a software tool supporting a rapid and easy development of WS-BPEL applications via translation of service orchestrations written in *Blite* into executable WS-BPEL programs. We illustrate our approach by means of a running example borrowed from the official specification of WS-BPEL.

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

---

---

<sup>☆</sup>This work is partially based on two preliminary papers appeared in [40, 22] 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)

## 1. Introduction

Information systems are by now at the very foundations of our society, mainly due to 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 the automation of 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), a paradigm for distributed and e-business computing that 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 loosely coupled, reusable software components named *services*.

*Web Services* (WSs) are currently one of the most successful and well-developed implementations of the SOC general paradigm. WSs make available the functionalities that a company wants to expose over the World Wide Web, so that they can be discovered and exploited both by human clients and other services. A key factor for their success is the exploitation of the Web architecture, that is nowadays an 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 allow proprietary interfaces and data formats to be replaced by a standard Web-messaging infrastructure based on XML technologies, thus facilitating automated integration of newly built and legacy applications, both within and across organization boundaries. For instance, the W3C standard WSDL (*Web Services Description Language*, [23]) permits to express WSs public interfaces, i.e. the functionalities they offer and require. These interfaces can then be exploited by client applications to determine the location of a remote WS and the operations it implements, as well as how to invoke each operation.

The above standard technologies are usually sufficient for simple applications integration needs. On the other hand, creation of complex B2B applications and automated integration of business processes across organizations 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, WS-BPEL (*Web Services Business Process Execution Language*, [49]) has become a *de facto* standard as a language for programming business processes, i.e. software entities capable of orchestrating available WSs by invoking them according to given sets of rules to meet business requirements. Notably, business processes may themselves be exposed as services, making service orchestration a recursive operation that permits to build complex services out of simpler ones.

However, designing and developing WS-BPEL applications is a difficult and error-prone task. The language has an XML syntax which makes writing WS-BPEL code awkward 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 might turn out to be cumbersome and ineffective when programming more complex applications. Further difficulties derive from the fact that WS-BPEL is equipped with such intricate features as concurrency, multiple service instances, message correlation, long-running business transactions, fault and compensation handlers. Most of all, WS-BPEL comes without a formal semantics and its specification document [49], written in ‘natural’ language, contains a fair number of acknowledged loose points that may give rise to different interpretations and lead to different implementation choices. Some of these loose points are due to an extensive use 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 [49, Section 10.4]) certainly cannot help the implementers, which can be led to implement very different semantics. Similarly, it seems somewhat inappropriate the choice of deeming 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 [49, Section 1]) and the handling of an incoming request message that no process instance is able to receive (see [49, Section 9.2]). Finally, some sentences are ambiguous and sometimes conflicting, which might produce misinterpretations (see also Section 2.3). For example, the relationship between WS-BPEL (multiple) *start activities* and the mechanisms handling race conditions is not fully clarified; moreover, subtle behaviour 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.

In this paper, we firstly point out major loose points of the WS-BPEL specification by means of many examples focussing on key topics 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 [4], Apache ODE [11] and Oracle BPEL Process Manager [3]. Our tests show that some WS-BPEL processes produce a very different behaviour when executed by different engines. As a matter of fact, the engines implement different semantics, which considerably undermines the portability of WS-BPEL programs across different platforms.

Portability is indeed of particular relevance in the SOC setting, however it is further compromised since the deployment procedure of WS-BPEL programs is not standardised. In fact, to execute a WS-BPEL program, besides the associated WSDL 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 SOC applications, to state and prove their properties, and to direct attention towards issues that might otherwise be overlooked. Therefore, we define *Blite*, a ‘lightweight’ orchestration language closely inspired by, but simpler than, WS-BPEL. *Blite* is the result of the tension between handiness and expressiveness which is typical when designing a formalism. Thus, to keep the semantics of the language rigorous but still manageable, the design of *Blite* focuses on the ‘procedural’ part of WS-BPEL and only retains those features of WS-BPEL that, in our opinion, are absolutely necessary to formally define the basic elements of service orchestrations and to characterise service engines. On the contrary, the set of WS-BPEL constructs is not intended to be a minimal one and indeed the language supports both programming styles of its two ‘official’ forerunners, the Microsoft’s block-structured language XLANG [55] and the IBM’s graph-oriented language WSFL (Web Services Flow Language, [42]).

We believe that using *Blite* as a language for orchestrating services offers many significant advantages. On the one hand, the *Blite* textual notation is certainly more manageable and user-friendly than those, possibly graphical, notations proposed for WS-BPEL, especially when programming larger applications. Besides, graphical design notations may also be a source of problems when they are not backed up by a rigorous semantics [54]. On the other hand, *Blite* is equipped with a formal operational semantics that precisely states the effect of execution of its constructs. Moreover, since *Blite* constructs directly correspond to those of a meaningful sublanguage of WS-BPEL, as the translation we present in the paper demonstrates, we can confidently state that *Blite*’s formal semantics provides this WS-BPEL’s sublanguage with a rigorous semantics. Some of the most intricate and complex features of WS-BPEL, as e.g. the interplay between compensation activities and the control flow of the originating process, or the relationship between the mechanisms for service instance creation and identification, are thus made clearer.

The translation of *Blite* into WS-BPEL is also the key for developing a framework for the execution of *Blite* programs. Actually, although a prototypical *Blite* engine following the dictates of *Blite*’s operational semantics is under implementation [51], this task is arduous and time-consuming. To speed up the experimentation with *Blite* and its assessment, we are exploiting ActiveBPEL that, according to our tests, is one of the freely available WS-BPEL engines that better complies with the WS-BPEL specification. We have indeed developed *BliteC*, a software tool that accepts as an input a *Blite* program and returns

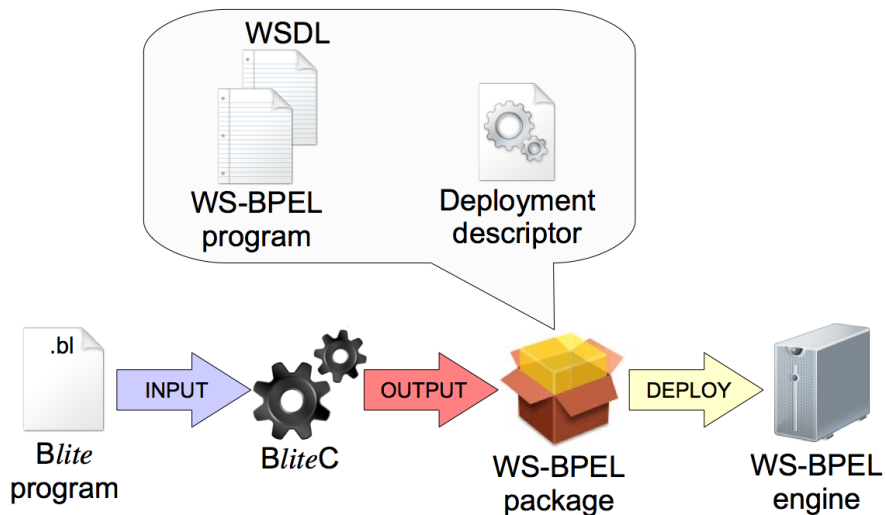


Figure 1: *BliteC* workflow

a package, containing the corresponding WS-BPEL program and the associated WSDL and process deployment descriptors, that is immediately executable by ActiveBPEL. This way, *BliteC* further simplifies the programmers work by also automating the deployment procedure. The workflow of use of *BliteC* is graphically summarized in Figure 1.

The rest of the paper is organized as follows. Section 2 provides a brief summary of WS-BPEL, introduces a running example and, by means of many specific programs illustrates major loose points of WS-BPEL and the tests carried out with the three previously mentioned WS-BPEL engines. Section 3 presents *Blite*'s syntax and operational semantics, and shows some example applications. Section 4 illustrates the main features of *BliteC* and the correspondence between *Blite* constructs and WS-BPEL activities. Finally, Section 5 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 a running example borrowed from the official WS-BPEL specification. We also present some illustrative WS-BPEL programs and use them to test and compare the behaviour of the three freely available engines<sup>1</sup> ActiveBPEL [4], Apache ODE [11] and Oracle BPEL Process Manager [3]. We conclude with an evaluation of the results of our experiments.

<sup>1</sup>The former two are open source projects, whereas the latter is distributed under the Oracle Technology Network Developer License. ActiveBPEL and Oracle BPEL Process Manager are also part of (commercial) tool suites, ActiveVOS and Oracle SOA Suite resp., for designing, developing, testing and deploying WS-BPEL applications.

## 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 Services 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. For the specification of orchestration, WS-BPEL provides many different activities that are distinguished between *basic activities* and *structured activities*. Orchestration exploits state information that is stored in variables and managed through message correlation. In fact, when messages are sent/received, the value of their parameters is stored in variables. Likewise block structured languages, the scope of variables extends to the whole immediately enclosing `<scope>` (or `<process>`) activity.

The basic activities are: `<invoke>`, to invoke an operation offered by a WS; `<receive>`, to wait for an invocation to arrive; `<reply>`, to send a message in reply to a previously received invocation; `<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. Notably, `<reply>` can be combined with `<receive>` to model request-response interactions.

The structured activities describe the control flow logic of a business process by composing basic and/or structured activities recursively. The structured activities are: `<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. Activities within a `<flow>` can be further synchronised by means of *flow links*. These are conditional transitions connecting activities to form directed acyclic graphs and are such that a target activity may only start when all its source activities have completed and the condition on the incoming flow links 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. Such a relationship is expressed at the WSDL level by specifying the roles played by each of the services in the interaction. However, this information is not enough to deliver messages to a service. 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* [31] 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.

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

-  to label an activity that initializes correlated variables;
-  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.

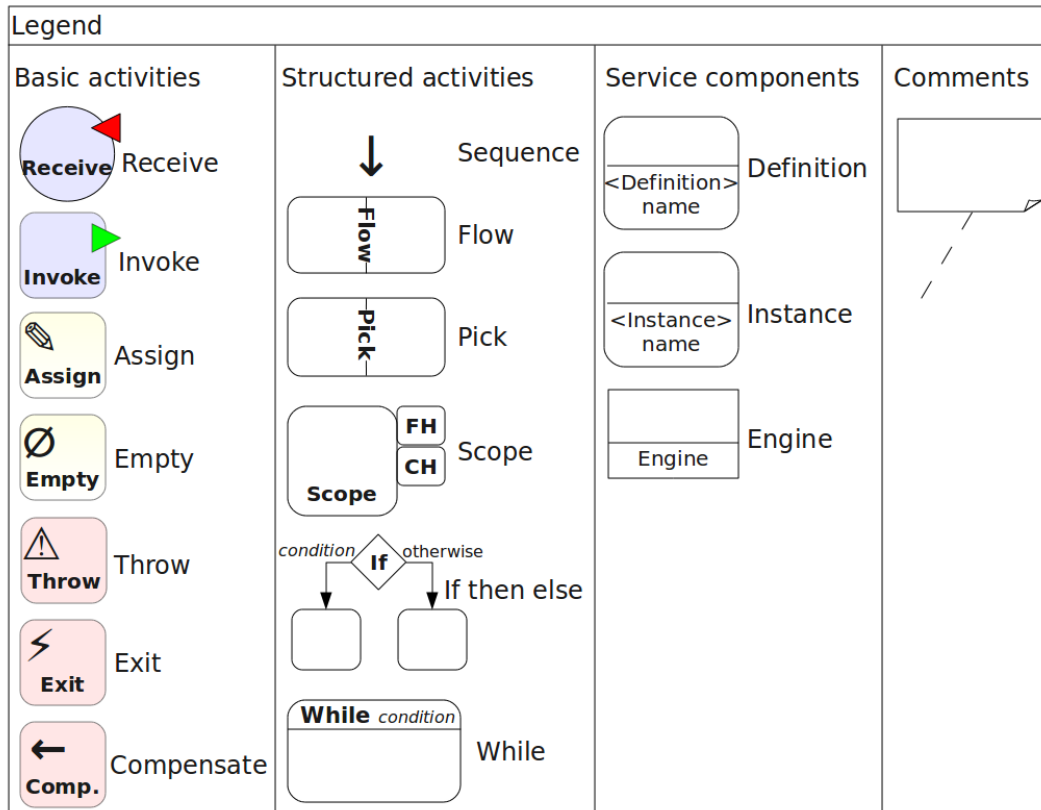


Figure 2: A graphical representation of WS-BPEL basic/structured activities and service components

Notably, in Figure 2 are reported only the WS-BPEL constructs that are relevant for the investigation conducted in Section 2.3 and that, in our opinion, are absolutely necessary to conveniently program service orchestrations. Therefore, in the rest of the paper, we shall not take into account flow links, event and termination handlers, and activities `<wait>`, `<compensateScope>`, `<validate>`, `<extensionActivity>`, `<repeatUntil>` and `<forEach>`. We leave for future work the extension of our investigation to the above constructs, as we argue in Section 5.

## 2.2. A shipping service in WS-BPEL

Our running example is a shipping service drawn from the official specification of WS-BPEL [49, Section 15.1]. In this section, this example will allow us to illustrate most of the language features, including communication activities, correlation sets, shared variables and control flow structures.

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 then
  send shipNotice
else
  itemsShipped := 0
  while itemsShipped < items do
    itemCount := getAvailableItems
    send shipNotice
    itemsShipped := itemsShipped + itemCount
```

The corresponding WS-BPEL program is graphically depicted in Figure 3. We comment below some excerpts of its code<sup>2</sup>. The definition starts with a receive activity for a message from a client containing a request for a shipment:

```
<receive partnerLink="client" operation="ShippingRequest"
  variable="shipRequest" createInstance="yes">
  <correlations> <correlation set="shipOrderSet" initiate="yes"/> </correlations>
</receive>
```

A shipping request, stored in `shipRequest`, is a structured information consisting of an order identifier, which is used to correlate the shipping notice(s) with the shipping order, a boolean indicating whether the order is to be shipped complete or not, and the total number of items in the order. In the graphical notation, the variable fields storing the three parts of the message, (i.e. `id`, `shipComplete` and `items`) have been made explicit. The correlation set `shipOrderSet`, which uniquely identifies a shipping order through the identifier stored in `id`, is initialized by this receive activity (`initiate="yes"`). Whenever prompted by a client request, the service creates an instance to serve that specific request (`createInstance="yes"`) and is immediately ready to concurrently serve other requests.

Afterwards, if a complete shipment has been requested, the created service instance sends a shipping notice to the client, by means of the following invoke activity:

```
<invoke partnerLink="client" operation="ShippingNotice"
  inputVariable="shipNotice">
  <correlations> <correlation set="shipOrderSet"/> </correlations>
</invoke>
```

and terminates. A shipping notice contains an order identifier, to correlate the message to the corresponding order, and the number of shipped items. The correlation mechanism is used here to guarantee that the above invoke activity is performed only if the identifier stored within variable `shipNotice` coincides with the value stored in the correlation set `shipOrderSet` (which has been set by the receive activity).

---

<sup>2</sup>The complete code of the WS-BPEL process and the associated WSDL document can be found in [49].

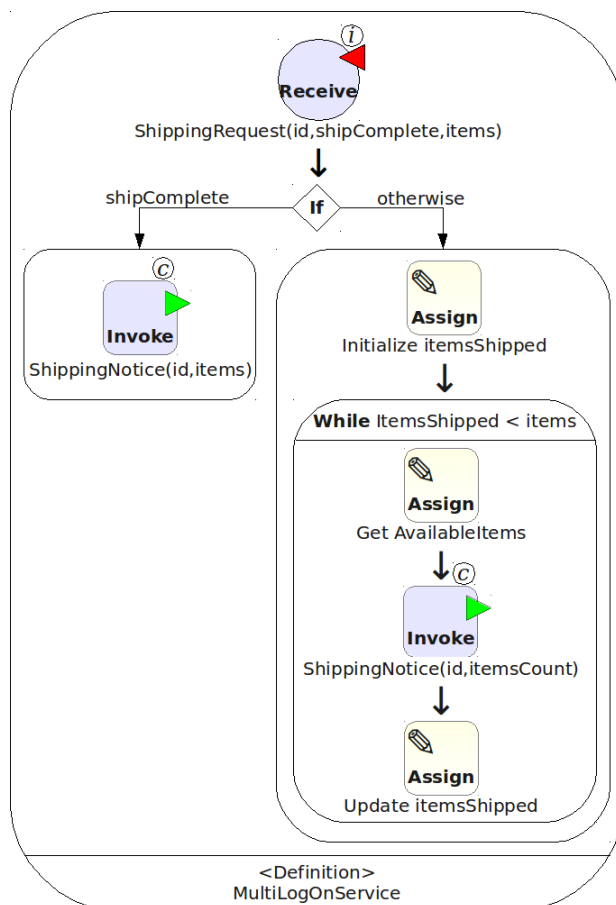


Figure 3: Graphical representation of the shipping service

Instead, if the items can be shipped piecemeal, the variable `itemsShipped`, acting as a counter for the number of items already shipped, is initialized as follows:

```
<assign> <copy> <from>0</from> <to>$itemsShipped</to> </copy> </assign>
```

Then, until all requested items have been shipped, the service instance repeatedly interacts with a back-end system to get the number of items available for a shipment, sends the corresponding shipping notice to the client and updates the value of `itemsShipped`.

### 2.3. Experimentation and assessment of three WS-BPEL engines

We now present the WS-BPEL programs used to test and compare the three engines. For our evaluation, we have taken into account fundamental features of WS-BPEL that remained unchanged since its initial version [25].

*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

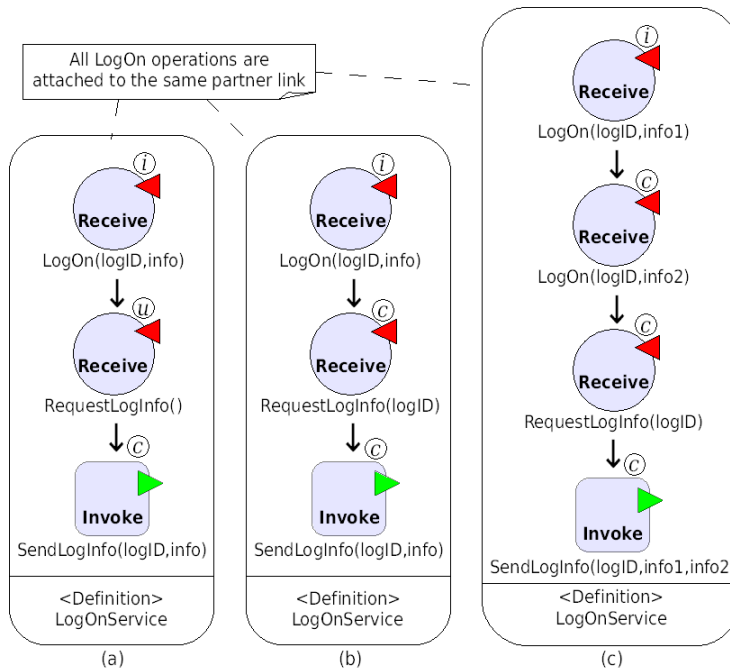


Figure 4: Message correlation

asynchronously obtained by implementing the callback operation `SendLogInfo` (on the use of asynchronous request-response patterns in service-oriented applications see also 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 4(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<sup>3</sup>. After that, the service can reply (and terminate) by sending the requested information to the client. Notably, the WS-BPEL process in Figure 4(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 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 4(b).

<sup>3</sup>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.

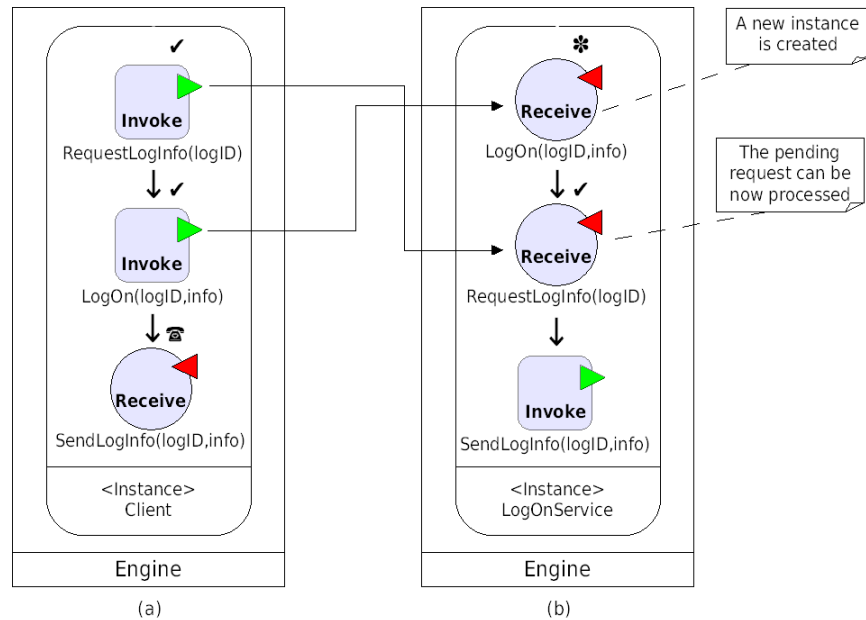


Figure 5: Asynchronous message delivering

A special case is when the two initial receives are on the same partner and operation, as in Figure 4(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 [49, Section 10.4] that does not mention that possible conflicting receives could arise. Now, let us assume that a client process has performed two log-on requests. This, accordingly to what seems to be 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, thus preventing creation of a wrong new instance. On the contrary, when executing this example, Oracle BPEL creates two instances, one for each received request. An important consequence, and indeed an unexpected side effect, is that the created instances are in conflict and will soon get stuck. Despite this behaviour can be reasonably considered wrong, the WS-BPEL specification does not explicitly forbid it.

*Example 2.2: Asynchronous message delivering.* In service-oriented systems communication paradigms are usually asynchronous (mainly for scalability reasons [17]), 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 [49, 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 to the previous service

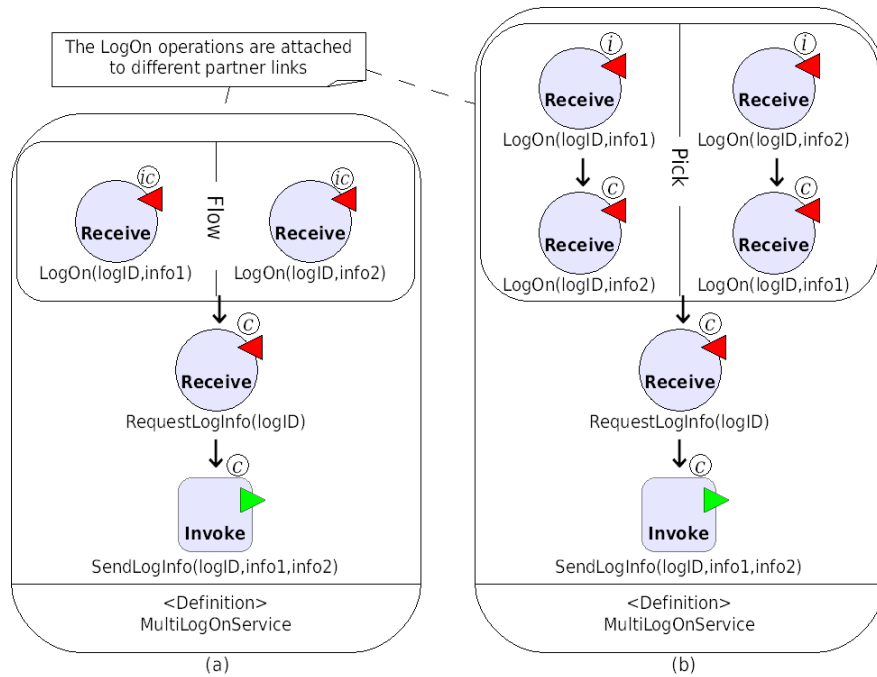


Figure 6: Multiple start activities

depicted in Figure 4(b). After the request for some user information is sent by the first invoke activity, a service instance is created as a result of consumption of the request for logging on to 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. Notably, messages that do not match the signature of any operation provided by a service (e.g. the numbers of parameters are different) are rejected by the engine and, as expected, do not affect the execution of running instances.

*Example 2.3: Multiple start and conflicting receive activities.* When defining services, the WS-BPEL specification permits using multiple start activities [49, 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

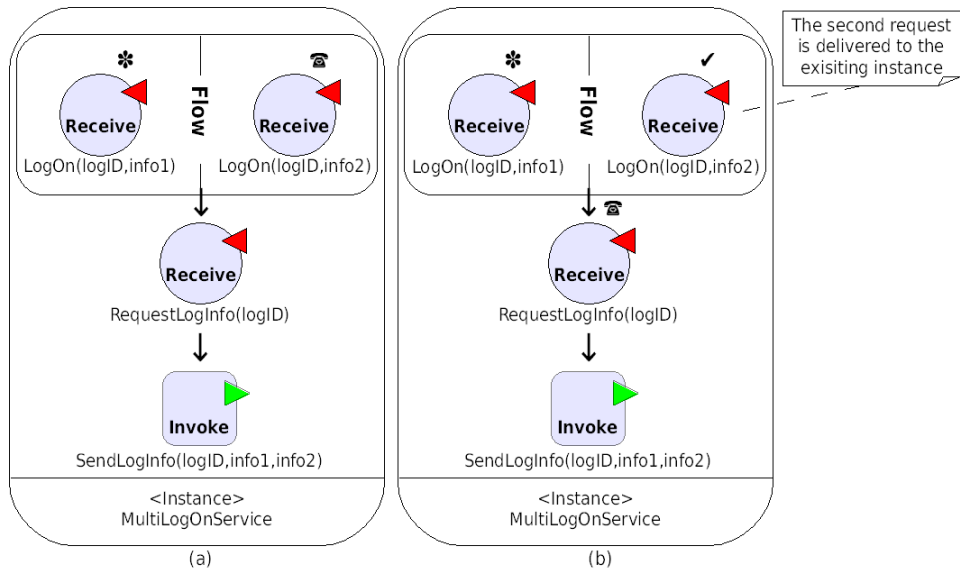


Figure 7: Multiple start activities: service instantiation

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 stored in 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 (i.e. the instance in Figure 7(a) should only reduce to that of Figure 7(b)).

However, in case of conflicting receives, the WS-BPEL specification document prescribes to raise the standard fault `bpel:conflictingReceive`, which seems to be somehow in 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. In our example, both the definition and the instance can perform a receive over `LogOn(logID,info2)` using the same partner link. Anyway, 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.

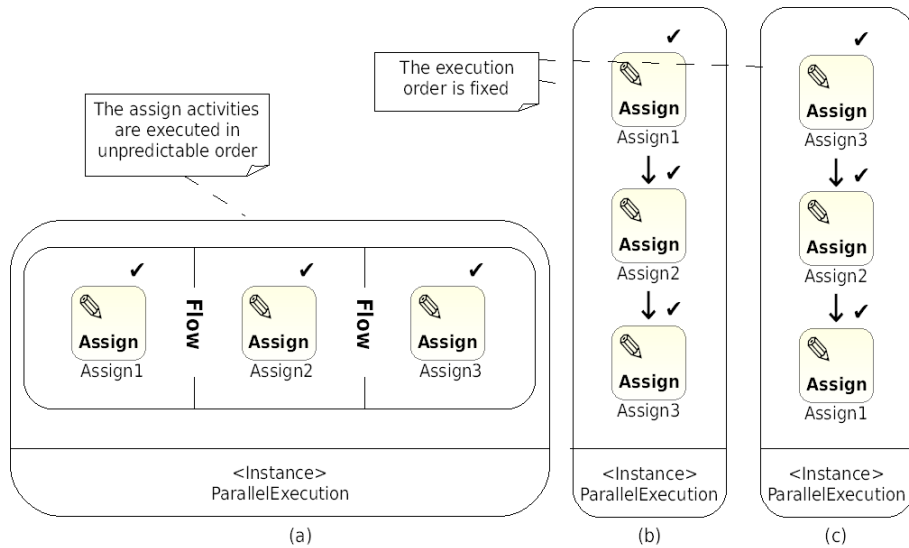


Figure 8: Scheduling of parallel activities

*Example 2.4: Scheduling of parallel activities.* While using the WS-BPEL engines, we have also experienced 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. One could argue that all the three engines comply with the WS-BPEL specification that only prescribes to execute parallel activities in an arbitrary order. In fact this is true but, as a matter of fact, this requirement is too permissive, since it has left room for different implementations and, hence, contributed to limit the portability of WS-BPEL programs.

*Example 2.5: Forced termination.* The WS-BPEL specification [49, 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 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

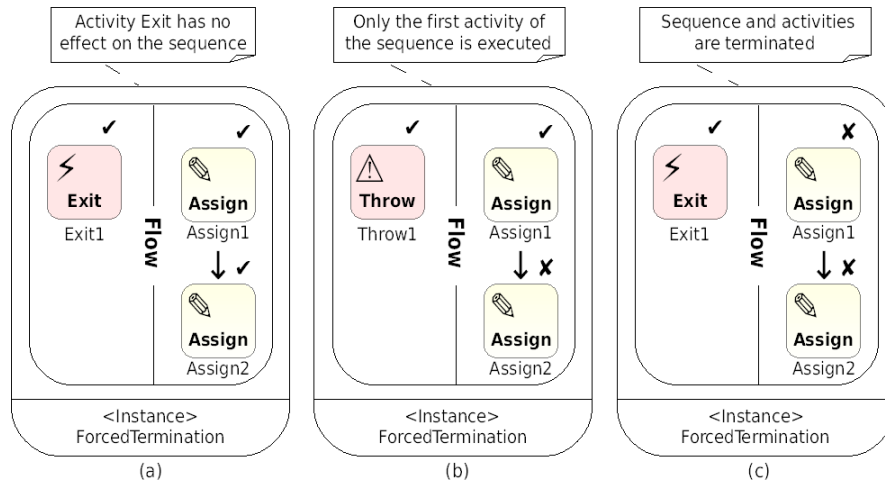


Figure 9: Forced termination

running activities must be terminated as soon as possible (Figure 9(c)) without any fault handling or compensation [49, Section 10.10]. However, differently from what the WS-BPEL specification seems to suggest, 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.

Notably, since the flow activity is differently implemented by the three engines (see Example 2.4), in the examples depicted in Figure 9 we have managed to guarantee that in each engine `<exit>` and `<throw>` are executed before the sequence of assignments (this way, the effect of the execution does not depend on the scheduling of the 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 [49, Section 10.10], it seems that execution of activities `<throw>` and `<exit>` should have higher priority than execution of the remaining ones. For example, consider again a sequence of two assign activities. By executing a parallel `<throw>` activity, the whole process should only reduce as shown in Figure 10(a). This is indeed the behaviour we experienced with ActiveBPEL. Instead, Oracle BPEL and Apache ODE seem not to implement any prioritized behavior for activities forcing termination and in fact they allow the above process to also evolve by firstly performing the first `<assign>` activity and then the `<throw>`, as shown in Figure 10(b). By suitably ordering the arguments of the flow activity, we have managed to guarantee that the results are not a consequence of the engines' different scheduling policies.



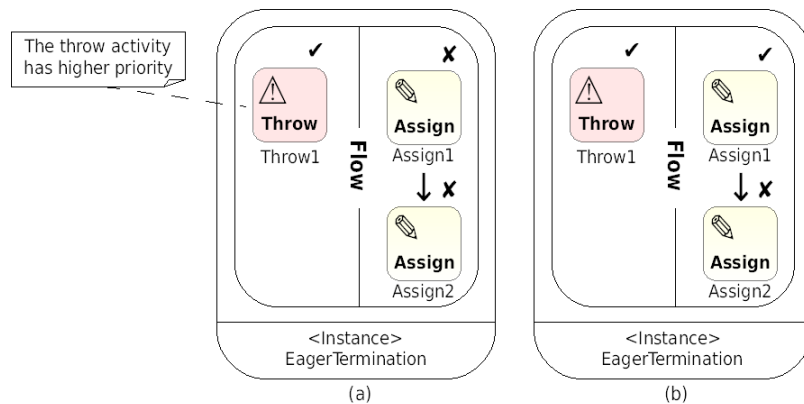


Figure 10: Eager execution of activities causing termination

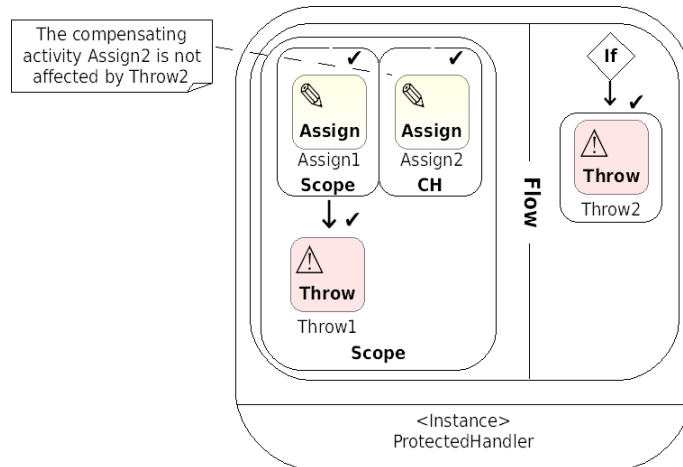


Figure 11: Handlers protection

*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. 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 [49, Sections 12.5 and 10.10]. Firstly, compensation handlers

	Oracle BPEL	ActiveBPEL	Apache ODE
Correlation (Ex.1)	👍	👍	👍
Consecutive conflicting receives (Ex.1)	👎 creates two conflicting instances	👍 creates one instance	👍 creates one instance
Asynchronous message delivering (Ex.2)	👍	👍	👍
Multiple start activities (Ex.3)	👎 raises a fault	👍 creates one instance	👎 doesn't support multiple start activities
Scheduling of parallel activities (Ex.4)	👎 fixed execution order	👎 fixed execution order	👍 unpredictable execution order
Short-lived activities (Ex.5)	👍 distinguishes short-lived activities	👎 doesn't distinguish short-lived activities	👍 distinguishes short-lived activities
Forced Termination (Ex.5)	👎 <exit> has no termination effect	👍	👍
Eager execution (Ex.6)	👎 lazy termination	👍 eager termination	👎 lazy termination
Handlers protection and installation (Ex.7)	👎 doesn't protect handlers and allows faulty scopes' compensation	👍 protects handlers and doesn't allow faulty scopes' compensation	👍 protects handlers and doesn't allow faulty scopes' compensation

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

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 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 tested implement different semantics, which implies that WS-BPEL programs are not portable. We have used ‘thumbs up’ and ‘thumbs down’ to indicate whether the engines comply with what we reasonably believe to be the intended semantics of WS-BPEL. 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 noting the limited support for the multiple start pattern and the eager execution as for 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 choose it for implementing and deploying *Blite* applications.

We believe that the engines’ different behaviour we have experienced is a consequence of the lack of a formal reference semantics for WS-BPEL, that would have disambiguated the intricate and complex features of the language, leaving less room for interpretation by implementers. On the other hand, thanks to the quite intuitive and direct correspondence

of *Blite* with a meaningful sublanguage of WS-BPEL we illustrate in Section 4.2, we can confidently state that *Blite*'s formal semantics provides this WS-BPEL's sublanguage with a rigorous semantics. Therefore we believe that our work, and works with similar goals, can serve both for driving implementation of new engines and 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 [49]. 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. *Blite*: a prototypical orchestration language

*Blite* is a prototypical orchestration language that results from distilling out of WS-BPEL those features that are, in our opinion, absolutely necessary to formally define the basic elements of service orchestrations and to characterise service engines. Its design has been driven by the aim of keeping the semantics of the language rigorous but manageable, while preserving a close correspondence with the procedural part of WS-BPEL. Thus, *Blite* retains partners and partner links, message correlation, concurrency, service instance creation/identification, long-running business transactions, and (a limited form of) fault and compensation handlers, while disregards request-response interactions, synchronization dependencies within flow activities, timed activities, event and termination handlers.

Moreover, the language provides a formal description of service deployments by only keeping relevant implementation details. Thus, the roles played by service partners in a service interaction are explicitly indicated by *partner links* and *partners*, while such aspects as physical *service binding* are abstracted away. As we will see in Section 4.2, this information is dealt with separately in the declarative parts associated to *Blite* programs, in order to allow *BliteC* to generate the corresponding WSDL documents and process deployment descriptors.

Before formally defining *Blite*, we provide some insights into its main features by means of the running example introduced in Section 2.2. We conclude by showing the specification in *Blite* of some WS-BPEL programs presented in Section 2.3.

### 3.1. The shipping service in Blite

The shipping service can be rendered in *Blite* as a *deployment*  $\{ [r \bullet \text{empty}] \}_{\{x_{id}\}}$  defining  $x_{id}$  as a correlation variable and containing the *service definition*  $[r \bullet \text{empty}]$ , where  $r$  is the primary activity of the service and  $\text{empty}$  is a fault handler that simply performs an empty activity when it catches a fault. The structured activity  $r$  is defined as follows:

$$\begin{aligned} & \text{rcv} \langle p_{\text{shipping}}, x_{\text{client}} \rangle \text{O}_{\text{shippingRequest}} \langle x_{id}, x_{\text{shipComplete}}, x_{\text{items}} \rangle ; \\ & \text{if} (x_{\text{shipComplete}}) \{ \text{inv} \langle x_{\text{client}} \rangle \text{O}_{\text{shippingNotice}} \langle x_{id}, x_{\text{items}} \rangle \} \{ a_{\text{piecemealShipment}} \} \end{aligned}$$

where the activity  $a_{\text{piecemealShipment}}$  is

$$\begin{aligned} & x_{\text{itemsShipped}} := 0 ; \text{while} (x_{\text{itemsShipped}} < x_{\text{items}}) \{ \\ & \quad x_{\text{itemsCount}} := \text{rand}(x_{\text{items}} - x_{\text{itemsShipped}}) ; \\ & \quad \text{inv} \langle x_{\text{client}} \rangle \text{O}_{\text{shippingNotice}} \langle x_{id}, x_{\text{itemsCount}} \rangle ; x_{\text{itemsShipped}} := x_{\text{itemsShipped}} + x_{\text{itemsCount}} \} \end{aligned}$$

The shipping service is instantiated by a *receive* activity, which is denoted by  $\text{rcv}$  and takes as arguments the *partner link*  $\langle p_{\text{shipping}}, x_{\text{client}} \rangle$ , the *operation*  $\text{O}_{\text{shippingRequest}}$  used to receive the shipping request, and the tuple of variables  $\langle x_{id}, x_{\text{shipComplete}}, x_{\text{items}} \rangle$  used for storing the request message.  $p_{\text{shipping}}$  is the partner associated to the shipping service, while  $x_{\text{client}}$  is a variable used to store the partner for sending shipping notices to the client. If a complete shipment has been requested, i.e. the boolean value  $\text{tt}$  is assigned to the variable  $x_{\text{shipComplete}}$ , the created service instance sends a shipping notice (i.e. a tuple composed of the order identifier and the number of items in the shipment) to the client by invoking the operation  $\text{O}_{\text{shippingNotice}}$  through an *invoke* activity, denoted by  $\text{inv}$ . Otherwise, variable  $x_{\text{itemsShipped}}$  is set to 0, by means of an *assign* activity, and a *while* loop is entered. At each step, the variable  $x_{\text{itemsCount}}$  is assigned the number of items available for a single shipment that is randomly computed by function  $\text{rand}(\cdot)$ . We are assuming that the function call  $\text{rand}(k)$  returns a random integer number greater than 0 and not greater than  $k$ , and represents an internal interaction with a back-end system (that, as in [49], we do not further describe). Then, a shipping notice containing the computed number is sent to the client through an invocation of operation  $\text{O}_{\text{shippingNotice}}$  and, before the end of the while step, the value of  $x_{\text{itemsShipped}}$  is updated.

Now, consider the following composition of the above deployment containing the shipping service definition and a deployment containing an *instance* of a client service

$$\{ [r \bullet \text{empty}] \}_{\{x_{id}\}} \parallel \{ \mu_{\text{client}} \vdash \text{inv} \langle p_{\text{shipping}}, p_{\text{client}} \rangle \text{O}_{\text{shippingRequest}} \langle y_{id}, y_{\text{shipComplete}}, y_{\text{items}} \rangle ; a_{\text{client}} \}_{\{y_{id}\}}$$

where  $\mu_{\text{client}}$  is the *state* of the client instance that maps variables  $y_{id}$ ,  $y_{\text{shipComplete}}$  and  $y_{\text{items}}$  to values 123, ff and 50, respectively. The client instance can perform the invoke activity and, hence, sends the *message*  $\ll \langle p_{\text{shipping}}, p_{\text{client}} \rangle : \text{O}_{\text{shippingRequest}} : \langle 123, \text{ff}, 50 \rangle \gg$  to the shipping service deployment, which causes the instantiation of the service definition. Thus, the system can evolve to

$$\{[r \bullet \text{empty}] , \mu_{shipping} \vdash [\text{if}(x_{shipComplete}) \{ \dots \} \{ \dots \} \bullet \text{empty}] \}_{\{x_{id}\}} \parallel \{ \mu_{client} \vdash a_{client} \}_{\{y_{id}\}}$$

where  $\mu_{shipping}$  maps  $x_{client}$ ,  $x_{id}$ ,  $x_{shipComplete}$  and  $x_{items}$  to  $p_{client}$ , 123, ff and 50, respectively.

To illustrate the constructs provided by *Blite* for dealing with faults and compensations, we consider an extension of the shipping service whose first activity of  $a_{piecemealShipment}$  is the *scope*  $[a_{priceCalculation} \bullet \text{throw} \star (a_{compDepartment} \mid a_{compClient})]$  where the primary activity  $a_{priceCalculation}$  calculates the shipping price according to the value assigned to  $x_{items}$  and sends the result to the accounts department, activity  $\text{throw}$  is the fault handler that propagates faults raised within the primary activity to the enclosing scope (like the  $\langle \text{rethrow} \rangle$  activity of WS-BPEL), and terms  $a_{compDepartment}$  and  $a_{compClient}$  are compensation activities that, respectively, send information about the non-shipped items to the accounts department and send a refund to the client according to the ratio (stored in  $x_{ratio}$ ) between the shipped items and the required ones. The compensation activities are composed by using the *parallel composition* operator  $\cdot \mid \cdot$  (a sort of WS-BPEL flow activity) that allows the two activities to be concurrently executed. We do not further describe the activities  $a_{priceCalculation}$ ,  $a_{compDepartment}$  and  $a_{compClient}$ . Moreover, we allow the shipping service to generate a fault, by means of an activity  $\text{throw}$  (within the body of the while construct), in case the shipping company has ended the stock of items (this is modelled by function call  $\text{rand}(k)$  returning an integer less than or equal to 0). The fault is handled by sending an error message to the client, through the invocation of operation  $o_{error}$ , and by compensating the inner scope, that has already successfully completed. Therefore, the activity  $a_{piecemealShipment}$  becomes the following scope activity

$$[a_{piecemealShipmentWithFault} \bullet (x_{msg} := \text{"sorry"}; \text{inv} \langle x_{client} \rangle o_{error} \langle x_{id}, x_{msg} \rangle) \star \text{empty}]$$

where the primary activity  $a_{piecemealShipmentWithFault}$  is

$$\begin{aligned} & [a_{priceCalculation} \bullet \text{throw} \star (a_{compDepartment} \mid a_{compClient})] ; \\ & x_{itemsShipped} := 0 ; \\ & \text{while} (x_{itemsShipped} < x_{items}) \{ \\ & \quad x_{itemsCount} := \text{rand}(x_{items} - x_{itemsShipped}) ; \\ & \quad \text{if} (x_{itemsCount} \leq 0) \{ x_{ratio} := x_{itemsShipped} / x_{items} ; \text{throw} \} \\ & \quad \{ \text{inv} \langle x_{client} \rangle o_{shippingNotice} \langle x_{id}, x_{itemsCount} \rangle ; x_{itemsShipped} := x_{itemsShipped} + x_{itemsCount} \} \} \end{aligned}$$

Consider again the system composed of the shipping service and client deployments, where now we specify  $a_{client}$  as the following term

$$\begin{aligned} & y_{itemsShipped} := 0 ; \text{while} (y_{itemsShipped} < y_{items}) \{ \\ & \quad \text{rcv} \langle p_{client} \rangle o_{shippingNotice} \langle y_{id}, y_{itemsCount} \rangle ; y_{itemsShipped} := y_{itemsShipped} + y_{itemsCount} \\ & \quad + \text{rcv} \langle p_{client} \rangle o_{error} \langle y_{id}, y_{err} \rangle ; \text{exit} \} \end{aligned}$$

The client instance requests a piecemeal shipment and then waits the shipping notices until either the shipment is completely fulfilled or an error message is received. Notably, we

<i>Deployments</i>	$d ::= \{s\}_c \mid d_1 \parallel d_2$	deployment, composition
<i>Services</i>	$s ::= [r \bullet a_f] \mid \mu \vdash a \mid \mu \vdash a, s$	definition, instance, multiset
<i>Start activities</i>	$r ::= \text{rcv } \ell^r \circ \bar{x} \mid \sum_{j \in J} \text{rcv } \ell_j^r \circ_j \bar{x}_j ; a_j$ $\mid r ; a \mid r_1 \parallel r_2 \mid [r \bullet a_f \star a_c]$	receive, pick sequence, parallel, scope
<i>Structured activities</i>	$a ::= b \mid \text{if}(e)\{a_1\}\{a_2\} \mid \text{while}(e)\{a\}$ $\mid a_1 ; a_2 \mid \sum_{j \in J} \text{rcv } \ell_j^r \circ_j \bar{x}_j ; a_j$ $\mid a_1 \parallel a_2 \mid [a \bullet a_f \star a_c]$	basic, conditional, iteration sequence, pick (with $ J  > 1$ ) parallel, scope
<i>Basic activities</i>	$b ::= \text{inv } \ell^i \circ \bar{x} \mid \text{rcv } \ell^r \circ \bar{x} \mid x := e$ $\mid \text{empty} \mid \text{throw} \mid \text{exit}$	invoke, receive, assign empty, throw, exit

Table 2: Syntax of *Blite*

exploit here the *pick* operator  $\cdot + \cdot$  (a sort of receive-guarded choice) to make a conditional choice based on the received message, and the *termination* activity `exit` to immediately terminate the execution of the client instance when an error message is received.

A possible computation of the system is as follows: (1) the client instance invokes the operation  $\text{o}_{shippingRequest}$  with arguments  $\langle 123, \text{ff}, 50 \rangle$ ; (2) a shipping service instance is created and the inner scope  $[a_{priceCalculation} \bullet \text{throw} \star (a_{compDepartment} \parallel a_{compClient})]$  successfully completes; (3) function call  $\text{rand}(50)$  returns 20 and a shipping notice with values  $\langle 123, 20 \rangle$  is sent to the client instance through operation  $\text{o}_{shippingNotice}$ ; (4) function call  $\text{rand}(30)$  returns 0 and a fault is raised; (5) the compensation activities  $a_{compDepartment}$  and  $a_{compClient}$  and, then, the fault handling activity  $\text{inv } \langle x_{client} \rangle \text{o}_{error} \langle x_{id}, x_{msg} \rangle$ , are executed.

### 3.2. Syntax

The syntax of *Blite* is given in Table 2. Besides basic and structured activities, *Blite* provides a syntax for specifying deployments, service instances and definitions, and the auxiliary category of start activities (namely a restricted form of structured activities that must be used for defining services).

*Deployments* are finite compositions of multisets of service *instances*  $\mu \vdash a$ , containing at most one service *definition*  $[r \bullet a_f]$  and having an associated *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. It groups a primary activity  $r$ , that offers a choice of alternative receives among multiple start activities, and a fault handling activity  $a_f$ . *Start activities*  $r$  are indeed structured activities that initially can only execute receive activities. We convene that the fault handling activity of a service definition may be omitted, in which case it is intended to be `throw`. Each service instance  $\mu \vdash a$  has its own (private) state  $\mu$ . States are (partial) functions mapping variables to values and are

written as collections of pairs of the form  $\{x \mapsto v\}$ . The state obtained by updating  $\mu$  with  $\mu'$ , written as  $\mu \circ \mu'$ , is inductively defined by:  $\mu \circ \mu'(x) = \mu'(x)$  if  $x \in \text{dom}(\mu')$  (where  $\text{dom}(\mu)$  denotes the domain of  $\mu$ ) and  $\mu(x)$  otherwise. The empty state is denoted by  $\emptyset$ .

Services are *structured activities* built from *basic activities*, i.e. service invocation  $\text{inv} \dots$ , service request processing  $\text{rcv} \dots$ , assignment  $\cdot := \cdot$ , empty activity  $\text{empty}$ , fault generation  $\text{throw}$  and instance forced termination  $\text{exit}$ , by exploiting operators for conditional choice  $\text{if}(\cdot)\{\cdot\}\{\cdot\}$ , iteration  $\text{while}(\cdot)\{\cdot\}$ , sequential composition  $\cdot ; \cdot$ , pick  $\sum_{j \in J} \text{rcv} \dots ; \cdot$  (with the constraint that  $|J| > 1$ ), parallel composition  $\cdot | \cdot$  and scope  $[\cdot \bullet \star \cdot]$ . We shall use  $\cdot + \cdot$  to abbreviate binary pick. Moreover, we let sequence have higher priority (i.e. bind more tightly) than parallel composition and pick, i.e.  $a_1 ; a_2 | a_3 ; a_4$  stands for  $(a_1 ; a_2) | (a_3 ; a_4)$  and  $a_1 ; a_2 + a_3$  stands for  $(a_1 ; a_2) + a_3$ . A scope activity  $[a \bullet a_f \star a_c]$  groups a primary activity  $a$  together with a fault handling activity  $a_f$  and a compensation activity  $a_c$ . We convene that fault and compensation activities may be omitted from a scope, in which case they are intended to be  $\text{throw}$  and  $\text{empty}$ , respectively.

Data can be shared among different activities through *shared variables* (ranged over by  $x, x', \dots$ ). The set of manipulable values (ranged over by  $v, v', \dots$ ) is left unspecified; however, we assume that it includes the set of *partner names* (ranged over by  $p, q, \dots$ ) and the set of *operation names* (ranged over by  $o, o', \dots$ ). We use  $u$  to range over partners and variables and  $w$  to range over values and variables. *Expressions* (ranged over by  $e, e', \dots$ ) are left unspecified<sup>4</sup> but contain, at least, values and variables.

Notation  $\bar{x}$  stands for tuples of objects, e.g.  $\bar{x}$  is a compact notation for denoting the tuple of variables  $\langle x_1, \dots, x_h \rangle$  (with  $h \geq 0$ ). We assume that variables in the same tuple are pairwise distinct. The special notation  $\tilde{x}$  stands for tuples of one or two objects, e.g.  $\tilde{p}$  denotes either  $\langle p_1, p_2 \rangle$  or  $\langle p_1 \rangle$ . Tuples can be constructed using a concatenation operator  $\cdot : \cdot$ , i.e.  $\langle p, u \rangle : \langle x_1, \dots, x_h \rangle$  returns  $\langle p, u, x_1, \dots, x_h \rangle$ . We will write  $Z \triangleq W$  to assign a symbolic name  $Z$  to the term  $W$ .

Partner links  $\ell^r$  of receive activities can be either  $\langle p \rangle$  or  $\langle p, u \rangle$ , 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 re-

---

<sup>4</sup>*Blite* is parametric w.r.t. the set of expressions as well as WS-BPEL is parametric w.r.t. the expression language supporting data manipulation. Since *Blite* specifications are passed as input to the *BliteC* tool to be translated into WS-BPEL programs (see in Section 4), *BliteC* accepts a simple, yet powerful, language of expressions that can be comfortably translated into XPath 1.0 [24], the language adopted by the three WS-BPEL engines we have compared in Section 2.3.

ceiving messages must be known at design-time, while the partners used to send messages in reply may be dynamically determined. Partner links  $\ell^i$  within invoke activities can be either  $\langle u \rangle$  or  $\langle u, p \rangle$ , 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 *Blite*, 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. Notwithstanding this, *Blite* partner links and operations provide enough information to tell such request-response interactions apart from those that are actually uni-directional (and indeed *BliteC* translates these constructs in two different ways, as shown in Table 7).

### 3.3. Operational semantics

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.

The semantics is defined over an enriched syntax that also includes *protected activities* ( $\langle a \rangle$ ), *unsuccessful termination* *stop*, *messages*  $\langle \tilde{p} : o : \bar{v} \rangle$  and *scopes* of the form  $[a \bullet a_f \star a_c \triangle a_d]$ . 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  $[a \bullet a_f \star a_c \triangle a_d]$  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 *Blite* 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



$a \mid \text{empty} \equiv a$	$\text{empty} ; a \equiv a ; \text{empty} \equiv a$	$\text{stop} \mid \text{stop} \equiv \text{stop}$	$\text{stop} ; a \equiv \text{stop}$
$\langle\langle a \rangle\rangle \equiv \langle a \rangle$	$\langle \text{sh} \rangle \equiv \text{sh}$	$\langle\langle \tilde{p} : o : \bar{v} \rangle\rangle \mid a \equiv \langle\langle \tilde{p} : o : \bar{v} \rangle\rangle \mid \langle a \rangle$	
$[a \bullet a_f \star a_c] \equiv [a \bullet a_f \star a_c \Delta \text{empty}]$		$\langle\langle \tilde{p} : o : \bar{v} \rangle\rangle \mid a_1 ; a_2 \equiv \langle\langle \tilde{p} : o : \bar{v} \rangle\rangle \mid (a_1 ; a_2)$	
$\langle\langle \tilde{p} : o : \bar{v} \rangle\rangle \mid a \bullet a_f \star a_c \Delta a_d \equiv \langle\langle \tilde{p} : o : \bar{v} \rangle\rangle \mid [a \bullet a_f \star a_c \Delta a_d] \quad \text{if } \neg a \Downarrow_{\text{throw}}$			
$\frac{a \equiv a' \quad a_f \equiv a'_f \quad a_c \equiv a'_c \quad a_d \equiv a'_d}{[a \bullet a_f \star a_c \Delta a_d] \equiv [a' \bullet a'_f \star a'_c \Delta a'_d]}$			
$\frac{r \equiv r' \quad a_f \equiv a'_f}{\{[r \bullet a_f], s\}_c \equiv \{s, [r' \bullet a'_f]\}_c}$		$\frac{a \equiv a'}{\{\mu \vdash a, s\}_c \equiv \{s, \mu \vdash a'\}_c}$	
$d_1 \parallel d_2 \equiv d_2 \parallel d_1$	$(d_1 \parallel d_2) \parallel d_3 \equiv d_1 \parallel (d_2 \parallel d_3)$	$\{\mu \vdash \text{empty}, s\}_c \equiv \{s\}_c$	
$\{\mu \vdash \text{stop}, s\}_c \equiv \{s\}_c$	$\{\mu \vdash \text{empty}\}_c \parallel d \equiv d$	$\{\mu \vdash \text{stop}\}_c \parallel d \equiv d$	

Table 3: Structural congruence for *Blite* activities and deployments

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  $\cdot \Downarrow_{\text{throw}}$  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  $\mu \vdash \text{empty}$  and  $\mu \vdash \text{stop}$  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  $\succ\rightarrow$ , exploits a labelled transition relation over structured activities, written  $\xrightarrow{\alpha}$ , where  $\alpha$  is generated by the grammar:

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

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

$\mu \vdash \text{inv } \ell^i \circ \bar{x} \xrightarrow{\tau} \ll \mu(\ell^i) : o : \mu(\bar{x}) \gg$ (inv)	$\text{rcv } \ell^r \circ \bar{x} \xrightarrow{? \ell^r : o : \bar{x}} \text{empty}$ (rec)	$\text{throw} \xrightarrow{\uparrow} \text{stop}$ (thr)
$\mu \vdash x := e \xrightarrow{x \leftarrow \mu(e)} \text{empty}$ (asg)	$\ll \tilde{p} : o : \bar{v} \gg \xrightarrow{! \tilde{p} : o : \bar{v}} \text{empty}$ (msg)	$\text{exit} \xrightarrow{\boxtimes} \text{stop}$ (term)
$\frac{\mu \vdash a_1 \xrightarrow{\alpha} a'_1}{\mu \vdash a_1 ; a_2 \xrightarrow{\alpha} a'_1 ; a_2}$ (seq)	$\frac{\mu \vdash a \xrightarrow{\alpha} a'}{\mu \vdash \langle a \rangle \xrightarrow{\alpha} \langle a' \rangle}$ (prot)	$\frac{h \in J}{\sum_{j \in J} \text{rcv } \ell_j^r \circ_j \bar{x}_j ; a_j \xrightarrow{? \ell_h^r : o_h : \bar{x}_h} a_h}$ (pick)
$\frac{a = \begin{cases} a_1 & \text{if } \mu(e) = \text{tt} \\ a_2 & \text{if } \mu(e) = \text{ff} \end{cases}}{\mu \vdash \text{if}(e)\{a_1\}\{a_2\} \xrightarrow{\tau} a}$ (if)	$\frac{a' = \begin{cases} a ; \text{while}(e) \{a\} & \text{if } \mu(e) = \text{tt} \\ \text{empty} & \text{if } \mu(e) = \text{ff} \end{cases}}{\mu \vdash \text{while}(e) \{a\} \xrightarrow{\tau} a'}$ (while)	
$\frac{\mu \vdash a_1 \xrightarrow{\alpha} a'_1 \quad \alpha \notin \{\boxtimes, \uparrow\} \quad \neg(a_2 \Downarrow_{\text{throw}} \vee a_2 \Downarrow_{\text{exit}})}{\mu \vdash a_1 \mid a_2 \xrightarrow{\alpha} a'_1 \mid a_2}$ (par <sub>1</sub> )	$\frac{a_1 \xrightarrow{\alpha} a'_1 \quad \alpha \in \{\boxtimes, \uparrow\}}{a_1 \mid a_2 \xrightarrow{\alpha} a'_1 \mid \text{end}(a_2)}$ (par <sub>2</sub> )	
$\frac{[\text{empty} \bullet a_f \star a_c \triangle a_d] \xrightarrow{(a_c)} \text{empty}}{[\text{stop} \bullet a_f \star a_c \triangle a_d] \xrightarrow{\tau} \langle a_d ; a_f \rangle}$ (done <sub>1</sub> )	$\frac{a \xrightarrow{(a'')} a'}{[a \bullet a_f \star a_c \triangle a_d] \xrightarrow{\tau} [a' \bullet a_f \star a_c \triangle a'' ; a_d]}$ (done <sub>3</sub> )	
$\frac{\mu \vdash a \xrightarrow{\alpha} a' \quad \alpha \notin \{\uparrow, (a'')\}}{\mu \vdash [a \bullet a_f \star a_c \triangle a_d] \xrightarrow{\alpha} [a' \bullet a_f \star a_c \triangle a_d]}$ (exec)	$\frac{a \xrightarrow{\uparrow} a'}{[a \bullet a_f \star a_c \triangle a_d] \xrightarrow{\tau} [a' \bullet a_f \star a_c \triangle a_d]}$ (fault)	

Table 4: Basic, auxiliary and structured activities

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  $\mu$ , that is omitted when unnecessary (writing  $a \xrightarrow{\alpha} a'$  instead of  $\mu \vdash a \xrightarrow{\alpha} a'$ ). Before commenting the rules, we introduce the auxiliary functions and predicates they exploit. Specifically, function  $\mu(e)$  evaluates expression  $e$  with respect to the state  $\mu$  and returns the computed value. However,  $\mu(\cdot)$  cannot be explicitly defined because the exact syntax of expressions is deliberately not specified. Predicates  $a \Downarrow_{\text{exit}}$  and  $a \Downarrow_{\text{throw}}$  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:

$$\text{exit} \Downarrow_{\text{exit}} \quad \frac{a_1 \Downarrow_{\text{exit}}}{a_1 ; a_2 \Downarrow_{\text{exit}}} \quad \frac{a \Downarrow_{\text{exit}}}{\langle a \rangle \Downarrow_{\text{exit}}} \quad \frac{a_1 \Downarrow_{\text{exit}} \vee a_2 \Downarrow_{\text{exit}}}{a_1 \mid a_2 \Downarrow_{\text{exit}}} \quad \frac{a \Downarrow_{\text{exit}}}{[a \bullet a_f \star a_c] \Downarrow_{\text{exit}}}$$

$$\begin{array}{c}
\frac{a \Downarrow_{\text{exit}}}{[a \bullet a_f \star a_c \triangle a_d] \Downarrow_{\text{exit}}} \quad \frac{a_1 \Downarrow_{\text{exit}} \wedge a_1 \equiv a_2}{a_2 \Downarrow_{\text{exit}}} \\
\text{throw} \Downarrow_{\text{throw}} \quad \frac{a_1 \Downarrow_{\text{throw}}}{a_1 ; a_2 \Downarrow_{\text{throw}}} \quad \frac{a \Downarrow_{\text{throw}}}{\langle a \rangle \Downarrow_{\text{throw}}} \quad \frac{a_1 \Downarrow_{\text{throw}} \vee a_2 \Downarrow_{\text{throw}}}{a_1 | a_2 \Downarrow_{\text{throw}}} \quad \frac{a_1 \Downarrow_{\text{throw}} \wedge a_1 \equiv a_2}{a_2 \Downarrow_{\text{throw}}}
\end{array}$$

Function  $\text{end}(\cdot)$ , 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

$$\begin{array}{lll}
\text{end}(\text{sh}) = \text{sh} & \text{end}(\langle a \rangle) = \langle a \rangle & \text{end}([a \bullet a_f \star a_c]) = [\text{end}(a) \bullet a_f \star a_c] \\
\text{end}(a_1 ; a_2) = \text{end}(a_1) & & \text{end}([a \bullet a_f \star a_c \triangle a_d]) = [\text{end}(a) \bullet a_f \star a_c \triangle a_d]
\end{array}$$

where  $a_1$  may not be congruent to empty or to  $\ll \tilde{p} : o : \bar{v} \gg$ , or to parallel compositions of them. In the remaining cases,  $\text{end}(\cdot)$  returns  $\text{stop}$ , except for parallel composition for which it acts as an homomorphism. Like the two predicates above, function  $\text{end}(\cdot)$  is closed under  $\equiv$ , i.e.  $\text{end}(a_1) = a$  and  $a_1 \equiv a_2$  imply  $\text{end}(a_2) = a$ .

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. 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  $\langle a \rangle$  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*<sub>1</sub>) and (*par*<sub>2</sub>)), except when a terminate/fault activity can be executed (rule (*par*<sub>2</sub>)), 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*<sub>1</sub>) and (*done*<sub>3</sub>), 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  $[empty \bullet a_f \star a_c \triangle a_d]$  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*<sub>1</sub>)). 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*<sub>2</sub>)). By

$\text{match}(\mathbf{c}, \mu, \mathbf{v}, \mathbf{v}) = \emptyset$	$\text{match}(\mathbf{c}, \mu, \mathbf{x}, \mathbf{v}) = \begin{cases} \{\mathbf{x} \mapsto \mathbf{v}\} & \text{if } \mathbf{x} \notin \mathbf{c} \vee (\mathbf{x} \in \mathbf{c} \wedge \mathbf{x} \notin \text{dom}(\mu)) \\ \emptyset & \text{if } \mathbf{x} \in \mathbf{c} \wedge \{\mathbf{x} \mapsto \mathbf{v}\} \in \mu \end{cases}$	
$\text{match}(\mathbf{c}, \mu, \langle \cdot \rangle, \langle \cdot \rangle) = \emptyset$	$\frac{\text{match}(\mathbf{c}, \mu, \mathbf{w}_1, \mathbf{v}_1) = \mu' \quad \text{match}(\mathbf{c}, \mu, \bar{\mathbf{w}}_2, \bar{\mathbf{v}}_2) = \mu''}{\text{match}(\mathbf{c}, \mu, (\mathbf{w}_1, \bar{\mathbf{w}}_2), (\mathbf{v}_1, \bar{\mathbf{v}}_2)) = \mu' \circ \mu''}$	
$\frac{ \text{match}(\mathbf{c}, \mu, \ell^r : \mathbf{o} : \bar{\mathbf{x}}, \tilde{\mathbf{p}} : \mathbf{o} : \bar{\mathbf{v}})  < n}{\mu \vdash \text{rcv } \ell^r \mathbf{o} \bar{\mathbf{x}} ; \mathbf{a} \Downarrow_{\tilde{\mathbf{p}} : \mathbf{o} : \bar{\mathbf{v}}}^{\mathbf{c}, n}}$	$\frac{\exists h \in J.  \text{match}(\mathbf{c}, \mu, \ell_h^r : \mathbf{o}_h : \bar{\mathbf{x}}_h, \tilde{\mathbf{p}} : \mathbf{o} : \bar{\mathbf{v}})  < n}{\mu \vdash \sum_{j \in J} \text{rcv } \ell_j^r \mathbf{o}_j \bar{\mathbf{x}}_j ; \mathbf{a}_j \Downarrow_{\tilde{\mathbf{p}} : \mathbf{o} : \bar{\mathbf{v}}}^{\mathbf{c}, n}}$	
$\frac{\mu \vdash \mathbf{a}_1 \Downarrow_{\tilde{\mathbf{p}} : \mathbf{o} : \bar{\mathbf{v}}}^{\mathbf{c}, n}}{\mu \vdash \mathbf{a}_1 ; \mathbf{a}_2 \Downarrow_{\tilde{\mathbf{p}} : \mathbf{o} : \bar{\mathbf{v}}}^{\mathbf{c}, n}}$	$\frac{\mu \vdash \mathbf{a}_1 \Downarrow_{\tilde{\mathbf{p}} : \mathbf{o} : \bar{\mathbf{v}}}^{\mathbf{c}, n} \vee \mu \vdash \mathbf{a}_2 \Downarrow_{\tilde{\mathbf{p}} : \mathbf{o} : \bar{\mathbf{v}}}^{\mathbf{c}, n}}{\mu \vdash \mathbf{a}_1   \mathbf{a}_2 \Downarrow_{\tilde{\mathbf{p}} : \mathbf{o} : \bar{\mathbf{v}}}^{\mathbf{c}, n}}$	
$\frac{\mu \vdash \mathbf{a} \Downarrow_{\tilde{\mathbf{p}} : \mathbf{o} : \bar{\mathbf{v}}}^{\mathbf{c}, n}}{\mu \vdash (\mathbf{a}) \Downarrow_{\tilde{\mathbf{p}} : \mathbf{o} : \bar{\mathbf{v}}}^{\mathbf{c}, n}}$	$\frac{\mu \vdash \mathbf{a} \Downarrow_{\tilde{\mathbf{p}} : \mathbf{o} : \bar{\mathbf{v}}}^{\mathbf{c}, n}}{\mu \vdash [\mathbf{a} \bullet \mathbf{a}_f \star \mathbf{a}_c \triangle \mathbf{a}_d] \Downarrow_{\tilde{\mathbf{p}} : \mathbf{o} : \bar{\mathbf{v}}}^{\mathbf{c}, n}}$	$\frac{\mu \vdash \mathbf{a} \Downarrow_{\tilde{\mathbf{p}} : \mathbf{o} : \bar{\mathbf{v}}}^{\mathbf{c}, n} \vee \mathbf{s} \Downarrow_{\tilde{\mathbf{p}} : \mathbf{o} : \bar{\mathbf{v}}}^{\mathbf{c}, n}}{\mu \vdash \mathbf{a}, \mathbf{s} \Downarrow_{\tilde{\mathbf{p}} : \mathbf{o} : \bar{\mathbf{v}}}^{\mathbf{c}, n}}$

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

rule ( $\text{done}_2$ ), 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 reduction relation over deployments defined in Table 6. The rules for communication and variables updating ( $(\text{com})$ ,  $(\text{new})$  and  $(\text{var})$ ) need a mechanism for checking if an assignment of some values  $\bar{\mathbf{v}}$  to  $\bar{\mathbf{w}}$  complies with the constraints imposed by the given correlation set  $\mathbf{c}$  and state  $\mu$  and, in case of success, returns a state  $\mu'$  for the variables in  $\bar{\mathbf{w}}$  that records the effect of the assignment. This mechanism is implemented by function  $\text{match}(\cdot, \cdot, \cdot, \cdot)$  defined through the rules in the upper part of Table 5. Notice that  $\text{match}(\cdot, \cdot, \cdot, \cdot)$  is undefined when  $\bar{\mathbf{w}}$  and  $\bar{\mathbf{v}}$  have different length or when  $\mathbf{x} \in \mathbf{c}$  and  $\{\mathbf{x} \mapsto \mathbf{v}'\} \in \mu$  for some  $\mathbf{v}' \neq \mathbf{v}$  (since the state  $\{\mathbf{x} \mapsto \mathbf{v}'\}$  does not comply with  $\mathbf{c}$  and  $\mu$ ). Rules  $(\text{com})$  and  $(\text{new})$  also use the auxiliary predicate  $\mathbf{s} \Downarrow_{\tilde{\mathbf{p}} : \mathbf{o} : \bar{\mathbf{v}}}^{\mathbf{c}, n}$ , defined inductively on the syntax of  $\mathbf{s}$  in the lower part of Table 5, that checks the ability of  $\mathbf{s}$  of performing a receive on operation  $\mathbf{o}$  exploiting the partner link  $\tilde{\mathbf{p}}$ , matching the tuple of values  $\bar{\mathbf{v}}$  and generating a state with fewer pairs than  $n$  that complies with  $\mathbf{c}$  and the current state of the activity performing the receive.

Finally, we comment on the rules in Table 6. By rule  $(\text{com})$ , communication can take place when two service instances perform matching receive and invoke activities comply-

$\frac{a_1 \xrightarrow{?t_1} a'_1 \quad a_2 \xrightarrow{!t_2} a'_2 \quad \text{match}(c_1, \mu_1, t_1, t_2) = \mu'_1 \quad \neg(\mu_1 \vdash a_1, s_1 \Downarrow_{t_2}^{c_1, \mu'_1})}{\{\mu_1 \vdash a_1, s_1\}_{c_1} \parallel \{\mu_2 \vdash a_2, s_2\}_{c_2} \succ \{\mu_1 \circ \mu'_1 \vdash a'_1, s_1\}_{c_1} \parallel \{\mu_2 \vdash a'_2, s_2\}_{c_2}} \text{ (com)}$
$\frac{[r \bullet a_f \star \text{empty}] \xrightarrow{?t_1} a_1 \quad a_2 \xrightarrow{!t_2} a'_2 \quad \text{match}(c_1, \emptyset, t_1, t_2) = \mu_1 \quad \neg(s_1 \Downarrow_{t_2}^{c_1, \mu_1})}{\{[r \bullet a_f], s_1\}_{c_1} \parallel \{\mu_2 \vdash a_2, s_2\}_{c_2} \succ \{\mu_1 \vdash a_1, [r \bullet a_f], s_1\}_{c_1} \parallel \{\mu_2 \vdash a'_2, s_2\}_{c_2}} \text{ (new)}$
$\frac{\mu \vdash a \xrightarrow{x \leftarrow v} a' \quad \text{match}(c, \mu, x, v) = \mu'}{\{\mu \vdash a, s\}_c \succ \{\mu \circ \mu' \vdash a', s\}_c} \text{ (var)} \quad \frac{d_1 \succ d'_1}{d_1 \parallel d_2 \succ d'_1 \parallel d_2} \text{ (part)}$
$\frac{\mu \vdash a \xrightarrow{\alpha} a' \quad \alpha \notin \{?t_1, !t_2, x \leftarrow v\}}{\{\mu \vdash a, s\}_c \succ \{\mu \vdash a', s\}_c} \text{ (enab)} \quad \frac{d \equiv d_1 \quad d_1 \succ d_2 \quad d_2 \equiv d'}{d \succ d'} \text{ (cong)}$

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

ing with the correlation set of the receiving instance. Notice that matching covers both partner link  $\tilde{p}$  and business data  $\bar{v}$ . 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  $\cdot \Downarrow \cdot$  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 of Section 2.3. In rules (com) and (new), the assumption about *well-formedness* of deployments finds full employment, because it avoids checking 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)).

### 3.4. Examples

We report here the *Blite* specifications of some WS-BPEL processes graphically presented in Section 2.3. The specifications of the other examples can be found in [41].

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

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

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

$$\{ [ (rcv \langle p_1, q \rangle \text{LogOn} \langle x_{\text{logID}}, x_{\text{info}_1} \rangle | rcv \langle p_2, q \rangle \text{LogOn} \langle x_{\text{logID}}, x_{\text{info}_2} \rangle) ; \\ rcv \langle r \rangle \text{RequestLogInfo} \langle x_{\text{logID}} \rangle ; inv \langle q \rangle \text{SendLogInfo} \langle x_{\text{logID}}, x_{\text{info}_1}, x_{\text{info}_2} \rangle ] \}_{\{x_{\text{logID}}\}} \\ || \{ \{x \mapsto id, y \mapsto d_1\} \vdash inv \langle p_1, q \rangle \text{LogOn} \langle x, y \rangle ; inv \langle r \rangle \text{RequestLogInfo} \langle x \rangle ; rcv \langle q \rangle \text{SendLogInfo} \langle x, z, k \rangle \}_{\{x\}} \\ || \{ \{x \mapsto id, y \mapsto d_2\} \vdash inv \langle p_2, q \rangle \text{LogOn} \langle x, y \rangle \}_{\{x\}}$$

After message  $\ll \langle p_1 \rangle : \text{LogOn} : \langle id, d_1 \rangle \gg$ , produced by invocation  $inv \langle p_1, q \rangle \text{LogOn} \langle x, y \rangle$ , has been processed by  $rcv \langle p_1, q \rangle \text{LogOn} \langle x_{\text{logID}}, x_{\text{data}_1} \rangle$ , the overall composition becomes

$$\{ [ (rcv \langle p_1, q \rangle \text{LogOn} \langle x_{\text{logID}}, x_{\text{info}_1} \rangle | rcv \langle p_2, q \rangle \text{LogOn} \langle x_{\text{logID}}, x_{\text{info}_2} \rangle) ; \\ rcv \langle r \rangle \text{RequestLogInfo} \langle x_{\text{logID}} \rangle ; inv \langle q \rangle \text{SendLogInfo} \langle x_{\text{logID}}, x_{\text{info}_1}, x_{\text{info}_2} \rangle, \\ \{ x_{\text{logID}} \mapsto id, x_{\text{info}_1} \mapsto d_1 \} \vdash [ rcv \langle p_2, q \rangle \text{LogOn} \langle x_{\text{logID}}, x_{\text{info}_2} \rangle ; \\ rcv \langle r \rangle \text{RequestLogInfo} \langle x_{\text{logID}} \rangle ; inv \langle q \rangle \text{SendLogInfo} \langle x_{\text{logID}}, x_{\text{info}_1}, x_{\text{info}_2} \rangle ] \}_{\{x_{\text{logID}}\}} \\ || \{ \{x \mapsto id, y \mapsto d_1\} \vdash inv \langle r \rangle \text{RequestLogInfo} \langle x \rangle ; rcv \langle q \rangle \text{SendLogInfo} \langle x, z, k \rangle \}_{\{x\}} \\ || \{ \{x \mapsto id, y \mapsto d_2\} \vdash inv \langle p_2, q \rangle \text{LogOn} \langle x, y \rangle \}_{\{x\}}$$

Now, the definition and the instance of the service compete for receiving the same message sent by the invoke activity  $inv \langle p_2, q \rangle \text{LogOn} \langle x, y \rangle$ . 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 *Blite* the above term can only reduce to

$$\{ [ (rcv \langle p_1, q \rangle \text{LogOn} \langle x_{\text{logID}}, x_{\text{info}_1} \rangle | rcv \langle p_2, q \rangle \text{LogOn} \langle x_{\text{logID}}, x_{\text{info}_2} \rangle) ; \\ rcv \langle r \rangle \text{RequestLogInfo} \langle x_{\text{logID}} \rangle ; inv \langle q \rangle \text{SendLogInfo} \langle x_{\text{logID}}, x_{\text{info}_1}, x_{\text{info}_2} \rangle, \\ \{ x_{\text{logID}} \mapsto id, x_{\text{info}_1} \mapsto d_1, x_{\text{info}_2} \mapsto d_2 \} \vdash [ rcv \langle r \rangle \text{RequestLogInfo} \langle x_{\text{logID}} \rangle ; \\ inv \langle q \rangle \text{SendLogInfo} \langle x_{\text{logID}}, x_{\text{info}_1}, x_{\text{info}_2} \rangle ] \}_{\{x_{\text{logID}}\}} \\ || \{ \{x \mapsto id, y \mapsto d_1\} \vdash inv \langle r \rangle \text{RequestLogInfo} \langle x \rangle ; rcv \langle q \rangle \text{SendLogInfo} \langle x, z, k \rangle \}_{\{x\}} \\ || \{ \{x \mapsto id, y \mapsto d_2\} \vdash \text{empty} \}_{\{x\}}$$

*Handlers protection.* The following *Blite* term corresponds to the example of Figure 11:

$$a \triangleq [ ([x := v_1 \bullet \text{throw} \star x := v_2] ; \text{throw} \bullet \text{throw} \star \text{empty}) | \text{if}(tt)\{\text{throw}\}\{\text{empty}\} \bullet \text{empty} ]$$

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

$$\begin{aligned}
\{\emptyset \vdash a\}_0 &\xrightarrow{(1)} \{\{x \mapsto v_1\} \vdash [ ([ [ \text{empty} \bullet \text{throw} \star x := v_2 ] ; \text{throw} \bullet \text{throw} \star \text{empty} ] \\
&\quad | \text{if}(\text{tt})\{\text{throw}\}\{\text{empty}\} ) \bullet \text{empty} ] ]_0 \\
&\xrightarrow{(2)} \{\{x \mapsto v_1\} \vdash [ ([ \text{empty} ; \text{throw} \bullet \text{throw} \star \text{empty} \triangle (x := v_2 ; \text{empty}) ] \\
&\quad | \text{if}(\text{tt})\{\text{throw}\}\{\text{empty}\} ) \bullet \text{empty} ] ]_0 \\
&\quad \equiv \{\{x \mapsto v_1\} \vdash [ ([ \text{throw} \bullet \text{throw} \star \text{empty} \triangle x := v_2 ] \\
&\quad | \text{if}(\text{tt})\{\text{throw}\}\{\text{empty}\} ) \bullet \text{empty} ] ]_0 \\
&\xrightarrow{(3)} \{\{x \mapsto v_1\} \vdash [ ([ \text{stop} \bullet \text{throw} \star \text{empty} \triangle x := v_2 ] \\
&\quad | \text{if}(\text{tt})\{\text{throw}\}\{\text{empty}\} ) \bullet \text{empty} ] ]_0 \\
&\xrightarrow{(4)} \{\{x \mapsto v_1\} \vdash [ ( \langle x := v_2 ; \text{throw} \rangle | \text{if}(\text{tt})\{\text{throw}\}\{\text{empty}\} ) \bullet \text{empty} ] ]_0 \\
&\xrightarrow{(5)} \{\{x \mapsto v_1\} \vdash [ ( \langle x := v_2 ; \text{throw} \rangle | \text{throw} ) \bullet \text{empty} ] ]_0 \\
&\xrightarrow{(6)} \{\{x \mapsto v_1\} \vdash [ ( \text{end}(\langle x := v_2 ; \text{throw} \rangle) | \text{stop} ) \bullet \text{empty} ] ]_0 \\
&\quad \equiv \{\{x \mapsto v_1\} \vdash [ ( \langle x := v_2 ; \text{throw} \rangle | \text{stop} ) \bullet \text{empty} ] ]_0 \\
&\xrightarrow{(7)} \{\{x \mapsto v_2\} \vdash [ ( \langle \text{throw} \rangle | \text{stop} ) \bullet \text{empty} ] ]_0
\end{aligned}$$

where we have numbered the reductions for ease of reference. 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. This activity is protected, by using the auxiliary operator  $\langle \cdot \rangle$ , from the effect of the forced termination triggered by the parallel throw activity (5-7). At this point, the computation can go on by raising the fault and then executing the empty compensation.

#### 4. **BliteC**: a tool for rapid development of WS-BPEL applications

*BliteC*<sup>5</sup> is developed in Java to guarantee its portability across different platforms, to exploit the well-established Java 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 [9] for creating and managing XML documents, JavaCC [8] for generating the parsers that validate the input documents, and JJTree [8] for allowing the parsers to build parse trees (already arranged to support the Visitor design pattern [26]). The architecture of *BliteC* is graphically depicted in Figure 12. The tool is composed of five main components:

---

<sup>5</sup>*BliteC* is a free software; it can be downloaded from <http://rap.dsi.unifi.it/blite> and redistributed and/or modified under the terms of the GNU General Public License.

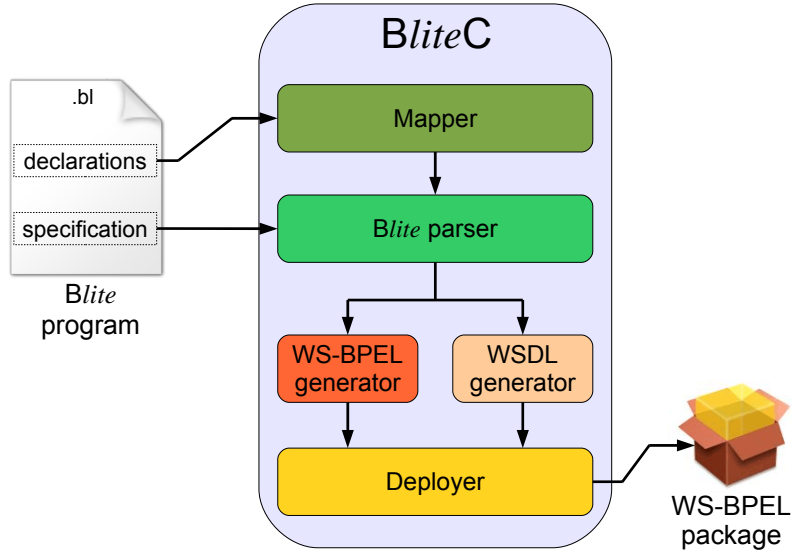


Figure 12: *BliteC* architecture

- *Mapper* parses the declarative part (see Section 4.1) of the input *Blite* program and initializes a map that associates each declared object to its name;
- *Blite parser* analyzes the *Blite* specification within the input program, completes the map created by *Mapper* and creates the parse tree of the *Blite* specification;
- *WS-BPEL* and *WSDL generators* use the data produced by the above components to generate a *WS-BPEL* process and the associated *WSDL* document;
- *Deployer* generates the process deployment descriptor and packages all created documents into a deployable file.

In the rest of the section, we present the syntax of *Blite* accepted by the tool and the declarative part of *Blite* programs through our running example, and explain the correspondence between *Blite* constructs and *WS-BPEL* activities.

#### 4.1. The shipping service in *BliteC*

A *Blite program* accepted by *BliteC* is composed of a *Blite* 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 *Blite* accepted by *BliteC* is a ‘machine readable’ version of the syntax reported in Table 2. As a matter of notation, the scope construct is denoted by  $[ a @ a_f * a_c ]$ , sequence by  $\text{seq } a_1 ; \dots ; a_n$  *qes*, flow by  $\text{flw } a_1 | \dots | a_n$  *wlf*, and pick by  $\text{pck } a_1 + \dots + a_n$  *kcp*. Notably, 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, relational and string operators, where the manipulable values are boolean, integer numbers, strings (as usual, written within double inverted commas), partner links, and literals (defined in the declarative part). Assignments can also exploit special operators for inserting and extracting data into and from XML-structured messages. Deployments only contain service definitions, as service instances are created at runtime because of service invocations. Finally, the symbol  $\triangleq$  used in Section 3 is replaced by the symbol  `::=` .

We show below the *Blite* specification<sup>6</sup>, written in the syntax accepted by *BliteC*, of the shipping service already introduced in Section 2.2 and specified in *Blite* in Section 3.1:

```

a_pieciemealShipment ::=
  seq
  x_itemsShipped := 0;
  while (x_itemsShipped < x_items) { seq
    inv <backend,cb_backend> o_num <x_id>;
    rcv <cb_backend> o_num <x_itemsCount,x_id>;
    inv <x_client> o_shippingNotice <x_id,x_itemsCount>;
    x_itemsShipped := x_itemsShipped + x_itemsCount
  }
  qes }
qes ;;

s_ship ::=
[ seq
  rcv <p_shipping,x_client> o_shippingRequest <x_id,x_shipComplete,x_items>;
  if (x_shipComplete)
    { inv <x_client> o_shippingNotice <x_id,x_items> }
  else { a_pieciemealShipment }
  qes
  @ empty];;

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

```

Here, differently from the specification in Section 3.1, we have replaced the call of function *rand*( $\cdot$ ) by a more realistic (synchronous) invocation of a back-end service.

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

Below, we show the declarative part of the shipping service specification:

```

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

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

```

---

<sup>6</sup>The complete specification, including the back-end and client services, can be found in [41].

```

VARIABLES {
  <x_id,x_shipComplete,x_items> => gen:shipOrder,<id,shipComplete,items>,
                                <xsd:int,xsd:boolean,xsd:int>;
  <x_id,x_items> => gen:shippingNoticeMsg,<id,items>,<xsd:int,xsd:int>;
  <x_id,x_itemsCount> => gen:shippingNoticeMsg;
  <x_id> => bck:id;
  <x_itemsCount,x_id> => bck:number;
  x_itemsShipped => xsd:integer; }

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

```

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. The user can specify the addresses of such documents 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. In our example, to interact with the back-end service the types of some message variables and a partner link have been imported from the WSDL document of the service, which is identified by the namespace prefix `bck`. Definitions belonging to standard namespaces (e.g. <http://www.w3.org/2001/XMLSchema>) are automatically imported.

Since WS-BPEL variables are typed, whereas *Blite* ones are not, within the VARIABLES block the user must declare the type of both local and message variables. Local variables, that can be used to temporarily store data and manipulate them, are declared by associations of the form `x => XML_Schema_type`; like e.g. `x_itemsShipped => xsd:integer`; . Messages, that are tuples of variables used either as sending source or as receiving target, can be declared in two ways:

- by using an imported message type, e.g. in `<x_itemsCount,x_id> => bck:number`; the message composed of variables `x_itemsCount` and `x_id` is typed as `number`, which is defined in the WSDL document identified by `bck`;
- by generating a new message type, e.g. the message `<x_id,x_shipComplete,x_items>` is typed as `shipOrder`, that defines messages composed of two integer parts, `id` and `items`, and a boolean part, `shipComplete`. The namespace prefix `gen` indicates that the type must be generated.

Similarly, partner links are typed in WS-BPEL and untyped in *Blite*. Therefore, except for the partner links used by the process to interact with its clients, that are automatically generated and typed by *BliteC*, the type of the other partner links must be defined within the PARTNERLINKS block. In our example, to interact with the back-end service, a partner

link with type `clientPLT` is specified, where the role `backend` is played by the partner through the port type `p_backendPT` (notice that the partner link type and the port type are defined in the imported WSDL document identified by `bck`). If the process would play also an active role in the interaction, a `MY_ROLE` association could be specified.

#### 4.2. From *Blite* to WS-BPEL

We provide here some insights about the transformation of *Blite* constructs into WS-BPEL activities which is reported in Table 7. 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 wants to produce packages intended to be executed by other WS-BPEL engines, the translation has possibly to be properly tailored. It is worth noting that, 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 *Blite* program. However, since *Blite* 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, i.e. `invoke`s and `receive`s, are translated in a different way depending on their arguments and their position in the code. Therefore, the translation of *Blite* programs proceeds in a top-down fashion and, in doing so, the WS-BPEL generator exploits the information previously collected by the Mapper and the *Blite* parser. For example, if a `receive` activity is positioned within a `pck` construct it is translated as an `<onMessage>` activity; if it is positioned after an `invoke` (in case of a request-response interaction) it is translated as a synchronous `<invoke>`; otherwise, it is simply translated as a `<receive>`. In addition to the excerpts shown in the table, the translation also settles the following activity attributes. If a `receive` is a start activity, to allow the process to be instantiated, the `createInstance` attribute must be set to `yes`. Moreover, if some correlation variables are involved, the corresponding correlation set (whose declaration is generated during the translation of the deployment term) must be specified as a further argument of the `<receive>` activity. The correlation attributes `initiate` and `pattern` are specified according to the type of the interaction.

The `invoke` activity is translated similarly; in particular, when it is used in a request-response interaction to send the response, it is translated as a `<reply>` activity. Thus, despite request-response interactions are expressed in *Blite* through pairs of uni-directional interactions, *BliteC* has enough information to tell such interactions apart from those that are actually uni-directional. The translation of the remaining basic activities and of the structured activities is straightforward. In particular, an `assign` activity involving message variables is translated by exploiting the type of such variables (defined in the declarative

<i>Blite</i>	WS-BPEL
<i>Receive activity</i>	
pck ... rcv <i>pl</i> 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 <i>pl</i> op <x1,...,xn>	<receive partnerLink="pl" operation="op" variable="x" />
<i>Invoke activity</i>	
inv <i>pl</i> op <x1,...,xn>	<invoke partnerLink="pl" operation="op" inputVariable="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" />
<i>Other basic activities</i>	
<i>x</i> := <i>e</i>	<assign> <copy> <from> <i>e</i> </from> <to> \$var_x.part_x </to> </copy> </assign>
empty	<empty />
throw	<throw />
exit	<exit />
<i>Structured activities</i>	
if ( <i>e</i> ) { <i>a</i> <sub>1</sub> } else { <i>a</i> <sub>2</sub> }	<if> <condition> <i>e</i> </condition> <i>a</i> <sub>1</sub> <else> <i>a</i> <sub>2</sub> </else> </if>
while ( <i>e</i> ) { <i>a</i> }	<while> <condition> <i>e</i> </condition> <i>a</i> </while>
seq <i>a</i> <sub>1</sub> ; ... ; <i>a</i> <sub><i>n</i></sub> qes	<sequence> <i>a</i> <sub>1</sub> ... <i>a</i> <sub><i>n</i></sub> </sequence>
flw <i>a</i> <sub>1</sub>   ...   <i>a</i> <sub><i>n</i></sub> wlf	<flow> <i>a</i> <sub>1</sub> ... <i>a</i> <sub><i>n</i></sub> </flow>
pck <i>a</i> <sub>1</sub> + ... + <i>a</i> <sub><i>n</i></sub> kcp	<pick> <i>a</i> <sub>1</sub> ... <i>a</i> <sub><i>n</i></sub> </pick>
[ <i>a</i> @ <i>a</i> <sub><i>f</i></sub> * <i>a</i> <sub><i>c</i></sub> ]	<scope> <faultHandlers> <catchAll> <sequence> <compensate/> <i>a</i> <sub><i>f</i></sub> </sequence> </catchAll> </faultHandlers> <compensationHandler> <i>a</i> <sub><i>c</i></sub> </compensationHandler> <i>a</i> </scope>
<i>Service definitions</i>	
[ <i>a</i> @ <i>a</i> <sub><i>f</i></sub> ]	<process> <faultHandlers> <catchAll> <sequence> <compensate/> <i>a</i> <sub><i>f</i></sub> </sequence> </catchAll> </faultHandlers> <i>a</i> </process>

Table 7: Transformation of *Blite* constructs into WS-BPEL activities

part) to identify the involved parts. A service definitions is translated as a scope, where the compensation handler is missing and the tag `<scope>` is replaced by `<process>`.

Finally, a *Blite* deployment is rendered as a WS-BPEL package whose WS-BPEL document contains the translation of the service definition within the deployment, while a composition of deployments results in a collection of WS-BPEL packages deployed in one or more WS-BPEL engines.

## 5. Concluding remarks

In this paper, we have argued about the difficulties that might arise when using WS-BPEL for programming business processes and introduced a framework including the orchestration language *Blite* and the software tool *BliteC* that supports 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. Our examples are very basic and some of them focus on peculiar aspects of WS-BPEL, but our tests reveal that they produce a different behaviour on the different engines. As a matter of fact, the lack of a formally defined semantics that serves as a point of reference for implementers and programmers gives rise to different interpretations and implementation choices, and prevents us from identifying possible bugs in individual engines and assessing the quality of WS-BPEL engines in an indisputable way.

As a second contribution, we have defined *Blite*, a prototypical orchestration language inspired by WS-BPEL but with a simpler syntax and a well-defined operational semantics. *Blite*'s formal semantics can help making some loose aspects of the WS-BPEL specification rigorous and can be used to drive implementations of future WS-BPEL engines. In fact, since each *Blite* construct corresponds to a WS-BPEL construct, we can exploit the semantics of *Blite* to assign a rigorous semantics to a large and quite expressive sub-language of WS-BPEL. This way, our study can also contribute to the many discussions on compensation and correlation which have been reported by the WS-BPEL Technical Committee [48] (see, e.g., discussions related to issues 66, 207 and 271).

Several rigorous semantics of WS-BPEL were indeed proposed in the literature (for an overview see [50]). Many of these efforts aim at formalizing a *complete* semantics for WS-BPEL using Petri nets [50, 43], but do not cover such dynamic aspects as service instantiation and message correlation. Other works [30, 33], instead, use process calculi and focus on small and relatively simple subsets of WS-BPEL. Another bunch of related works

[36, 45] 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 dynamic aspects are fully taken into account. A very general and flexible framework for error recovery has been recently introduced in [32]; this framework extends [33] with dynamic compensation, modelling in particular the dependency between fault handling and the request-response communication pattern.

Some other relevant related works are [19, 18, 15]. In the first two, the authors propose a formal approach to model compensation in transactional calculi and present a detailed comparison with [20]. The third one is an extension of asynchronous  $\pi$ -calculus with long-running (scoped) transactions. The language has a scope construct which plays a role similar to the scope activity of *Blite*, but, differently from the semantics we propose, it does not capture the order in which compensations should be activated according to the so-called ‘default compensation handling’ (which prescribes compensations to be activated in the reverse order w.r.t. the order of completion of the original scopes).

Concerning the language *Blite*, we intend to investigate its extension to cover some WS-BPEL constructs that at the time being have been left out, such as timed activities, event and termination handlers, and more flexible forms of fault and compensation handling involving named faults and compensation of specified scopes. Their addition to *Blite* as primitive constructs would require to significantly revise, and make more complex, the formal definition of the operational semantics of the language, while we do not envisage any major issue in extending *BliteC* to translate such constructs into WS-BPEL code, because each of them has a direct analogous in WS-BPEL. Some of the above constructs have been actually considered in a limited form. For example, as concerns compensation handling only the default behaviour has been modelled. The reason for this choice is that this automatic mechanism relieves the programmer from the task of manually implementing a compensation policy. At the same time, implementing the default behaviour is more challenging than implementing compensation of specified scopes, since in the former case the order of completion of the original scopes has to be taken into account. Instead, some other constructs have been disregarded because considered ‘redundant’, that is reasonably expressible in terms of more primitive constructs. For example, request-response operations can be expressed in *Blite* by means of two pairs of invoke and receive activities, namely bi-directional interaction is essentially rendered as a pair of uni-directional interactions. Other than from an operational point of view, this modelling is feasible also from a practical point of view. In fact, the tool *BliteC* is able to retrieve all the information that is necessary for telling apart in the automatically generated WSDL interface those operations that are actually uni-directional from those that are part of a bi-directional interaction. Similarly, flow links, a contribution coming from the WS-BPEL’s forerunner WSFL, that permit introducing synchronization dependencies within flow activities, are

expressible by means of variables and conditional tests (likewise [38]).

We have also described *BliteC*, a software tool that translates service orchestrations written in *Blite* into readily executable WS-BPEL programs. This way *BliteC* enables execution of *Blite* programs and, furthermore, facilitates programming in WS-BPEL. This latter aim 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 [3], Intalio-Designer [7], ActiveVOS Designer [5], and Eclipse BPEL Project [6]. Although their use is quite intuitive, developing large applications by using them can be awkward and annoying compared to the more classic textual approach. Indeed, graphical notations turn out to be suitable for beginner WS-BPEL programmers to represent simple business process workflows, but do not allow more expert programmers to exploit commonly used functionalities, such as e.g. copy/cut/paste, and are inappropriate for expressing some (textual) information, such as e.g. correlation sets. Moreover, graphical designers have a significant negative impact on performance during the programming phase (that is, indeed, the phase of the software development process on which we focus on), since they usually are plugins of heavy software development environments such as JDeveloper [1] and Eclipse [2]. Some other works with a similar aim are [44, 57, 10]. The first two present some tools that produce WS-BPEL processes starting, respectively, from UML- and Petri Nets-based representations of SOC applications. Due to the use of graphical representations, also these tools suffer from the problems previously mentioned. Instead, the third one proposes a mapping from a  $\pi$ -calculus based formalism into WS-BPEL. In all three approaches, only non-executable WS-BPEL processes are generated, i.e. the generated code should be thought of as a template code where, besides binding and deployment details, programmers have also to define such things as partner links, variables, port types, correlations sets, etc. by editing the generated files. Another related work is [47], which proposes a different approach to develop SOC applications that still relies on a formal language. However, input programs, rather than being translated into and deployed as WS-BPEL processes, are directly executed in a purposely developed engine.

Actually, we are also following this same line of research. We are indeed implementing (see [51]) a prototypical *Blite* engine following the dictates of *Blite*'s operational semantics. In fact, we are developing two software tools based on the Java technology: an engine for directly executing *Blite* programs and a development environment specifically designed for writing and testing them. The engine is a modular software composed of three main components: a *compiler*, which parses *Blite* programs and generates their corresponding static model; a *runtime environment*, which executes stepwise each activity of a program instance (according to the Composite design pattern [26]) by also taking into account its static model; a *communication environment*, which takes care of communication and deployment aspects. Currently, we have implemented only a *local* communication

environment, which allows *Blite* programs to only interact each other within a local engine, but we plan to extend it to support the common WSs standards WSDL, SOAP and HTTP. We also intend to investigate a different approach based on the JBI standard [35] that would require our engine to be integrated with an Enterprise Service Bus supporting such technology. Meanwhile, we are developing an IDE based on the NetBeans Platform to support easy and rapid writing of *Blite* programs and their testing through a simulation tool relying on a graphical representation of their execution.

Practically, *BliteC* and the above engine are implementations of *Blite*. Among other implementations of formal languages for WSs, we want to mention: JOLIE [47], an interpreter written in Java for a programming language based on SOCK [33]; JCaSPiS [14], a Java implementation of the service-oriented calculus CaSPiS [16] based on the IMC framework [13]; JSCL [28], a coordination middleware for services based on the event notification paradigm of Signal Calculus [29]; and PiDuce [21], a distributed run-time environment devised for experimenting web services technologies that implements a variant of asynchronous  $\pi$ -calculus extended with native XML values, datatypes and patterns.

Currently, WS-BPEL packages generated by *BliteC* are intended to be deployed on ActiveBPEL. Since ActiveBPEL is quite compliant with the WS-BPEL specification, this choice has allowed us to define a rather intuitive and direct translation, which makes us confident that the resulting WS-BPEL programs comply with the semantics of the original *Blite* orchestrations. This is of course witnessed by all the tests we have done. Actually, *BliteC* has been designed so that the generation of process deployment descriptors for different engines can be easily integrated, and we plan to enable it to produce packages also for other freely available engines, such as Oracle BPEL Process Manager, Apache ODE and Beepell [34]. Of course, to preserve the semantics of the original *Blite* orchestrations, one has to study the inner implementation of every supported engine and to possibly define a customized translation. Since there is no formal description of engine's behaviour, this study has to be carried out by means of experimental tests and, most of all, no formal proof of semantics preservation can be done.

It is also worth noticing that the semantics of *Blite* could be rather tough to render by WS-BPEL engines other than ActiveBPEL, because their behaviour may significantly differ on low-level implementation details (e.g. message queue handling) or may more strictly (or inappropriately) enforce some WS-BPEL constraints. For instance, it may happen that a process instance should receive a message from a partner according to the *Blite* semantics, while instead the message cannot be effectively accepted according to the semantics of the considered engine, due to e.g. some peculiar correlation constraint. In such a case, *BliteC* should identify the potential conflicting receives in the generated WS-BPEL program and, e.g., replace them by a single receive enabling some proper coordination activities.



At the time being, to facilitate the interoperation with non-*Blite* services, *BliteC* resorts to XSD types and XML literal definitions in the declarative part of *Blite* programs. Anyway, we plan to devise type and literal definitions specific for *Blite* and to integrate them in *BliteC*.

Our long term goal 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 WSs technology [46, 56]. In the end, this ‘proof technology’ can pave the way for the development of (semi-)automatic property validation tools (as e.g. in [27]).

As a step in this direction, in [37] we have defined an encoding from *Blite* to COWS [38], 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 *Blite* specifications (and the corresponding WS-BPEL applications) by exploiting the analytical tools already developed for COWS, such as the type system introduced in [39] to check confidentiality properties, the stochastic extension defined in [52] to enable quantitative reasoning on service behaviours, the static analysis introduced in [12] to establish properties of the flow of information between services, and the logical verification environment presented in [27] to express and check functional properties of services. Another similar approach that we plan to investigate is based on an encoding from *Blite* to UML statechart models that would enable a model checking analysis through the UMC tool [53]. This way, we would be able to specify in *Blite* a service orchestration, validate its behaviour by using formal tools, and deploy it as a WS-BPEL program.

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

**Supplementary data.** Further examples of use of *Blite* and the complete *BliteC* specification of the shipping service scenario can be found in the electronic appendix [41] associated to this article.

## References

- [1] Oracle JDeveloper. <http://www.oracle.com/technology/products/jdev>.
- [2] The Eclipse project. <http://www.eclipse.org>.

- [3] Oracle BPEL Process Manager 10.1.3, 2007. <http://www.oracle.com/technology/bpel>.
- [4] ActiveBPEL 5.0.2, 2009. <http://sourceforge.net/projects/activebpel502>.
- [5] ActiveVOS Designer 5.0.2, 2009. <http://www.activevos.com/>.
- [6] Eclipse BPEL project 0.4.0, 2009. <http://www.eclipse.org/bpel>.
- [7] Intalio|Designer Community Edition 6.0.1, 2009. <http://www.intalio.com/products/bpm/community-edition/designer>.
- [8] JavaCC 4.2, 2009. <https://javacc.dev.java.net>.
- [9] JDOM 1.1, 2009. <http://www.jdom.org>.
- [10] F. Abouzaid and J. Mullins. A Calculus for Generation, Verification and Refinement of BPEL Specifications. In *WWV, ENTCS 200*, pp. 43–65. Elsevier, 2008.
- [11] Apache Software Foundation. Apache ODE 1.3.4, 2010. <http://ode.apache.org>.
- [12] J. Bauer, F. Nielson, H. Nielson, and H. Pilegaard. Relational analysis of correlation. In *SAS, LNCS 5079*, pp. 32–46. Springer, 2008.
- [13] L. Bettini, R. De Nicola, D. Falassi, M. Lacoste, and M. Loreti. A Flexible and Modular Framework for Implementing Infrastructures for Global Computing. In *DAIS, LNCS 3543*, pp. 181–193. Springer, 2005.
- [14] L. Bettini, R. De Nicola, M. Lacoste, and M. Loreti. Implementing Session Centered Calculi. In *COORDINATION, LNCS 5052*, pp. 17–32. Springer, 2008.
- [15] L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long-running transactions. In *FMOODS, LNCS 2884*, pp. 124–138. Springer, 2003.
- [16] M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and Pipelines for Structured Service Programming. In *FMOODS, LNCS 5051*, pp. 19–38. Springer, 2008.
- [17] A. Brown, S. Johnston, and K. Kelly. Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications. Tech. rep., Rational Software, 2002.
- [18] R. Bruni, M. Butler, C. Ferreira, C. Hoare, H. Melgratti, and U. Montanari. Comparing two approaches to compensable flow composition. In *CONCUR, LNCS 3653*, pp. 383–397. Springer, 2005.
- [19] R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL*, pp. 209–220. ACM, 2005.
- [20] M. Butler and C. Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In *COORDINATION, LNCS 2949*, pp. 87–104. 2004.
- [21] S. Carpineti, C. Laneve, and L. Padovani. PiDuce - a project for experimenting Web services technologies. *Science of Computer Programming*, 74(10):777–811, 2009.

- [22] L. Cesari, A. Lapadula, R. Pugliese, and F. Tiezzi. A tool for rapid development of WS-BPEL applications. In *SAC*, pp. 2438–2442. ACM, 2010.
- [23] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. Tech. rep., W3C, 2001.
- [24] J. Clark and S. DeRose. XML Path Language (XPath) 1.0. Tech. rep., W3C, November 1999.
- [25] F. Curbera, Y. Golland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Services Version 1.0. Tech. rep., IBM, 2002.
- [26] G. Erich, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [27] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A model checking approach for verifying COWS specifications. In *FASE, LNCS 4961*, pp. 230–245. Springer, 2008.
- [28] G. Ferrari, R. Guanciale, and D. Stollo. JSCL: A middleware for service coordination. In *FORTE, LNCS 4229*, pp. 46–60. Springer, 2006.
- [29] G. Ferrari, R. Guanciale, and D. Stollo. Event based service coordination over dynamic and heterogeneous networks. In *ICSOC, LNCS 4294*, pp. 453–458. Springer, 2006.
- [30] P. Geguang, Z. Xiangpeng, W. Shuling, and Q. Zongyan. Semantics of BPEL4WS-like fault and compensation handling. In *FM, LNCS 3582*, pp. 350–365. Springer, 2005.
- [31] M. Gudgin, M. Hadley, and T. Rogers. Web Services Addressing 1.0 - Core. TR, W3C, 2006.
- [32] C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro. On the Interplay Between Fault Handling and Request-Response Service Invocations. In *ACSD*, pp. 90–199. IEEE, 2008.
- [33] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: a calculus for service oriented computing. In *ICSOC, LNCS 4294*, pp. 327–338. Springer, 2006.
- [34] T. Hallwyl, F. Henglein, and T. Hildebrandt. A standard-driven implementation of WS-BPEL 2.0. In *SAC*, pp. 2472–2476. ACM, 2010.
- [35] Java Community Process. JSR-000208 Java Business Integration 1.0, 2005. <http://jcp.org/aboutJava/communityprocess/final/jsr208/index.html>.
- [36] C. Laneve and G. Zavattaro. Foundations of web transactions. In *FoSSaCS, LNCS 3441*, pp. 282–298. Springer, 2005.
- [37] A. Lapadula. *A Formal Account of Web Services Orchestration*. PhD Thesis in Computer Science, DSI, Università degli Studi di Firenze, 2008. <http://rap.dsi.unifi.it/cows>.
- [38] A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *ESOP, LNCS 4421*, pp. 33–47. Springer, 2007.
- [39] A. Lapadula, R. Pugliese, and F. Tiezzi. Regulating data exchange in service oriented applications. In *FSEN, LNCS 4767*, pp. 223–239. Springer, 2007.

- [40] A. Lapadula, R. Pugliese, and F. Tiezzi. A formal account of WS-BPEL. In *COORDINATION, LNCS 5052*, pp. 199–215. Springer, 2008.
- [41] A. Lapadula, R. Pugliese, and F. Tiezzi. Using formal methods to develop WS-BPEL applications (supplementary data). 2010. Electronic appendix associated to the paper.
- [42] F. Leymann. Web Services Flow Language (WSFL 1.0). Tech. rep., IBM, 2001.
- [43] N. Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0. In *WS-FM, LNCS 4937*, pp. 77–91. Springer, 2008.
- [44] P. Mayer, A. Schroeder, and N. Koch. Mdd4soa: Model-driven service orchestration. In *EDOC*, pp. 203–212. IEEE, 2008.
- [45] M. Mazzara and R. Lucchi. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):96–118, 2006.
- [46] L. Meredith and S. Bjorg. Contracts and types. *Comm. of the ACM*, 46(10):41–47, 2003.
- [47] F. Montesi, C. Guidi, and G. Zavattaro. Composing Services with JOLIE. In *ECOWS*, pp. 13–22. IEEE, 2007.
- [48] OASIS WSBPEL TC. WS-BPEL issues list. [http://www.oasis-open.org/committees/download.php/20228/WS\\_BPEL\\_issues\\_list.html](http://www.oasis-open.org/committees/download.php/20228/WS_BPEL_issues_list.html).
- [49] OASIS WSBPEL TC. Web Services Business Process Execution Language Version 2.0. Tech. rep., OASIS, 2007.
- [50] 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). Tech. rep., BPM Center Report, 2005. <http://www.BPMcenter.org>.
- [51] P. Panconi. Blite-se and Blide, 2009. <http://code.google.com/p/blite-se>.
- [52] D. Prandi and P. Quaglia. Stochastic COWS. In *ICSOC, LNCS 4749*, pp. 245–256. 2007.
- [53] M. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. A state/event-based model-checking approach for the analysis of abstract system properties. *Science of Computer Programming*, 2010. To appear.
- [54] M. ter Beek, F. Mazzanti, and A. Sulova. An experience on formal analysis of a high-level graphical SOA design. In *FM+AM, LNI 179*, pp. 79–98. GI, 2010.
- [55] S. Thatte. XLANG: Web Services for Business Process Design. Tech. rep., Microsoft, 2001.
- [56] F. van Breugel and M. Koshkina. Models and verification of BPEL. Tech. rep., York University, 2006. <http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf>.
- [57] W. M. P. van der Aalst and K. B. Lassen. Translating unstructured workflow processes to readable BPEL: Theory and implementation. *Information & Software Technology*, 50(3):131–159, 2008.