A Formal Account of WS-BPEL*

Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi

Dipartimento di Sistemi e Informatica Università degli Studi di Firenze

Abstract. We introduce B*lite*, a lightweight language for web services orchestration designed around some of WS-BPEL peculiar features like partner links, process termination, message correlation, long-running business transactions and compensation handlers. B*lite* formal presentation helps clarifying some ambiguous aspects of the WS-BPEL specification, which have led to engines implementing different semantics and, thus, have undermined portability of WS-BPEL programs over different platforms. We illustrate the main features of B*lite* by means of many examples, some of which are also exploited to test and compare the behaviour of three of the most known free WS-BPEL engines.

1 Introduction

There is an ever increasing acceptance of WS-BPEL (Web Services Business Process Execution Language, [23]) as a standard language for service composition within and across multiple enterprises. The fact that it has become an OASIS standard, however, has not solved all the difficulties of using the language. Indeed, WS-BPEL comes without a formal semantics and its specification document [23], written in 'natural' language, contains a fair number of acknowledged ambiguous aspects that may lead to different interpretations. For example, the relationship between WS-BPEL (multiple) *start activities* and the mechanisms handling race conditions has not been fully explored; moreover, if suitable measures for 'protecting' such critical activities as fault and compensation handlers are not taken into account, then subtle behaviours can arise when implementing activities that cause immediate termination of other activities.

The design of WS-BPEL applications is difficult and error-prone also due to the presence of such intricate features as concurrency and race conditions, forced termination, multiple instances and message correlation, long-running business transactions and compensation handlers. It would thus benefit from the use of *formal methods* because these can provide a framework to precisely describe some aspects of an application, to state and prove its properties, and to direct attention towards issues that might otherwise be overlooked.

As a step in this direction, in this paper we introduce B*lite*, a 'lightweight' variant of WS-BPEL designed around the above mentioned features. B*lite*, being obtained by dropping redundant features from the full-fledged language, permits to focus on those fragments of the design that are more challenging and need more attention. For example, B*lite* clarifies the relationship between compensation activities and the control flow of the originating process, and illustrates the mechanisms for service instance creation and

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

D. Lea and G. Zavattaro (Eds.): COORDINATION 2008, LNCS 5052, pp. 199–215, 2008.

identification, and their interplay. Our study can also contribute to the many discussions on compensation and correlation which have been reported by the WS-BPEL technical committee [22] (see, e.g., discussions related to issues 66, 207 and 271).

Moreover, B*lite*'s formal presentation can help clarifying many ambiguous aspects of the WS-BPEL specification and, thus, can be used prescriptively to drive implementations of future WS-BPEL engines. In fact, by means of several examples, we test and compare three of the most known free BPEL engines, namely ActiveBPEL [1], Apache ODE [2] and Oracle BPEL Process Manager [3]. As a matter of fact, the considered engines exhibit quite different behaviours and diverge from the WS-BPEL specification in many important aspects. This is complicating the task of developing WS-BPEL applications and undermining their portability across different platforms.

We also believe that the formalization of WS-BPEL's operational semantics, through the introduction of B*lite*, can also 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 standards enabling the web services technology, WS-BPEL does not provide support for guided forms of application development and analysis. On the other hand, it has been shown that type systems, model checking and (bi)simulation analysis provide adequate tools to address topics relevant to the web services technology (see e.g. [21,26]). In the end, this 'proof technology' can pave the way for the development of (semi-)automatic property validation tools.

The rest of the paper is organized as follows. Section 2 presents B*lite*'s syntax and operational semantics. Section 3 illustrates most of the language features at work on modelling a shipping service scenario borrowed from the official WS-BPEL specification. Section 4 presents many peculiar examples and the results of our experimentation with the three WS-BPEL engines mentioned above. Section 5 touches upon more closely related work and directions for future work.

2 B*lite*: A 'Lightweight' Variant of WS-BPEL

The language B*lite*[1] is a simplification of WS-BPEL designed around some of its peculiar features like partner links, process termination, message correlation, long-running business transactions and compensation handlers. B*lite* is the result of the usual tension between handiness and expressiveness. Therefore, to keep the design of the language manageable, we intentionally left out other important aspects, including timeouts, event and termination handlers, flow graphs, and sophisticated forms of data handling.

B*lite* provides a formal description of service deployments by only retaining relevant implementation details such as partner links, service definitions and correlation sets. For example, the roles played by service partners in a service interaction are explicitly indicated through *partner links* and *partners*, while such aspects as physical *service binding* described in associated WSDL documents are abstracted away. In request-response interactions, for example, partner links indicate two partners because the requesting partner must provide a callback operation used by the receiving partner to send notifications.

[1] We refer the interested reader to [18] for a deeper presentation of which aspects of WS-BPEL are supported by B*lite* and their mapping.

Table 1. Syntax of B*lite*

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

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

An important aspect is that, in general, the information provided by partner links is not enough to deliver messages to a service. Indeed, since services are instantiated to serve the received requests, 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* [9] or low-level correlation methods based on SOAP headers. Specifically, B*lite* exploits *correlation variables* that permit to declare the parts of a message that are instance dependent, e.g. *order number* or *client id*, so that a message can be routed to the correct service instance on the basis of the values of the correlation variables it provides, independently of any routing mechanism.

Syntax. The syntax of B*lite* is given in Table 1. Services are *structured activities* built from *basic activities* by exploiting operators for conditional choice $\mathsf{if}(\cdot)\{\cdot\}\{\cdot\}$, iteration $\mathsf{while}(\cdot)\,\{\cdot\}$, sequential composition $\cdot\,;\cdot$, pick $\sum_{j\in J}\mathsf{rcv}\cdots;\cdot$ (i.e., external choice with the constraint that $|\,J\,| > 1$), parallel composition $\cdot \mid \cdot$ and scope $[\cdot \bullet \cdot \star \cdot]$. A scope activity $[\mathsf{a} \bullet \mathsf{a}_f \star \mathsf{a}_c]$ groups a primary activity a together with a fault handling activity a_f and a compensation activity a_c. *Start activities* r are structured activities that initially can only execute receive activities.

In the sequel, we shall use $\cdot + \cdot$ to abbreviate binary external choice. We let sequence have higher priority (i.e. bind more tightly) than parallel composition and external choice, i.e. $\mathsf{a_1}\,;\mathsf{a_2} \mid \mathsf{a_3}\,;\mathsf{a_4}$ stands for $(\mathsf{a_1}\,;\mathsf{a_2}) \mid (\mathsf{a_3}\,;\mathsf{a_4})$ and $\mathsf{a_1}\,;\mathsf{a_2} + \mathsf{a_3}$ stands for $(\mathsf{a_1}\,;\mathsf{a_2}) + \mathsf{a_3}$. Moreover, we adopt the convention that fault and compensation activities may be omitted from a scope construct, in which case they are intended to be throw and empty, respectively.

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

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

Partner links ℓ^r of receive activities can be either $\langle \mathsf{p}, \mathsf{u} \rangle$ or $\langle \mathsf{p} \rangle$, where p is the partner providing the operation and u is a partner or variable used to send messages in reply. Indeed, service partners used for receiving messages must be known at design-time, while the partners used to send messages in reply may be dynamically determined. Partner links ℓ^i within invoke activities can be either $\langle \mathsf{u}, \mathsf{p} \rangle$ or $\langle \mathsf{u} \rangle$, where u is the partner providing the operation and, possibly, p is a partner used to receive messages in reply. Like before, this latter partner must be statically known, thus it cannot be a variable.

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

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

The operational semantics of B*lite* deployments is defined in terms of a structural congruence and a reduction relation. The *structural congruence*, written $\equiv$, identifies syntactically different terms which intuitively represent the same term. It is defined as

Table 2. Structural congruence for B*lite* activities and deployments

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

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

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

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

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

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

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

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

the least congruence relation induced by a given set of equational laws. In Table 2, we explicitly show, in the upper part, the laws for empty, stop, protected activities, messages and scopes, and, in the lower part, the laws for services and deployments. Standard laws stating, e.g., that sequence is associative, parallel composition is commutative and associative, are omitted. A few observations on the structural laws are in order. Activity empty acts as the identity element both for sequence and parallel composition. Multiple stop in parallel are equivalent to just one stop, moreover stop disables subsequent activities. The protection operator is idempotent, and short-lived activities are implicitly protected, thus messages can go in/out of the scope of a protection operator. Default compensation is initially empty. Messages do not block subsequent activities and scope completion, except when throw is active in the scope (this is checked by predicate $\cdot \Downarrow_{\mathsf{throw}}$ that will be explained later on). Structural congruence is extended to scopes, instances and deployments in the obvious way. Moreover, the order in which definition and instances occur within a deployment does not matter, and deployment composition is commutative and associative. Instances like $\mu \vdash \mathsf{empty}$ and $\mu \vdash \mathsf{stop}$ are terminated and, thus, can be removed. Similarly, deployments only containing terminated instances are terminated too and can be removed.

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

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

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

Table 3. Basic, auxiliary and structured activities

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

$$\mu \vdash \mathsf{x} := \mathsf{e} \xrightarrow{\mathsf{x} \leftarrow \mu(\mathsf{e})} \mathsf{empty} \quad (\mathsf{asg}) \qquad \mathsf{throw} \xrightarrow{\lceil^*} \mathsf{stop} \quad (\mathsf{thr})$$

$$\mathsf{exit} \xrightarrow{\boxplus} \mathsf{stop} \quad (\mathsf{term}) \qquad \ll \bar{\mathsf{p}} : \mathsf{o} : \bar{\mathsf{v}} \gg \xrightarrow{!\bar{\mathsf{p}}:\mathsf{o}:\bar{\mathsf{v}}} \mathsf{empty} \quad (\mathsf{msg}) \qquad \frac{\mu \vdash \mathsf{a} \xrightarrow{\alpha} \mathsf{a}'}{\mu \vdash (\!|\mathsf{a}|\!) \xrightarrow{\alpha} (\!|\mathsf{a}'|\!)} \quad (\mathsf{prot})$$

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

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

$$\frac{\mu \vdash \mathsf{a}_1 \xrightarrow{\alpha} \mathsf{a}_1' \quad \alpha \notin \{\boxplus, \lceil^*\} \quad \neg(\mathsf{a}_2 \Downarrow_{\mathsf{throw}} \vee \mathsf{a}_2 \Downarrow_{\mathsf{exit}})}{\mu \vdash \mathsf{a}_1 \,|\, \mathsf{a}_2 \xrightarrow{\alpha} \mathsf{a}_1' \,|\, \mathsf{a}_2} \quad (\mathsf{par}_1) \qquad \frac{\mathsf{a}_1 \xrightarrow{\alpha} \mathsf{a}_1' \quad \alpha \in \{\boxplus, \lceil^*\}}{\mathsf{a}_1 \,|\, \mathsf{a}_2 \xrightarrow{\alpha} \mathsf{a}_1' \,|\, \mathsf{end}(\mathsf{a}_2)} \quad (\mathsf{par}_2)$$

$$[\mathsf{empty} \bullet \mathsf{a}_f \star \mathsf{a}_c \triangle \mathsf{a}_d] \xrightarrow{(\mathsf{a}_c)} \mathsf{empty} \quad (\mathsf{done}_1) \qquad [\mathsf{stop} \bullet \mathsf{a}_f \star \mathsf{a}_c \triangle \mathsf{a}_d] \xrightarrow{\tau} (\!|\mathsf{a}_d ; \mathsf{a}_f|\!) \quad (\mathsf{done}_2)$$

$$\frac{\mu \vdash \mathsf{a} \xrightarrow{\alpha} \mathsf{a}' \quad \alpha \notin \{\lceil^*, (\mathsf{a}'')\}}{\mu \vdash [\mathsf{a} \bullet \mathsf{a}_f \star \mathsf{a}_c \triangle \mathsf{a}_d] \xrightarrow{\alpha} [\mathsf{a}' \bullet \mathsf{a}_f \star \mathsf{a}_c \triangle \mathsf{a}_d]} \quad (\mathsf{exec})$$

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

$$\frac{\mathsf{a} \xrightarrow{\lceil^*} \mathsf{a}'}{[\mathsf{a} \bullet \mathsf{a}_f \star \mathsf{a}_c \triangle \mathsf{a}_d] \xrightarrow{\tau} [\mathsf{a}' \bullet \mathsf{a}_f \star \mathsf{a}_c \triangle \mathsf{a}_d]} \quad (\mathsf{fault})$$

signal from inside a scope; (a) indicates successful completion of a scope that can be compensated by the structured activity a.

The relation $\xrightarrow{\alpha}$ is defined by the rules in Table 3 with respect to a state μ, that is omitted when unnecessary (writing $\mathsf{a} \xrightarrow{\alpha} \mathsf{a}'$ instead of $\mu \vdash \mathsf{a} \xrightarrow{\alpha} \mathsf{a}'$). Before commenting the rules, we introduce the auxiliary functions and predicates they exploit. Specifically, the predicates $\mathsf{a} \Downarrow_{\mathsf{exit}}$ and $\mathsf{a} \Downarrow_{\mathsf{throw}}$ check the ability of a of performing exit or throw, respectively. They are defined inductively on the syntax of activities and act as an homomorphism in all cases, but for conditional choice and iteration for which they hold false, and for the following cases

$$\mathsf{exit} \Downarrow_{\mathsf{exit}} \qquad \mathsf{throw} \Downarrow_{\mathsf{throw}} \qquad \frac{\mathsf{a}_1 \Downarrow_{\mathsf{exit}}}{\mathsf{a}_1 ; \mathsf{a}_2 \Downarrow_{\mathsf{exit}}} \qquad \frac{\mathsf{a}_1 \Downarrow_{\mathsf{throw}}}{\mathsf{a}_1 ; \mathsf{a}_2 \Downarrow_{\mathsf{throw}}} \qquad \frac{\mathsf{a} \Downarrow_{\mathsf{exit}}}{[\mathsf{a} \bullet \mathsf{a}_f \star \mathsf{a}_c \triangle \mathsf{a}_d] \Downarrow_{\mathsf{exit}}}$$

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

Table 4. Matching rules / Is there an active receive along $\tilde{\mathsf{p}}$ and o matching $\bar{\mathsf{v}}$?

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

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

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

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

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

$$\mathsf{end}(\mathsf{sh}) = \mathsf{sh} \qquad \mathsf{end}(\langle\!|\mathsf{a}|\!\rangle) = \langle\!|\mathsf{a}|\!\rangle \qquad \mathsf{end}(\mathsf{a}_1\,;\mathsf{a}_2) = \mathsf{end}(\mathsf{a}_1)$$

$$\mathsf{end}([\mathsf{a} \bullet \mathsf{a}_f \star \mathsf{a}_c \vartriangle \mathsf{a}_d]) = [\mathsf{end}(\mathsf{a}) \bullet \mathsf{a}_f \star \mathsf{a}_c \vartriangle \mathsf{a}_d]$$

where a_1 may not be congruent to empty or to $\ll \tilde{\mathsf{p}}:\mathsf{o}:\bar{\mathsf{v}} \gg$, or to parallel compositions of them. In the remaining cases, end$(\cdot)$ returns stop, except for parallel composition for which it acts as an homomorphism.

We now briefly comment on the rules in Table 3. 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 $\langle\!|\mathsf{a}|\!\rangle$ 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_1) and (par_2)), except when a terminate/fault activity can be executed (rule (par_2)), in which case all parallel activities must immediately terminate except for short-lived activities and protected fault/compensation handlers. In other words, termination activities throw and exit are executed eagerly.

By rules (done_1) and (done_3), scope completions can be compensated according to the WS-BPEL *default* compensation behaviour (i.e. in the reverse order of completion) by the immediately enclosing scope. Notably, scopes like $[\mathsf{empty} \bullet \mathsf{a}_f \star \mathsf{a}_c \vartriangle \mathsf{a}_d]$ have not completed yet and when a scope completes, the default compensation a_d of inner scopes is not passed to the enclosing scope (rule (done_1)). Rule (exec) permits to

Table 5. Reduction rules for B*lite* deployments (where $t_1 = \ell^r : o : \bar{x}$ and $t_2 = \tilde{p} : o : \bar{v}$)

$$\frac{a_1 \xrightarrow{?t_1} a_1' \quad a_2 \xrightarrow{!t_2} a_2' \quad \mathsf{match}(c_1, \mu_1, t_1, t_2) = \mu_1' \quad \neg(\mu_1 \vdash a_1, s_1 \Downarrow_{t_2}^{c_1, |\mu_1'|})}{\{\mu_1 \vdash a_1, s_1\}_{c_1} \| \{\mu_2 \vdash a_2, s_2\}_{c_2} \rightarrowtriangle \{\mu_1 \circ \mu_1' \vdash a_1', s_1\}_{c_1} \| \{\mu_2 \vdash a_2', s_2\}_{c_2}} \text{ (com)}$$

$$\frac{[r \bullet a_f \star \mathsf{empty}] \xrightarrow{?t_1} a_1 \quad a_2 \xrightarrow{!t_2} a_2' \quad \mathsf{match}(c_1, \emptyset, t_1, t_2) = \mu_1 \quad \neg(s_1 \Downarrow_{t_2}^{c_1, |\mu_1|})}{\{[r \bullet a_f], s_1\}_{c_1} \| \{\mu_2 \vdash a_2, s_2\}_{c_2} \rightarrowtriangle \{\mu_1 \vdash a_1, [r \bullet a_f], s_1\}_{c_1} \| \{\mu_2 \vdash a_2', s_2\}_{c_2}} \text{ (new)}$$

$$\frac{\mu \vdash a \xrightarrow{x \leftarrow v} a' \quad \mathsf{match}(c, \mu, x, v) = \mu'}{\{\mu \vdash a, s\}_c \rightarrowtriangle \{\mu \circ \mu' \vdash a', s\}_c} \text{ (var)} \qquad \frac{d_1 \rightarrowtriangle d_1'}{d_1 \| d_2 \rightarrowtriangle d_1' \| d_2} \text{ (part)}$$

$$\frac{\mu \vdash a \xrightarrow{\alpha} a' \quad \alpha \notin \{?t_1, !t_2, x \leftarrow v\}}{\{\mu \vdash a, s\}_c \rightarrowtriangle \{\mu \vdash a', s\}_c} \text{ (pass)} \qquad \frac{d \equiv d_1 \quad d_1 \rightarrowtriangle d_2 \quad d_2 \equiv d'}{d \rightarrowtriangle d'} \text{ (cong)}$$

perform any action of the primary activity a except for fault emission and scope completion. In particular, inner forced terminations are propagated externally outside the scope. Differently from forced termination, faults arising within a scope are managed internally (rule (fault)), and the corresponding handler is installed when the main activity completes (rule (done_2)). By rule (done_2), default compensation is performed *after* termination of the primary activity and before fault handling. Note that compensation activities do not store any state with them: hence, if the state changes between the compensation being stored and executed, the current state is used.

A few auxiliary functions are also used in the semantics of deployments defined in Table 5. The rules for communication and updating of variables ((com), (new) and (var)) need a mechanism for checking if an assignment of some values $\bar{v}$ to $\bar{w}$ complies with the constraints imposed by the given correlation set c and state μ and, in case of success, returns a state μ' for the variables in $\bar{w}$ that records the effect of the assignment. This mechanism is implemented by the function $\mathsf{match}(\cdot, \cdot, \cdot, \cdot)$ defined through the rules in the upper part of Table 4. Notice that $\mathsf{match}(\cdot, \cdot, \cdot, \cdot)$ is undefined when $\bar{w}$ and $\bar{v}$ have different length or when $x \in c$ and $\{x \mapsto v'\} \in \mu$ for some $v' \neq v$ (since the state $\{x \mapsto v\}$ does not comply with c and μ). Rules (com) and (new) also use the auxiliary predicate $s \Downarrow_{\tilde{p}:o:\bar{v}}^{c,n}$, defined inductively on the syntax of s in the lower part of Table 4, that checks the ability of s of performing a receive on the operation o exploiting the partner link $\tilde{p}$, matching the tuple of values $\bar{v}$ and generating a state with fewer pairs than n that complies with c and the current state of the activity performing the receive.

Finally, we linger on the rules in Table 5. By rule (com), communication can take place when two service instances perform matching receive and invoke activities complying with the correlation set of the receiving instance. Notice that matching covers both 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 more 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 different service instances and to model the precedence of an existing service instance over a new service instantiation (rule (new), see also the *Multiple start and conflicting receive activities* example in Section 4). In rules (com) and (new), the assumption about *well-formedness* of deployments finds full employment, because it avoids to check every single deployment for possible conflicting receive activities. By rule (new), service instantiation can take place when a service definition and a service instance perform matching receive and invoke activities, respectively. By rule (var), correlation variables cannot be reassigned if the new value does not match with the old one. Moreover, if an assignment takes place, its effect is global to the instance, i.e. the state is updated. By rule (pass), 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 A Shipping Service Scenario

We consider an extended version of the shipping service described in the official specification of WS-BPEL [23] (Section 15.1). This example will allow us to illustrate most of the language features, including correlation sets, shared variables, flow control structures, fault and compensation handling. We will see that, in particular, scope activities are especially useful for modelling fault handling and compensation behaviours, while exit activities are useful to exit from while loops and terminate the customer instance.

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 shipment: shipments where the items are held and shipped together and shipments where the items are shipped piecemeal until the order is fulfilled. The service specification in B*lite* is

$$\begin{array}{l} \mathsf{s}_{ship} \triangleq [\ \mathsf{rcv}\,\langle \mathsf{p}_{ship}, \mathsf{x}_{cust}\rangle\ \mathsf{o}_{req}\,\langle \mathsf{x}_{id}, \mathsf{x}_c, \mathsf{x}_{items}\rangle\ ; \\ \quad \mathsf{if}\,(\mathsf{x}_c)\,\{\,\mathsf{inv}\,\langle \mathsf{x}_{cust}\rangle\ \mathsf{o}_{notice}\,\langle \mathsf{x}_{id}, \mathsf{x}_{items}\rangle\,\}\,\{\,[\ \mathsf{a}_{ship} \bullet \mathsf{inv}\,\langle \mathsf{x}_{cust}\rangle\ \mathsf{o}_{err}\,\langle \mathsf{x}_{id}, \text{“}\mathit{sorry}\text{”}\rangle\ ]\,\}\ ] \end{array}$$

$$\begin{array}{l} \mathsf{a}_{ship} \triangleq [\ \mathsf{a}_{priceCalc} \star \mathsf{a}_{comp}\ ]\ ;\ \mathsf{x}_{shipped} := 0\ ; \\ \quad \mathsf{while}\,(\mathsf{x}_{shipped} < \mathsf{x}_{items})\,\{ \\ \quad\quad \mathsf{x}_{count} := \mathit{rand}()\ ; \\ \quad\quad \mathsf{if}\,(\mathsf{x}_{count} \leqslant 0)\,\{\,\mathsf{x}_{ratio} := \mathsf{x}_{shipped} \,/\, \mathsf{x}_{items}\ ;\ \mathsf{throw}\,\} \\ \quad\quad\quad\quad\quad \{\,\mathsf{inv}\,\langle \mathsf{x}_{cust}\rangle\ \mathsf{o}_{notice}\,\langle \mathsf{x}_{id}, \mathsf{x}_{count}\rangle\ ;\ \mathsf{x}_{shipped} := \mathsf{x}_{shipped} + \mathsf{x}_{count}\,\}\,\} \end{array}$$

p_{ship} is the partner associated to the shipping service, o_{req} is the operation used to receive the shipping request, and $\langle \mathsf{x}_{id}, \mathsf{x}_c, \mathsf{x}_{items}\rangle$ is the tuple of variables used for the request shipping message: x_{id} stores the order identifier, that is used to correlate the ship notice(s) with the ship order, x_c stores a boolean indicating whether the order is to be shipped complete or not, and x_{items} stores the number of items in the order. Shipping notices and error messages to customers are sent using the partner stored in x_{cust} and the operations o_{notice} and o_{err}, respectively. A notice message is a tuple composed of the order identifier and the number of items in the shipping notice. When partial shipment is acceptable, $\mathsf{x}_{shipped}$ is used to record the number of items already shipped.

Our example extends that in [23] by allowing the service to generate a fault in case the shipping company has ended the stock of items (this is modelled by function *rand*()

returning an integer less than or equal to 0). The fault is handled by sending an error message to the customer and by compensating the inner scope, that has already completed successfully. Function *rand*() returns a random integer number and represents an internal interaction with a back-end system. For the sake of simplicity, we do not further describe this interaction. Moreover, we do not show services $\mathsf{a}_{priceCalc}$ and a_{comp}. Basically, the former calculates the shipping price according to the value assigned to x_{items} and sends the result to the accounts department. The latter is the corresponding compensation activity, that sends information about the non-shipped items to the accounts department and sends a refund to the customer according to the ratio (stored in x_{ratio}) between the shipped items (stored in $\mathsf{x}_{shipped}$) and the required ones (stored in x_{items}). Now, consider the following composition of a deployment containing the shipping service definition and a deployment containing a customer's invocation of the service

$$\{\,\mathsf{s}_{ship}\,\}_{\{\mathsf{x}_{id}\}} \| \{\mu_{cust} \vdash \mathsf{inv}\langle \mathsf{p}_{ship}, \mathsf{p}_{cust}\rangle\, \mathsf{o}_{req}\langle \mathsf{y}_{id}, \mathsf{y}_c, \mathsf{y}_{items}\rangle \;;\; \mathsf{a}_{cust}\}_{\{\mathsf{y}_{id}\}}$$

where $\mu_{cust} = \{\mathsf{y}_{id} \mapsto 123, \mathsf{y}_c \mapsto \mathsf{ff}, \mathsf{y}_{items} \mapsto 50\}$ and a_{cust} is the following term

$$\begin{array}{l} \mathsf{y}_{shipped} := 0 \;;\; \mathsf{while}\,(\mathsf{y}_{shipped} < \mathsf{y}_{items})\,\{ \\ \qquad \mathsf{rcv}\langle \mathsf{p}_{cust}\rangle\, \mathsf{o}_{notice}\langle \mathsf{y}_{id}, \mathsf{y}_{count}\rangle \;;\; \mathsf{y}_{shipped} := \mathsf{y}_{shipped} + \mathsf{y}_{count} \\ \qquad + \mathsf{rcv}\langle \mathsf{p}_{cust}\rangle\, \mathsf{o}_{err}\langle \mathsf{y}_{id}, \mathsf{y}_{err}\rangle \;;\; \mathsf{exit}\,\} \end{array}$$

In the first computational step, the customer's invocation is consumed and an instance of the shipping service is created. Thus the overall computation becomes

$$\{\,\mathsf{s}_{ship}\,,\, \mu_{ship} \vdash [\,\mathsf{a}_{ship} \bullet \mathsf{inv}\langle \mathsf{x}_{cust}\rangle\, \mathsf{o}_{err}\langle \mathsf{x}_{id}, \text{“}sorry\text{”}\rangle\,]\,\}_{\{\mathsf{x}_{id}\}} \| \{\mu_{cust} \vdash \mathsf{a}_{cust}\}_{\{\mathsf{y}_{id}\}}$$

where μ_{ship} is $\{\mathsf{x}_{id} \mapsto 123, \mathsf{x}_c \mapsto \mathsf{ff}, \mathsf{x}_{items} \mapsto 50, \mathsf{x}_{cust} \mapsto \mathsf{p}_{cust}\}$. The computation can now go on, e.g., with the inner scope $[\mathsf{a}_{priceCalc} \star \mathsf{a}_{comp}]$ that successfully completes while its continuation fails, e.g., *rand*() returns an integer less than or equal to 0.

4 Evaluations of BPEL Engines

We now present some illustrative examples and use them to test and compare the behaviour of three well-known free WS-BPEL engines, namely Oracle BPEL Process Manager [3], ActiveBPEL Engine [1], and Apache ODE [2] (the latter two are open source projects, whereas Oracle BPEL is distributed under the Oracle Technology Network Developer License). For our evaluation, we have taken into account fundamental features of WS-BPEL that remained unchanged since its initial version. Due to lack of space, we refer the interested reader to [18] for further details and examples.

Example 1 (Message correlation). For our simplification purposes, tuples can be used to represent XML messages by adopting the convention that the first field of each tuple acts as a 'tag' (like originally proposed in the coordination language Linda [12]). Tuples

plus 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 two uncorrelated receive activities of the following service definition:

$$\{\,[\mathsf{rcv}\,\langle \mathsf{p}\rangle\,\mathsf{o}\,\langle \mathsf{x}\rangle\,;\,\mathsf{rcv}\,\langle \mathsf{p}\rangle\,\mathsf{o}'\,\langle \mathsf{y}\rangle\,;\,\mathsf{a}]\,\}_{\{\mathsf{x},\mathsf{y}\}}$$

The fact that the messages for operations o and o' are uncorrelated implies that, e.g., if there are concurrent instances then successive invocations for the same instance can be mixed up and be delivered to different instances. If one thinks it right, this behavior can be prevented simply by correlating consecutive messages by means of some correlation data, e.g. the first received value as in the following modified service definition:

$$\{\,[\mathsf{rcv}\,\langle \mathsf{p}\rangle\,\mathsf{o}\,\langle \mathsf{x}\rangle\,;\,\mathsf{rcv}\,\langle \mathsf{p}\rangle\,\mathsf{o}'\,\langle \mathsf{x},\mathsf{y}\rangle\,;\,\mathsf{a}]\,\}_{\{\mathsf{x},\mathsf{y}\}}$$

A particular case is when the two previous receives are identical, i.e. when we have:

$$\{\,[\mathsf{rcv}\,\langle \mathsf{p}\rangle\,\mathsf{o}\,\langle \mathsf{x}\rangle\,;\,\mathsf{rcv}\,\langle \mathsf{p}\rangle\,\mathsf{o}\,\langle \mathsf{x}\rangle\,;\,\mathsf{a}]\,\}_{\{\mathsf{x}\}}$$

Note that the WS-BPEL specification permits to consecutively receive a same request on a specific partner and operation ([23], Section 10.4), and does not mention that possible conflicting receives could arise. To illustrate, include a client process as follows:

$$\{\,[\mathsf{rcv}\,\langle \mathsf{p}\rangle\,\mathsf{o}\,\langle \mathsf{x}\rangle\,;\,\mathsf{rcv}\,\langle \mathsf{p}\rangle\,\mathsf{o}\,\langle \mathsf{x}\rangle\,;\,\mathsf{a}]\,\}_{\{\mathsf{x}\}} \,\|\, \{\,\{\mathsf{y}\mapsto \mathsf{v}\}\vdash \mathsf{inv}\,\langle \mathsf{p}\rangle\,\mathsf{o}\,\langle \mathsf{y}\rangle\,;\,\mathsf{inv}\,\langle \mathsf{p}\rangle\,\mathsf{o}\,\langle \mathsf{y}\rangle\,\}_{\emptyset}$$

The client process performs two requests that, according to the semantics of B*lite*, trigger only one instantiation of the service. Thus, the only possible evolution leads to

$$\{\,[\mathsf{rcv}\,\langle \mathsf{p}\rangle\,\mathsf{o}\,\langle \mathsf{x}\rangle\,;\,\mathsf{rcv}\,\langle \mathsf{p}\rangle\,\mathsf{o}\,\langle \mathsf{x}\rangle\,;\,\mathsf{a}]\,,\,\{\mathsf{x}\mapsto \mathsf{v}\}\vdash [\mathsf{a}]\,\}_{\{\mathsf{x}\}}$$

Differently from B*lite*, when executing this example, Oracle BPEL creates two instances, one for each received request. An important consequence, and an unexpected side effect, is that the created instances are in conflict and, then, will never be executed. Instead, ActiveBPEL and Apache ODE, just like B*lite*, exploit the received data to correlate two consecutive receives and, thus, prevent creation of a new instance. However, if the client performs a third invocation $\mathsf{inv}\,\langle \mathsf{p}\rangle\,\mathsf{o}\,\langle \mathsf{y}\rangle$, Apache ODE is not able to serve this last request, while ActiveBPEL behaves properly.

Example 2 (Persistent messages). In service-oriented systems communication paradigms are usually asynchronous (mainly for scalability reasons [5]), 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 produced, and a sender cannot determine if and when a sent message will be received. We can guess from [23], Section 10.4, that this is also the case of WS-BPEL. To illustrate, consider the following B*lite* term:

$$\begin{array}{l}\{\,[\mathsf{rcv}\,\langle \mathsf{p}\rangle\,\mathsf{o}_1\,\langle \mathsf{x}\rangle\,;\,\mathsf{rcv}\,\langle \mathsf{p}\rangle\,\mathsf{o}_2\,\langle \mathsf{x},\mathsf{z}\rangle\,;\,\mathsf{a}]\,\}_{\{\mathsf{x}\}}\\ \|\,\{\,\{\mathsf{y}_1\mapsto \mathsf{v},\mathsf{y}_2\mapsto \mathsf{v}'\}\vdash \mathsf{inv}\,\langle \mathsf{p}\rangle\,\mathsf{o}_2\,\langle \mathsf{y}_1,\mathsf{y}_2\rangle\,;\,\mathsf{inv}\,\langle \mathsf{p}\rangle\,\mathsf{o}_1\,\langle \mathsf{y}_1\rangle\,\}_{\emptyset}\end{array}$$

After the message $\ll \langle \mathsf{p}\rangle : \mathsf{o}_2 : \langle \mathsf{v},\mathsf{v}'\rangle \gg$ is produced by the first invoke activity, a service instance is created as a result of consumption of the message produced by the second invoke activity.

$$\{\,[\mathsf{rcv}\,\langle \mathsf{p}\rangle\,\mathsf{o}_1\,\langle \mathsf{x}\rangle\,;\,\mathsf{rcv}\,\langle \mathsf{p}\rangle\,\mathsf{o}_2\,\langle \mathsf{x},\mathsf{z}\rangle\,;\,\mathsf{a}]\,\}_{\{\mathsf{x}\}}$$
$$\|\,\{\,\{\mathsf{y}_1 \mapsto \mathsf{v}, \mathsf{y}_2 \mapsto \mathsf{v}'\} \vdash \ll\langle \mathsf{p}\rangle : \mathsf{o}_2 : \langle \mathsf{v},\mathsf{v}'\rangle \gg \;\mid \mathsf{inv}\,\langle \mathsf{p}\rangle\,\mathsf{o}_1\,\langle \mathsf{y}_1\rangle\,\}_{\emptyset} \rightarrowtail$$
$$\{\,[\mathsf{rcv}\,\langle \mathsf{p}\rangle\,\mathsf{o}_1\,\langle \mathsf{x}\rangle\,;\,\mathsf{rcv}\,\langle \mathsf{p}\rangle\,\mathsf{o}_2\,\langle \mathsf{x},\mathsf{z}\rangle\,;\,\mathsf{a}], \{\mathsf{x} \mapsto \mathsf{v}\} \vdash [\mathsf{rcv}\langle \mathsf{p}\rangle\,\mathsf{o}_2\,\langle \mathsf{x},\mathsf{z}\rangle\,;\,\mathsf{a}]\,\}_{\{\mathsf{x}\}}$$
$$\|\,\{\,\{\mathsf{y}_1 \mapsto \mathsf{v}, \mathsf{y}_2 \mapsto \mathsf{v}'\} \vdash \ll\langle \mathsf{p}\rangle : \mathsf{o}_2 : \langle \mathsf{v},\mathsf{v}'\rangle \gg\,\}_{\emptyset}$$

Now, the first produced message is not considered expired and, thus, can be consumed by the newly created service instance.

$$\{\,[\mathsf{rcv}\,\langle \mathsf{p}\rangle\,\mathsf{o}_1\,\langle \mathsf{x}\rangle\,;\,\mathsf{rcv}\,\langle \mathsf{p}\rangle\,\mathsf{o}_2\,\langle \mathsf{x},\mathsf{z}\rangle\,;\,\mathsf{a}], \{\mathsf{x} \mapsto \mathsf{v}, \mathsf{z} \mapsto \mathsf{v}'\} \vdash [\mathsf{a}]\,\}_{\{\mathsf{x}\}}$$
$$\|\,\{\,\{\mathsf{y}_1 \mapsto \mathsf{v}, \mathsf{y}_2 \mapsto \mathsf{v}'\} \vdash \mathsf{empty}\,\}_{\emptyset}$$

All the examined BPEL engines 'tacitly' agree with this communication paradigm, although no explicit requirement is reported in the WS-BPEL specification.

Example 3 (Multiple start and conflicting receive activities). The WS-BPEL specification permits to use multiple start activities ([23], Section 10.4), however it is not clear how conflicting receive activities must be handled. The following example shows that conflicting receive activities can be enabled when a service definition with multiple start activities is instantiated. Consider the three composed deployments

$$\{\,[\,(\mathsf{rcv}\,\langle \mathsf{p}_1\rangle\,\mathsf{o}\,\langle \mathsf{x}\rangle \mid \mathsf{rcv}\,\langle \mathsf{p}_2\rangle\,\mathsf{o}\,\langle \mathsf{x},\mathsf{z}\rangle\,)\,;\mathsf{a}]\,\}_{\{\mathsf{x}\}} \| \{\,\{\mathsf{y} \mapsto \mathsf{v}\} \vdash \mathsf{inv}\,\langle \mathsf{p}_1\rangle\,\mathsf{o}\,\langle \mathsf{y}\rangle\,\}_{\emptyset}$$
$$\|\,\{\,\{\mathsf{y}_1 \mapsto \mathsf{v}, \mathsf{y}_2 \mapsto \mathsf{v}'\} \vdash \mathsf{inv}\,\langle \mathsf{p}_2\rangle\,\mathsf{o}\,\langle \mathsf{y}_1,\mathsf{y}_2\rangle\,\}_{\emptyset}$$

After message $\ll\langle \mathsf{p}_1\rangle : \mathsf{o} : \langle \mathsf{v}\rangle \gg$, produced by invocation $\mathsf{inv}\,\langle \mathsf{p}_1\rangle\,\mathsf{o}\,\langle \mathsf{y}\rangle$, has been processed by $\mathsf{rcv}\,\langle \mathsf{p}_1\rangle\,\mathsf{o}\,\langle \mathsf{x}\rangle$, the overall composition becomes

$$\{\,[\,(\mathsf{rcv}\,\langle \mathsf{p}_1\rangle\,\mathsf{o}\,\langle \mathsf{x}\rangle \mid \mathsf{rcv}\,\langle \mathsf{p}_2\rangle\,\mathsf{o}\,\langle \mathsf{x},\mathsf{z}\rangle\,)\,;\mathsf{a}]\,,\,\{\mathsf{x} \mapsto \mathsf{v}\} \vdash [\mathsf{rcv}\,\langle \mathsf{p}_2\rangle\,\mathsf{o}\,\langle \mathsf{x},\mathsf{z}\rangle\,;\mathsf{a}]\,\}_{\{\mathsf{x}\}}$$
$$\|\,\{\,\{\mathsf{y}_1 \mapsto \mathsf{v}, \mathsf{y}_2 \mapsto \mathsf{v}'\} \vdash \mathsf{inv}\,\langle \mathsf{p}_2\rangle\,\mathsf{o}\,\langle \mathsf{y}_1,\mathsf{y}_2\rangle\,\}_{\emptyset}$$

Now, the definition and the instance of the service compete for receiving the same message sent by the invoke activity $\mathsf{inv}\,\langle \mathsf{p}_2\rangle\,\mathsf{o}\,\langle \mathsf{y}_1,\mathsf{y}_2\rangle$. In cases like this, the WS-BPEL specification requires that the invocation is only delivered to the existing instance, which prevents creation of a new instance. In fact, in B*lite* the above term can only reduce to

$$\{\,[\,(\mathsf{rcv}\,\langle \mathsf{p}_1\rangle\,\mathsf{o}\,\langle \mathsf{x}\rangle \mid \mathsf{rcv}\,\langle \mathsf{p}_2\rangle\,\mathsf{o}\,\langle \mathsf{x},\mathsf{z}\rangle\,)\,;\mathsf{a}]\,,\,\{\mathsf{x} \mapsto \mathsf{v}, \mathsf{z} \mapsto \mathsf{v}'\} \vdash [\mathsf{a}]\,\}_{\{\mathsf{x}\}}$$

In case of conflicting receives, the WS-BPEL specification document prescribes to raise the standard fault `bpel:conflictingReceive`. For example, this situation readily occurs when a service exploits multiple start activities, because of race conditions on incoming messages among the service definition and the created instances. However, in such cases, it does not seem fair to raise a fault because the correlation data contained within each incoming message should be sufficient to decide if the message has to be routed to a specific instance or to the service definition. This is indeed a tricky question. For example, Oracle BPEL raises the fault `bpel:conflictingReceive` also in these situations. ActiveBPEL behaves differently and, just like B*lite*, exploits correlation to restrict instantiation to one service instance, whereas multiple start activities are not currently supported by Apache ODE.

Example 4 (Scheduling for parallel execution). While using the BPEL engines, we have also experimented that they implement the parallel operator in a different manner. For example, in WS-BPEL, the expected behaviour of the following term:

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

is that the three assignments 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 from left to right in case of ActiveBPEL and from right to left in case of Oracle BPEL).

Example 5 (Forced termination). The WS-BPEL specification ([23], Section 12.6) says: "The `<sequence>` and `<flow>` constructs *must* be terminated by terminating their behavior and applying termination to all nested activities currently active within them". This definition 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, Oracle BPEL interprets the behaviour $\mathsf{end}(\mathsf{a}_1 \,;\, \mathsf{a}_2)$ as it were $\mathsf{a}_1 \,;\, \mathsf{a}_2$ if it is prompted by activity `<exit>`, and as $\mathsf{end}(\mathsf{a}_1)$, if it is prompted by activity `<throw>`. ActiveBPEL is more faithful to WS-BPEL and B*lite* for which all currently running activities are terminated as soon as possible without any fault handling or compensation ([23], Section 10.10). But, differently from B*lite*, ActiveBPEL does not distinguish short-lived from basic activities and makes them terminate in the same way. Finally, Apache ODE is compliant with B*lite*, because function $\mathsf{end}(\cdot)$ retains short-lived activities.

Example 6 (Eager execution of termination activities). As previously stated, in order to be compliant with the WS-BPEL requirement stating that termination activities must end immediately all currently running activities ([23], Section 10.10), in the semantics of B*lite* activities throw and exit have higher priority than the remaining ones. E.g., consider the following structured activity:

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

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

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

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

Example 7 (Protected handlers). The following structured activity consists of a top-level scope with two inner parallel scopes, one of which being a sequence of two scopes.

$$\begin{aligned}\mathsf{a} \triangleq [\,(\,[\,[\,\mathsf{a}_1 \bullet \mathsf{throw} \star \mathsf{a}_c\,]\,;[\,\mathsf{a}_2 \bullet \mathsf{throw} \star \mathsf{empty}\,] \bullet \mathsf{throw} \star \mathsf{empty}\,] \\ \mid [\,\mathsf{a}_3 \bullet \mathsf{throw} \star \mathsf{empty}\,]\,) \bullet \mathsf{empty}\,]\end{aligned}$$

For the sake of presentation, suppose that a_1 performs an assignment and completes, say $\mathsf{a}_1 \triangleq \mathsf{x}_1 := \mathsf{v}_1$, while both activities a_2 and a_3 perform an assignment and reduce to the throw activity, say $\mathsf{a}_i \triangleq \mathsf{x}_i := \mathsf{v}_i \,;\, \mathsf{throw}$ for $i = 2, 3$. Now, consider a deployment containing a service instance $\mu \vdash \mathsf{a}$ such that variables x_1, x_2 and x_3 are not in $dom(\mu)$. A possible computation is the following one

$$
\begin{aligned}
\{\mu \vdash \mathsf{a}\}_{\mathsf{c}} &\xrightarrow{(1)} \{\mu_1 \vdash [\,(\,[\,[\,\mathsf{empty} \bullet \mathsf{throw} \star \mathsf{a}_c\,]\,; \\
&\qquad [\,\mathsf{a}_2 \bullet \mathsf{throw} \star \mathsf{empty}\,] \bullet \mathsf{throw} \star \mathsf{empty} \triangle \mathsf{empty}\,] \\
&\qquad \mid [\,\mathsf{a}_3 \bullet \mathsf{throw} \star \mathsf{empty}\,]\,) \bullet \mathsf{empty} \star \mathsf{empty}\,]\}_{\mathsf{c}} \\
&\xrightarrow{(2)} \{\mu_1 \vdash [\,(\,[\,[\,\mathsf{a}_2 \bullet \mathsf{throw} \star \mathsf{empty}\,] \bullet \mathsf{throw} \star \mathsf{empty} \triangle (\mathsf{a}_c\,;\mathsf{empty})\,] \\
&\qquad \mid [\,\mathsf{a}_3 \bullet \mathsf{throw} \star \mathsf{empty}\,]\,) \bullet \mathsf{empty} \star \mathsf{empty}\,]\}_{\mathsf{c}} \\
&\xrightarrow{(3)} \{\mu_2 \vdash [\,(\,[\,[\,\mathsf{throw} \bullet \mathsf{throw} \star \mathsf{empty}\,] \bullet \mathsf{throw} \star \mathsf{empty} \triangle \mathsf{a}_c\,] \\
&\qquad \mid [\,\mathsf{a}_3 \bullet \mathsf{throw} \star \mathsf{empty}\,]\,) \bullet \mathsf{empty} \star \mathsf{empty}\,]\}_{\mathsf{c}} \\
&\xrightarrow{(4)} \{\mu_2 \vdash [\,(\,[\,[\,\mathsf{stop} \bullet \mathsf{throw} \star \mathsf{empty}\,] \bullet \mathsf{throw} \star \mathsf{empty} \triangle \mathsf{a}_c\,] \\
&\qquad \mid [\,\mathsf{a}_3 \bullet \mathsf{throw} \star \mathsf{empty}\,]\,) \bullet \mathsf{empty} \star \mathsf{empty}\,]\}_{\mathsf{c}} \\
&\xrightarrow{(5)} \{\mu_2 \vdash [\,(\,[\,(\!|\mathsf{throw}|\!) \bullet \mathsf{throw} \star \mathsf{empty} \triangle \mathsf{a}_c\,] \\
&\qquad \mid [\,\mathsf{a}_3 \bullet \mathsf{throw} \star \mathsf{empty}\,]\,) \bullet \mathsf{empty} \star \mathsf{empty}\,]\}_{\mathsf{c}} \\
&\xrightarrow{(6)} \{\mu_2 \vdash [\,(\,[\,(\!|\mathsf{stop}|\!) \bullet \mathsf{throw} \star \mathsf{empty} \triangle \mathsf{a}_c\,] \\
&\qquad \mid [\,\mathsf{a}_3 \bullet \mathsf{throw} \star \mathsf{empty}\,]\,) \bullet \mathsf{empty} \star \mathsf{empty}\,]\}_{\mathsf{c}} \\
&\xrightarrow{(7)} \{\mu_2 \vdash [\,(\,(\!|\mathsf{a}_c\,;\mathsf{throw}|\!)\,] \mid [\,\mathsf{a}_3 \bullet \mathsf{throw} \star \mathsf{empty}\,]\,) \bullet \mathsf{empty} \star \mathsf{empty}\,]\}_{\mathsf{c}} \\
&\xrightarrow{(8)} \{\mu_3 \vdash [\,(\,(\!|\mathsf{a}_c\,;\mathsf{throw}|\!)\,] \mid [\,\mathsf{throw} \bullet \mathsf{throw} \star \mathsf{empty}\,]\,) \bullet \mathsf{empty} \star \mathsf{empty}\,]\}_{\mathsf{c}} \\
&\xrightarrow{(9)} \{\mu_3 \vdash [\,(\,(\!|\mathsf{a}_c\,;\mathsf{throw}|\!)\,] \mid [\,\mathsf{stop} \bullet \mathsf{throw} \star \mathsf{empty}\,]\,) \bullet \mathsf{empty} \star \mathsf{empty}\,]\}_{\mathsf{c}} \\
&\xrightarrow{(10)} \{\mu_3 \vdash [\,(\,(\!|\mathsf{a}_c\,;\mathsf{throw}|\!)\,] \mid (\!|\mathsf{throw}|\!)\,) \bullet \mathsf{empty} \star \mathsf{empty}\,]\}_{\mathsf{c}} \\
&\xrightarrow{(11)} \{\mu_3 \vdash [\,(\,\mathsf{end}(\,(\!|\mathsf{a}_c\,;\mathsf{throw}|\!)\,) \mid (\!|\mathsf{stop}|\!)\,) \bullet \mathsf{empty} \star \mathsf{empty}\,]\}_{\mathsf{c}} \\
&\equiv \{\mu_3 \vdash [\,(\,(\!|\mathsf{a}_c\,;\mathsf{throw}|\!)\,] \mid \mathsf{stop}\,) \bullet \mathsf{empty} \star \mathsf{empty}\,]\}_{\mathsf{c}}
\end{aligned}
$$

where the reductions are labelled by numbers indicating the corresponding steps. When a_1 completes, the compensation handler a_c is inserted into the default compensation activities of its enclosing scope (1-2). When execution of a_2 rises a fault, then the fault is caught by the corresponding fault handler (3-7) that activates the default compensation $\mathsf{a}_c\,;\mathsf{throw}$. This activity is protected, by using the auxiliary operator $(\!|\cdot|\!)$, from the effect of the forced termination triggered by the parallel scope $[\,\mathsf{a}_3 \bullet \mathsf{throw} \star \mathsf{empty}\,]$ (7-11).

We end by remarking two aspects of the compensation mechanism prescribed by the WS-BPEL specification ([23], Sections 12.5 and 10.10). First, compensation handlers of faultily terminated scopes should not be installed. Second, fault and compensation handlers should not be affected by the activities causing the forced termination. However, both aspects are not faithfully implemented in Oracle BPEL, while ActiveBPEL and Apache ODE meet these specific requirements and adhere to B*lite* semantics.

Evaluation Results. The results of our experiments, summarized in Table 6, point out that the engines we have experimented with are not fully compliant with B*lite*, that, in our opinion, faithfully represents the intended semantics of WS-BPEL. This is also a consequence of the lack of a formal semantics for WS-BPEL, that would have disambiguated the intricate and complex features of the language. We believe that B*lite*, and

Table 6. B*lite* compliance

	Oracle BPEL	ActiveBPEL	Apache ODE
Message correlation (Ex. 1)	+	+	+
Consecutive conflicting receives (Ex. 1)	−	+	+/−
Persistent messages (Ex. 2)	+	+	+
Multiple start (Ex. 3)	−	+	−
Parallel execution (Ex. 4)	−	−	+
Short-lived activities (Ex. 5)	+	−	+
Function $\mathsf{end}(\cdot)$ (Ex. 5)	−	+	+
Eager execution (Ex. 6)	−	+	−
Protected handlers (Ex. 7)	−	+	+
Compensation handler installation (Ex. 7)	−	+	+

works with similar goals, other than as a guide for the development of faithful implementations since the early stages, can be also used to make future versions of existing implementations more compatible.

5 Concluding Remarks

We have introduced B*lite*, a significative and non-redundant fragment of WS-BPEL, designed around some of its peculiar features like partner links, process termination, message correlation, long-running business transactions and compensation handlers. Our formal presentation of B*lite* helps clarifying some undefined/ambiguous aspects of the WS-BPEL specification. For example, we have formalized the close relationship between multiple start activities and race conditions. By means of several examples, we have also pointed out that the behaviour of three of the most used free BPEL engines (namely, ActiveBPEL, Apache ODE and Oracle BPEL Process Manager) differs from each other and from the WS-BPEL specification in many important aspects.

Several formal semantics of WS-BPEL were proposed in the literature (for an overview see [24]). Many of these efforts aim at formalizing a *complete* semantics for WS-BPEL using Petri nets [24,19], but do not cover such dynamical aspects as service instantiation and message correlation. Other works [11,14] using process calculi focus instead on small and relatively simple subsets of WS-BPEL. Another bunch of related works [15,20] formalize the semantics of WS-BPEL by encoding parts of the language into more foundational orchestration languages. Our work differs for the number of features that are simultaneously modelled and for the fact that dynamical aspects are fully taken into account. Recently, a very general and flexible framework for error recovery has been introduced in [13]; this framework extends [14] with dynamic compensation, modelling in particular the dependency between fault handling and the request-response communication pattern.

Some other relevant related works are [7,6,4]. In the first two, the authors propose a formal approach to model compensation in transactional calculi and present a detailed comparison with [8]. The third is an extension of the asynchronous π-calculus with long-running (scoped) transactions. The language has a scope construct which plays

a role similar to the scope activity presented in our semantics, but it is not aimed at capturing the order in which compensations should be activated. On the contrary, the semantics we propose faithfully captures the intended semantics of WS-BPEL, thus for example compensations are activated in the reverse order w.r.t. the order of completion of the original scopes.

Our programme is to provide a framework for the design and the verification of WS-BPEL applications that supports analysis of service orchestration. As a further step in this direction, in [18] we have also defined an encoding from B*lite* to COWS [16], a calculus for orchestration of web services that we recently proposed, and we have formalized the properties enjoyed by the encoding. By relying on these results, we plan to devise methods to analyze B*lite* specifications (and the WS-BPEL applications they model) by exploiting the analytical tools already developed for COWS, such as the stochastic extension defined in [25] that enables quantitative reasoning on service behaviours, the type system introduced in [17] that permits to check confidentiality properties, and the logic and model checker presented in [10] that permits expressing and checking functional properties of services.

Acknowledgements. We thank the anonymous referees for their useful comments.

References

1. ActiveBPEL 4.1 (September 2007), `http://www.active-endpoints.com`
2. Apache ODE 1.1.1 (August 2007), `http://ode.apache.org`
3. Oracle BPEL Process Manager 10.1.3 (December 2007), `http://www.oracle.com/technology/bpel`
4. Bocchi, L., Laneve, C., Zavattaro, G.: A calculus for long-running transactions. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 124–138. Springer, Heidelberg (2003)
5. Brown, A., Johnston, S., Kelly, K.: Using service-oriented architecture and component-based development to build web service applications, TR, Rational Software Corp. (2002)
6. Bruni, R., Butler, M.J., Ferreira, C., Hoare, C.A.R., Melgratti, H.C., Montanari, U.: Comparing two approaches to compensable flow composition. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 383–397. Springer, Heidelberg (2005)
7. Bruni, R., Melgratti, H.C., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: POPL, pp. 209–220. ACM, New York (2005)
8. Butler, M.J., Ferreira, C.: An operational semantics for StAC, a language for modelling long-running business transactions. In: De Nicola, R., Ferrari, G.L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 87–104. Springer, Heidelberg (2004)
9. Box, D.: et al. Web services addressing. W3C member submission, August 10 (2004)
10. Fantechi, A., Gnesi, S., Lapadula, A., Mazzanti, F., Pugliese, R., Tiezzi, F.: A model checking approach for verifying COWS specifications. In: FASE. LNCS, Springer, Heidelberg (to appear, 2008)
11. Geguang, P., Xiangpeng, Z., Shuling, W., Zongyan, Q.: Semantics of BPEL4WS-like fault and compensation handling. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 350–365. Springer, Heidelberg (2005)
12. Gelernter, D.: Generative communication in Linda. ACM TOPLAS 7(1), 80–112 (1985)

13. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: On the interplay between fault handling and request-response service invocations. In: ACSD, IEEE CS Press, Los Alamitos (to appear, 2008)
14. Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: SOCK: a calculus for service oriented computing. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 327–338. Springer, Heidelberg (2006)
15. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 282–298. Springer, Heidelberg (2005)
16. Lapadula, A., Pugliese, R., Tiezzi, F.: A Calculus for Orchestration of Web Services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 33–47. Springer, Heidelberg (2007)
17. Lapadula, A., Pugliese, R., Tiezzi, F.: Regulating data exchange in service oriented applications. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 223–239. Springer, Heidelberg (2007)
18. Lapadula, A., Pugliese, R., Tiezzi, F.: A formal account of WS-BPEL (full version), Technical report, Univ. Firenze (2008), http://rap.dsi.unifi.it/cows
19. Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0. In: Web Services and Formal Methods. LNCS, vol. 4937, pp. 77–91. Springer, Heidelberg (2008)
20. Mazzara, M., Lucchi, R.: A pi-calculus based semantics for WS-BPEL. Journal of Logic and Algebraic Programming 70(1), 96–118 (2006)
21. Meredith, L.G., Bjorg, S.: Contracts and types. Commun. ACM 46(10), 41–47 (2003)
22. OASIS WSBPEL TC. WS-BPEL issues list, http://www.oasis-open.org/committees/download.php/20228/WS_BPEL_issues_list.html
23. OASIS WSBPEL TC. Web Services Business Process Execution Language Version 2.0 (April 2007), http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
24. Ouyang, C., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M., Verbeek, H.M.W.: Formal semantics and analysis of control flow in WS-BPEL (revised version). Technical report, BPM Center Report (2005), http://www.BPMcenter.org
25. Prandi, D., Quaglia, P.: Stochastic COWS. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 245–256. Springer, Heidelberg (2007)
26. van Breugel, F., Koshkina, M.: Models and verification of BPEL. Technical report (2006), http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf