Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 200 (2008) 43–65
www.elsevier.com/locate/entcs

A Calculus for Generation, Verification and Refinement of BPEL Specifications [1]

Faisal Abouzaid and John Mullins

CRAC Lab., Computer & Software Eng. Dept., École Polytechnique de Montréal. P.O. Box 6079, Station Centre-ville, Montreal (Quebec), Canada, H3C 3P8. [2]

Abstract

Business Process Execution Language for Web Services (WS-BPEL) is the emerging standard for designing Web Services compositions. In this context, formal methods can contribute to increased reliability and consistency in the BPEL design process. In this paper we propose an approach based on the HAL Toolkit that allows verification of the correctness of the behavior of a π-based specification of interacting Web Services, and generates the BPEL processes that have the same behavior. This correlation based on two-way mapping between the π-based orchestration calculus and BPEL. This approach facilitates the verification and refinement process and may be applied to any BPEL implementation.

Keywords: Web services, orchestration, BPEL, formal methods, pi-calculus.

1 Introduction

BPEL [11] is the language used to express Web Services (WS) orchestrations which has been accepted as the standard since April 2007. It expresses the execution logic of a business process based on interactions between the process and its partners. A BPEL process defines how multiple service interactions between partners can be coordinated internally in order to achieve a business goal (orchestration).

In order to increase reliability and consistency in a BPEL design process, we propose a formal framework that also allows for the integration of a well-established general purpose model-checker toolkit and for the generation, from the model-checkers's modelling language of a BPEL process that has the same behavior as the verified abstract model. We first need an abstract modelling language which reflects, as far as possible, the intentions of orchestration conductors and workflow constructs. For this purpose we present a new specification language based on the π-calculus

[1] Research partially supported by the second author's individual NSERC grant (Government of Canada).
[2] Email: mohamed-faical.abouzaid,john.mullins@polymtl.ca

doi:10.1016/j.entcs.2008.04.092

that is dedicated to WF languages, which we call BP-calculus. The advantage of the π-calculus, compared to other formalisms, is its capacity to model mobility, by passing channel names, as data, through channels. Another advantage is that BPEL is claimed by its designers to be based on the π-calculus.

We give the semantics of the BP-calculus in terms of BPEL. More work is needed to prove that the π-calculus based semantics for BPEL provided by Lucchi and Mazzara [8] is indeed the reverse mapping of our proposed mapping. In order to test the proposed approach, a prototype tool is being built. The prototype loads a formal definition of the services written in BP-calculus and allows for its formal verification using formulae expressed in π-logic [4]. The verification is done with the HAL (HD-Automata Laboratory) Toolkit [5], an integrated tool set for the specification, verification and analysis of systems modelled in π-calculus. The HAL toolkit uses the π-logic to specify the required properties. Finally, we illustrate the approach with a meaningful example.

1.1 Related work

Numerous works on formal Web Services verification have been conducted in the paste few years. We cite some of the most significant contributions related to this work. Lucchi and Mazzara [8] have proposed a mapping from a BPEL process to a π-based calculus which they call *web*π and which focuses on transactional aspects of the BPEL language. This work holds on BPEL 1.0 and does not handle some of the most recent innovations proposed in the new standard BPEL 2.0 [11]. Moreover, no proof of the correctness of the mapping is proposed. In [12] the authors have presented a first attempt at mapping WF-nets (a sub-class of Petri nets) onto BPEL processes. Their objective is to use a graphical formal language to create BPEL specifications, in order to facilitate the design and verification of composite WSs. A two-way mapping from Lotos to BPEL, is presented in [3,2]. Unlike our mapping, none of these mappings are extended to complex BPEL constructs such as compensation or fault handlers, nor do they preserve the BPEL designer's intention.

COWS [7] is a new foundational language for service-oriented computing whose design has been influenced by WS-BPEL. COWS allows for the encoding of more specific languages such as the WS-calculus [6]. We expect that COWS should also be able to encode the BP-calculus. Unlike COWS, however, the BP-calculus provides an explicit translation to BPEL, which takes into account all constructs of BPEL 2.0.

To summarize the contributions of this paper:

- We proposed a BPEL-based semantics for a new specification language based on the π-calculus, which will serve as a reverse mapping to the π-calculus based semantics introduced by Lucchi and Mazzara [8];
- This mapping has been implemented in a tool integrating the toolkit HAL and generating BPEL code from a specification given in the BP-calculus;
- Some previous work has been done on integrating model-checker toolkits and

generating BPEL code that has the same behavior as the model ([12],[3]), but as far as we know, our proposal is the first to take into account all significant structured activities, including scopes and handlers.

1.2 Structure of the paper

The rest of this paper is organized as follows. The next section briefly describes the BPEL language; and Section 3 presents the BP-language, our formalism for the modelling of BPEL processes. In Section 4, we present the mapping from the BP-calculus to BPEL. In Section 5, after giving an outline of the verification process, we illustrate the approach with a representative example. We conclude the paper in Section 6.

2 WS-BPEL language

A BPEL specification schedules the activities of a given process, the partners involved in the process, the messages exchanged between these partners, and the process for the handling of faults and exceptions (see Section 4.2 for details).

In April 2007, OASIS, the international standards consortium, announced that WS-BPEL 2.0 had been approved as an OASIS Standard[3]. With reference to [11], we list here the main new features of the WS-BPEL 2.0 specification:

- New functionalities have been added to variables and `<assign>` and `<copy>` activities. These include support and validation of XML schema complex types.
- A `<rethrow>` activity has been added to the fault handlers, and a `<terminationHandler>` has been added to the scopes.
- Partner link can now be declared local to a scope, and a `join` option has been added to the correlation sets. In addition, a `messageExchange` construct has been added to pair up concurrent `<receive>` and `<reply>` activities.
- Some new activities, such as serial and parallel `<forEach>` and `<repeatUntil>` have been added. `<Switch>` has been changed to `<if>-<elseif>-<else>` and `<terminate>` has been changed to `<exit>`. `<Compensate>` is renamed `<compensate>` and `<compensateScope>`.

3 The formalism

In this section the syntax (Section 3.1) and operational semantics (Section 3.2) of a new workflow calculus, that we call BP-calculus are defined. The design of the BP-calculus has been guided by the following general considerations:

- The calculus must express the usual routing constructs of existing workflow languages, in particular BPEL.

[3] http://www.oasis-open.org/news/oasis-news-2007-04-12.php

P	::=	IG	(input guarded)
	|	$(P \mid P')$	(parallel)
	|	$(P \parallel_{\tilde{u}} P')$	(sequential)
	|	$if\ (x = y)\ then\ P\ else\ Q$	(conditional)
	|	$\overline{x}^t\langle\tilde{u}\rangle.P\ (t \in \{'inv', 'rep', .throw'\})$	(annotated output)
	|	$!x(\tilde{y}).P$	(lazy replication)
IG	::=	$0 \mid x^s(\tilde{y}).P \mid IG + IG'$	(guarded choice)
S	::=	$(\nu\ \tilde{x})\{P, H\}$	(scope initiation)
H	::=	$\prod_i W_i(P_{i_1}, \cdots, P_{in_i})$	(scope's execution environment)
E	::=	$S \mid P \mid S\|P \mid E$	(global system)

Table 1
BP-calculus Syntax

- The calculus must serve as a theoretical formalism which allows for formal reasoning and not as a language to be implemented as is.

Given these considerations it also seems appropriate to define (Section 3.3) service containers (which are essential constructs to workflow languages) as instantiations of multi-hole contexts since they deal with scope concepts.

3.1 Syntax

The process syntax is given in Table 1. We provide a brief informal account of the intended interpretation of the processes:

- $\tilde{y} = (y_1, ..., y_n)$, (resp. $\tilde{u} = (u_1, ..., u_n)$) range over a countably infinite set **Var** of variables (resp. names).
- $\overline{x}^t\langle\tilde{u}\rangle$ $(t \in \{invoke, reply, throw\})$ is the usual output which can be an invocation, or a reply to a solicitation, or the throw of a fault, and which can be translated by a `reply`, an `invoke` or a `throw`. Semantically the annotation does not interfere. $\overline{x}^t\langle\rangle$ is a signal.
- $IG + IG'$ behaves like a guarded choice and is intended to be translated by a `pick`. IG is an input guarded process. We do not consider non-determinism in service behavior. The annotated input $(x^s(\tilde{y}))$ allows us to distinguish between simple input (no annotation), a fault catch (s = 'catch') processed by a fault handler or an event capture (s='onEvent').
- a replicated input $!x(\tilde{u}).P$ that consumes a message initiated by a $\overline{x}\langle\tilde{w}\rangle$ and behaves like $P\{\tilde{w}/\tilde{u}\}|!x(\tilde{u}).P$. The use of lazy replication is due to the fact that in BPEL, process instances are created by activities that receive messages (i.e. `receive` activities and `pick` activities).
- $P \parallel_{\tilde{u}} Q$ expresses a sequential composition of processes P' and Q' with synchronization upon $\tilde{u}$ (i.e $P = P'|\overline{x}^t\langle\tilde{u}\rangle$ and $Q ::= \{\tilde{u}/\tilde{z}\}x(\tilde{z}).Q'$). This operator is used when a process needs to transmit synchronization information $(\tilde{u})$ to its "follower". We use the notation $P \parallel Q$ (or $P.Q$) when nothing is transmitted.
- $if\ then\ else$ expresses a classical choice based on names equality and is intended to be naturally translated by an `if then else` construct in BPEL 2.0.

Input $(x(u).P)$ and replicated input $(!x(u).P)$ bind the names u and x. The scope of these binders is the process P. The free and bound names of processes are noted $fn(P)$ and $bn(P)$ respectively.

To accomplish the description of the BP-calculus, we need a mechanism that abstracts the BPEL scopes. Scopes act as containers for BPEL processes and handlers. A scope contains a primary structured activity which defines its normal behavior; it might contain variable definitions and handlers (fault, compensation, event and termination handlers). In case of normal execution, a scope is activated at the same time as its activities are and terminates when all its activities have been accomplished.

- Scope initialization occurs when a process or a scope is entered. It consists of instantiating and initializing the scope's variables and partner links; instantiating the correlation sets; and installing fault, termination and event handlers.
- H is the scope's execution environment that is modelled as the parallel composition of handlers W_i. Each handler is a wrapper for a tuple of processes $\widehat{P} = (P_1, \ldots, P_n)$. Not all handlers are mandatory.
- In a global system E, the restriction $((\nu\ \tilde{x})\{P, H\})$ binds names $\tilde{x}$, and its scope is the process P. The case where the variable x is restricted to a simple process P that is not within a scope, is the usual restriction of the π-calculus and is denoted by $(\nu x)P$.
- $W_i(P_{i1}, \cdots, P_{in_i})$ is the process obtained from the multi-hole context $W_i[\cdot]_1 \cdots [\cdot]_{n_i}$ by replacing each occurrence of $[\cdot]_j$ with P_{ij}. It is intended to abstract the BPEL handlers and will be detailed in Section 3.3.

3.2 Operational Semantics

Normally we define a reduction semantics by using a structural relation and a reduction relation. The structural relation is meant to express intrinsic meanings of the operators. The reduction relation defines the way in which processes evolve dynamically by means of operational semantics.

Definition 3.1 The structural congruence is the smallest equivalence relation closed under the rules of Table 2 and rules of the form

$$\frac{P \equiv Q}{\mathcal{C}[P] \equiv \mathcal{C}[Q]}$$

where $\mathcal{C}[\cdot]$ stands for any context of the form $R|[\cdot]$, $R \parallel_{\tilde{u}} [\cdot]$, $[\cdot] \parallel_{\tilde{u}} R$ or $(\nu x)\{[\cdot], H\}$.

Definition 3.2 The reduction relation $(\rightarrow)$ is the smallest relation closed under the rules in Table 3.

3.3 Instantiations of multi-hole contexts as handlers' abstractions.

Compensation Handler If defined, the compensation handler contains the activity to be performed if the activity of the scope is to be compensated.

$$
\begin{array}{rcl}
P \mid 0 & \equiv & P \\
P \mid Q & \equiv & Q \mid P \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R \\
(\nu\ \tilde{x})\{P, H\} \mid (\nu\ \tilde{x})\{Q, H'\} & \equiv & (\nu\ \tilde{x})\{Q, H'\} \mid (\nu\ \tilde{x})\{P, H\} \\
(\nu\ \tilde{x})\{P, H\} \mid 0 & \equiv & (\nu\ \tilde{x})\{P, H\} \\
(\nu\ \tilde{x})\{P, H\} \mid \big((\nu\ \tilde{x})\{Q, H'\} \mid (\nu\ \tilde{x})\{R, H''\}\big) & \equiv & \big((\nu\ \tilde{x})\{P, H\} \mid ((\nu\ \tilde{x})\{Q, H'\})\big) \mid (\nu\ \tilde{x})\{R, H''\} \\
P\ \|_{\tilde{u}}\ Q & \equiv & (\nu\ \ \tilde{x})\{P.\overline{x}^t\langle\tilde{u}\rangle \mid x(\tilde{u}).Q, \emptyset\} \\
P\ \|_{\tilde{u}}\ 0 & \equiv & P \\
P & \equiv & P\ \|_{\tilde{u}}\ 0 \\
P\ \|_{\tilde{u}}\ (Q\ \|_{\tilde{v}}\ R) & \equiv & (P\ \|_{\tilde{u}}\ Q)\ \|_{\tilde{v}}\ R
\end{array}
$$

Table 2
Structural Congruence

$$
\begin{array}{cr}
\dfrac{}{\overline{x}^t\langle y\rangle.P \mid x(z).Q \longrightarrow P \mid \{y/z\}Q} & \text{REACT} \\[2ex]
\dfrac{P \longrightarrow P'}{P|Q \longrightarrow P'|Q} & \text{PAR} \\[2ex]
\dfrac{P \longrightarrow P'}{(\nu\ x)\{P,H\} \longrightarrow (\nu\ x)\{P',H\}} & \text{P-RES} \\[2ex]
\dfrac{H \longrightarrow H'}{(\nu\ x)\{P,H\} \longrightarrow (\nu\ x)\{P,H'\}} & \text{H-RES} \\[2ex]
\dfrac{P \longrightarrow P' \quad Q \longrightarrow Q'}{(\nu x)\{P,H_1\}|(\nu x)\{Q,H_2\} \longrightarrow (\nu x)\{P',H_1\}|(\nu x)\{Q',H_2\}} & \text{C-RES} \\[2ex]
\dfrac{P \longrightarrow P' \ \textit{and}\ u \notin (fn(P) \cup fn(Q))}{P\|_{\tilde{u}}Q \longrightarrow P'\|_{\tilde{u}}Q} & \text{SEQ} \\[2ex]
\dfrac{P \equiv P' \quad P' \longrightarrow Q' \quad Q \equiv Q'}{P \longrightarrow Q} & \text{STRUCT} \\[2ex]
\dfrac{P \longrightarrow P' \ \textit{and}\ P \longrightarrow P' \ \textit{and}\ x=y}{\textit{if}(x=y)\ \textit{then}\ P\ \textit{else}\ Q \longrightarrow P'} & \text{IFT} \\[2ex]
\dfrac{P \longrightarrow P' \ \textit{and}\ P \longrightarrow P' \ \textit{and}\ x \neq y}{\textit{if}\ (x=y)\ \textit{then}\ P\ \textit{else}\ Q \longrightarrow Q'} & \text{IFF} \\[2ex]
\dfrac{x_1(\tilde{y}) \longrightarrow 0}{x_1(\tilde{y}).P_1 + x_2(\tilde{y}).P_2 \longrightarrow P_1} & \text{CHOICE}
\end{array}
$$

Table 3
Reaction and Transition Rules of Process behaviour

Correlation sets : are declared within a process or scope element, and can be used on every messaging activity to correlate messages with service instances.

Fault Handler Faults signalled by the `<Throw>` element are caught by the fault handler. The `<catch>` element permits the handling of a fault specified by a fault name, while the `<catchAll>` element catches any signalled fault.

Event Handler Event handlers define the activities' relative events such as incoming message or timeouts.

Termination Handler After terminating the scope's primary activity and all running event handler instances, the scope's customised or default termination handler is executed.

3.3.1 Handler wrappers

We formalize these by means of the multi-hole context. Note that for all these handlers, $throw$, en_i, dis_i are bound names for the whole system and are channels used for communication between processes.

Fault Handler

Given a tuple of faults $(\tilde{x})$ related to a tuple of processes $\widehat{P} = (P_1, \ldots, P_n)$:

$$W_{FH}(\widehat{P}) ::= en_{fh}().\Big(\sum_i \big(x_i^{catch}(\tilde{y}).(\overline{throw}^{throw}\langle\rangle \mid P_i)\big).\big(\overline{y_1}^{inv}\langle\rangle \mid \overline{y}_{fh}^{inv}\langle\rangle\big) + dis_{fh}()\Big)$$

A fault handler is enabled using the en_{fh} channel. The fault handler uses a guarded sum to execute an activity P_i, associated with the triggered fault (i) . After executing the associated activity, it then signals its termination to the activating process on the channel y_1 and to the scope on channel y_{fh}. If necessary, the fault handler is disabled using dis_{fh} channel. Internal faults are signaled using the `<throw>` activity.

Event Handler

Given a tuple of events $(\tilde{x})$ related to a tuple of processes $\widehat{P} = (P_1, \ldots, P_n)$:

$$W_{EH}(\widehat{P}) \;::=\; (\nu\tilde{x})\; en_{eh}().\Big(\prod_i \big(!x_i^{onEvent}(\tilde{y}).\overline{z}_i\langle\tilde{y}\rangle\big) + dis_{eh}()) \mid \prod_i (z_i(\tilde{u}).P_i)\Big)$$

An event handler enables itself by using en_{eh} channel, then waits for a set of events on the channels $(\tilde{x})$ each one of which are associated with an event. When the event occurs, the associated activity P_i is triggered. This is a typical usage of the `pick` construct. The event handler is disabled by using the dis_{eh} channel.

Compensation Handler

Let P_1 and P_2 be the scope and compensation activities.

$$W_{CH}(P_1, P_2) ::= \\ en_{ch}().\Big(z(\tilde{y}).(CC(P_1,\tilde{y}) \mid \overline{throw}^{throw}\langle\rangle) + inst_{ch}().(z(\tilde{y}).P_2 \mid \overline{y}_{ch}^{inv}\langle\rangle)\Big)$$

where:

$$CC(P_1,\tilde{y}) = \prod_{z' \in S_n(P_1)} \overline{z'}^{inv}\langle\tilde{y}\rangle$$

compensate children scopes (through channels in $S_n(P_1)$) of activity P_1.

A compensation handler associated with a scope z is first installed at the beginning of the scope through an input on channel $inst_{ch}$ ($\overline{y}_{ch}^{inv}$ signals this installation); it then executes its compensation activity P_2. If the compensation handler is invoked but not installed, it signals the termination of the scope activity through channel $throw$ and performs children compensation (CC). The compensation handler is invoked using the `<compensate>` activity.

Termination Handler (BPEL 2.0)

The termination handler is an innovation introduced in BPEL 2.0. With reference to [11], the termination handler is defined as follows: "The forced termination of a scope begins by disabling the scope's event handlers, and by terminating its primary activity and all running event handler instances. Following this, the customised `<terminationHandler>` for the scope, if present, is run. Otherwise, the default termination handler is run."

$$W_{TH}(P) ::= \; term(\tilde{u}).\big(\overline{dis_{eh}}^{inv}\langle\rangle \mid \overline{o}^{inv}\langle\tilde{y}\rangle \mid (P \mid \overline{throw}^{throw}\langle\rangle)\big)$$

A termination handler is invoked by the terminating scope using channel $term$. It disables the event handler using channel dis_{eh} and terminates the scope's primary activity using channel o. The customised or default Termination process P is then run.

Scope

Finally, putting all this together leads to the following semantics where the scope is represented by a hole context. Only the scope process P coresponding to the main actvity of the scope has to be provided by the designer.

Let $\tilde{x} = (throw, en_{eh}, en_{fh}, en_{ch}, dis_{fh}, inst_{ch}, dis_{eh}, term_{ch}, y_{fh}, y_{fh}, y_{fh})$. Then

$$\begin{aligned}(\nu\ \tilde{x})\{P,H\} ::= (\nu\ \tilde{x})\ & \big(W_{EH}(\widehat{A_{eh}}) \mid W_{FH}(\widehat{A_{fh}}) \mid W_{CH}(P_1,P_2) \mid W_{TH}(T)\\ & \mid \overline{en_{eh}}\langle\ \rangle.\overline{en_{fh}}\langle\rangle.\overline{en_{ch}}\langle\ \rangle.0\\ & \mid P \parallel \bar{t}\langle\ \rangle.0\\ & \mid c().(\overline{dis_{eh}}\langle\ \rangle.\overline{dis_{fh}}\langle\ \rangle.0 \mid \overline{inst_{ch}}\langle\ \rangle.\overline{term_{ch}}\langle\ \rangle.0)\\ & \mid y_{eh}().y_{fh}().y_{ch}()(x_z().(\overline{throw}\langle\ \rangle.0 \mid \overline{dis_{fh}}\langle\ \rangle.0) + t().\bar{c}\langle\ \rangle.0)\ \big)\end{aligned}$$

where the handlers, and the main process P, are performed in parallel with other processes which have the intentional semantics:

- $\overline{en_{eh}}\langle\ \rangle.\overline{en_{fh}}\langle\ \rangle.\overline{en_{ch}}\langle\ \rangle.0$ enables handlers
- $A \parallel \bar{t}\langle\ \rangle.0$ indicates normal termination by an output on channel t
- In case of normal termination, $c().(\overline{dis_{eh}}\langle\ \rangle.\overline{dis_{fh}}\langle\ \rangle.0 \mid \overline{inst_{ch}}\langle\ \rangle.\overline{term_{ch}}\langle\ \rangle.0)$ disables event and fault handlers, installs the compensation handler and runs the termination handler.
- $x_z().(\overline{throw}^{throw}\langle\ \rangle.0 \mid \overline{dis_{fh}}\langle\ \rangle.0) + t().\bar{c}\langle\ \rangle.0$ expresses that the scope can receive a termination signal on x_z from its parents, or can terminate normally by receiving a signal on t.
- $y_{eh}(), y_{fh}(), y_{ch}()$ are the channels used to indicate termination of handlers.

4 Automatic generation of BPEL code

We provide here, a detailed translation from BP-calculus into BPEL. The inverse mapping is a simplified version of Lucchi's semantics [8]. The aim is to provide

the means of specifying a process in either of both languages and to process formal verification in BP-calculus. The result of the verification can be iteratively refined. The final result is then an automatically generated BPEL code that is formally verified.

In the following translation we often abstract from some details since we aim only to provide designers with a template.

4.1 Outline of the translation

A WS orchestration shows WSs running in parallel, and this can be represented by a main BP-calculus process composed by parallel or synchronizing actions. So the basis of the mapping is the correlation between our BP-calculus and the BPEL activities.

We would like to map the BP-calculus process onto an hierarchical decomposition of specific adequate BPEL constructs. For example, it is important to identify a sequence of processes, although it is represented by a parallel operator and it is important to map it onto a BPEL sequence construct rather than onto the more general flow construct. It is the sequential operator ($\|$) that does this work.

The translation of processes into BPEL involves some restrictions on the initial process. Some of them are explained below:

- The possibility of a process receiving names on several channels coming from the same partner is not defined in BPEL. This means that such a configuration is not allowed: $\overline{x}\langle y\rangle|...x(y)...|...x(y)...$
- When a process receives a name, the process can only use the name to execute an output action in order to avoid the possibility of different services supporting the same operation. This condition is known as "*output capabilities of input names*" and is the basis of Local-π [9].
- Service-oriented computing (SOC) does not deal with non-determinism. Thus, only a choice (+) between terms that are prefixed inputs can be performed. This ensures conformance with the BPEL `pick` construct.

We assume that all bound variables occurring in the BP-calculus expression being translatied have been renamed with names distinct from each other and distinct from any free names in the expression. This obviates the need to actually invoke alpha-conversion in the translation. When translating communication primitives we assume that channels in the BP-calculus represent service operations and that partners (i.e portypes) are transmitted as variables.

4.2 The translation

The following is a translation of all BP-calculus constructs that will allow the automatic generation of BPEL code from a formal specification in BP-calculus. To this end we define the function $[\![.]\!] : P_{BP-calculus} \longrightarrow A_{BPEL}$ which maps BP-processes into BPEL activities.

Nil process

The process 0 does nothing and neither does the BPEL `empty` activity.

$$[\![0]\!] ::= \texttt{<empty/>}$$

Input

An input on a channel x is encoded as:

$$[\![x(\tilde{y}).P]\!] ::= \texttt{<receive partner="}y_1\texttt{" operation="}x\texttt{" variable="}y_2,..,y_n\texttt{"/>}\ [\![P]\!]$$

Note that the channel x is used to identify the desired operation in the service, and that y_1 is the channel name for the response and is used as the partner name. $\{y_2, ..., y_n\}$ denotes a set of variables. Because BPEL uses XML schema simple and complex types, it permits the use of sets of variables.

Note also that an annotated input ('catch' or 'onEvent') is expressed by appropriate constructs within fault and event handlers.

Output

The output operation is annotated to specify its nature (invoke, throw or reply).

Invoke Given a name x to identify the specific service operation:

$$[\![\overline{x}^{inv}\langle\tilde{u}\rangle.P]\!] ::= \texttt{<invoke partner="}u_1\texttt{" operation="}x\texttt{" variable="}u_2,..,u_n\texttt{"/>}[\![P]\!]$$

Reply This construct behaves exactly like an `invoke`:

$$[\![\overline{x}^{rep}\langle\tilde{u}\rangle.P]\!] ::= \texttt{<reply partner="}u_1\texttt{" operation ="}x\texttt{" variable="}u_2,..,u_n\texttt{"/>}[\![P]\!]$$

Throw Given a name t dedicated to fault notification an output with a 'throw' annotation is expressed by a `throw` construct as follows:

$$[\![\overline{t}^{throw}\langle\tilde{u}\rangle.P]\!] ::= \texttt{<throw faultName="}u_1\texttt{" faultVariable = "}u_2,..,u_n\texttt{"/>}[\![P]\!]$$

Parallel composition

The flow construct allows for parallel composition as does the parallel operator |.

$$[\![Q_1 \mid Q_2]\!] ::= \texttt{<flow>}\ [\![Q_1]\!][\![Q_2]\!]\ \texttt{</flow>}$$

Note that BPEL allows links that express synchronization dependencies between activities. These dependencies can, however, be expressed using basic constructs, therefore the use of such links should be avoided.

Sequential composition

The sequence operator $\|_{\tilde{u}}$ has been introduced to express a sequence between synchronized processes in order to generate a `<sequence>` element. Names $\tilde{u}$ are used for synchronization purposes.

$[\![Q_1 \ \|_{\tilde{u}} \ Q_2]\!]$ ::= <sequence>
$[\![Q_1]\!]$
<assign><copy>
<from variable="u"/><to variable="v"/>
</copy></assign>
$[\![Q_2]\!]$
</sequence>

The expression $(\nu y)(Q_1.\overline{y}\langle\tilde{u}\rangle \mid y(\tilde{v}).Q_2)$ is equivalent to $Q_1 \|_{\tilde{u}} Q_2$ and can thus be translated by the <sequence> construct. The <copy> element allows for the substitution of variable names coresponding to $\tilde{v}$ by those of $\tilde{u}$ in activity Q_2.

Choice

A guarded sum (choice) is translated as follows:

$[\![x_1(\tilde{i}).Q_1 \ + \ x_2(\tilde{j}).Q_2]\!]$ ::= <pick>
<onMessage partnerLink="i_1" operation="x_1"
variable="$i_2, .., i_n$" >
$[\![Q_1]\!]$
</onMessage>
<onMessage partnerLink="j_1" operation="x_2"
variable="$j_2, .., j_n$" >
$[\![Q_2]\!]$
</onMessage>
</pick>

Conditional

The new conditional syntax in BPEL 2.0 is as shown in the table below:

$[\![if \ (x = y) \ then \ Q_1 \ else \ Q_2]\!]$ ::=<if name = "ConditionName">
<condition >
getVariableProperty("VarName","x") =
getVariableProperty("VarName","y")
</condition>
$[\![Q_1]\!]$ <else> $[\![Q_2]\!]$</else>
</if>

Replication

The idea is to translate each $!P$ into a process A_P , recursively defined as $A_P(x) \stackrel{def}{=} P|A_P(x)$, to provide an unbounded number of copies of P. In the context of a lazy replication ($!x(\tilde{y}).P$), let $A_P = x(\tilde{y}).P|A_P$. Thus the translation of replication is:

$[\![A_P \ ]\!]$ ::= <process name="Ap">
<flow>
<sequence>

<receive partner="y_1" operation="x"
variable="$y_2, .., y_n$"/> (y1 is the caller of the replication)
$[\![P]\!]$
<invoke partner="q" operation="x"
variable="$y_2, .., y_n$"/>
</sequence>
$[\![Q]\!]$
</flow>
</process>

and

$[\![Q]\!]$ ::= <partnerLinks>
<partnerLink name="q" /> <partnerLink name="Ap"/>
</partnerLinks>
<sequence>
<receive partnerLink="Ap" operation="q"
variable="$y_2, .., y_n$"
createInstance="yes"/>
<if> <condition=StopCondition> <exit/>
<else>
<sequence>
<assign> ... </assign> (Prepare variables for recusive call)
<invoke partnerLink="Ap" operation="x"
inputVariable="$y_2, .., y_n$" outputVariable="$y_2, .., y_n$"/>
(Recusive call)
</sequence>
</else>
</if>
<reply partnerLink="y_1" operation="x"
variable="$y_2, .., y_n$"/> (Final answer to caller)
</sequence>

4.3 Translation of a restriction

A restriction is translated by means of a scope. A scope is a BPEL complex construct that requires a lot of attention. A restriction is used to limit the use of a variable within a set of processes; and this is the role of the `scope` activity in BPEL. The `<scope>` activity is used to define a nested activity with its own associated `<partnerLinks>`, `<messageExchanges>`, `<variables>`, `<faultHandlers>`, `<compensationHandler>`, `<terminationHandler>`, and `<eventHandlers>`. Then the expression $[\![(\nu x)\ \{Q, H\}]\!]$ where

$$H = W_{FH}(\widehat{A}_{fh}) \mid W_{EH}(\widehat{A}_{eh}) \mid W_{CH}(A_{ch}, C) \mid W_{TH}$$

is translated as follows:

$[\![W_{FH}(\widehat{A})]\!] :=$ (Fault Handler)	<faultHandlers> <catch faultVariable="x"> $[\![A_1]\!]$ </catch> <catch> ... </catch> <catchAll> $[\![A_n]\!]$ </catchAll> </faultHandlers>
$[\![W_{EH}(\widehat{A})]\!] :=$ (Event Handler)	<eventHandlers> <onEvent partnerLink="y_1" operation="x" variable="y" > <correlations> <correlation set="S_i" /> </correlations> $[\![A_i]\!]$ </onEvent> </eventHandlers>
$[\![W_{CH}(A_{ch}, C)]\!] :=$ (Compensation Handler)	<compensationHandler> $[\![A_{ch}(C, CC)]\!]$ </compensationHandler>
$[\![W_{TH}(A_{th})\})]\!] :=$ (Termination Handler)	<terminationHandler> $[\![A_{th}]\!]$ </terminationHandler>

Table 4
Translation of handlers

<scope>
<variables>
<variable name="x"/>
</variables>
<faultHandlers>$[\![W_{FH}(\widehat{A}_{fh})]\!]$</faultHandlers>
<eventHandlers>$[\![W_{EH}(\widehat{A}_{eh})]\!]$ </eventHandlers>
<compensationHandler>$[\![W_{CH}(A_{ch}, C)]\!]$ </compensationHandler>
<terminationHandlers>$[\![W_{TH}]\!]$ </terminationHandlers>
$[\![Q]\!]$
</scope>

The translation of handlers is given in Table 4.

The BPEL code for handlers is automatically generated according to the syntax shown in Section 3. The translation uses templates provided by the environment. The designer has only to specify the main activity of each handler (see the example in Section 5). Handlers' names must be maintained as they are in order to facilitate the translation into BPEL. All handlers are reccognized by their names and are translated accordingly. The portions of code shown in table 4 are generated for each handler in accordance with the above definitions. Note that annotated inputs allow for the generation of `catch` and `onEvent` elements within fault and event handlers.

Note that whenever a `<catchAll>` (for any fault) in the fault handler `<compensa-`

`tionHandler>` or `<terminationHandler>` is missing for a `<scope>`, they must be specifically created. The syntax of these default handlers can be found in [11].

5 Illustration of the approach

The proposed verification approach is the following. First, processes are specified in the BP-calculus. This specification is translated into a syntax compatible with the HAL tool. This translation is not isomorphic because annotations are lost, but their absence does not interfere in the verification process.

Properties are expressed in π-logic, also in a syntax that is compatible with the verification tool. The π-logic ([5], [4]) extends the modal logic introduced by Milner [10] with some expressive modalities ('strong next', 'weak next', 'eventually'). As long as no properties are satisfied, the designer can correct the BP calculus, taking in account the counter-example provided by the HAL toolkit, and then relaunch the verification tool. The resulting BP-calculus specification is then translated into BPEL.

In the remainder of this section, we illustrate this approach with the example of a process handling an ask for a stock quote: a *Caller* asks for a stock quote; the BPEL process takes the stock rating from a *Provider* and sends it to the Caller.

For sake of simplicity we consider a unique scope, with an event handler and a fault handler that only covers the following errors when asking for the quote: the quote is not available; the symbol is not valid; and timeout.

5.1 The BP-model

Using definitions provided in Section 4.3 the service is described by the process:

$$AskQuote \;=\; (\nu\; throw, en_{eh}, en_{fh}, dis_{fh}, dis_{eh})\{A, H\}$$

where

$$\begin{aligned} A = {} & wantquote(\text{“}caller\text{”}, req) \\ & \parallel \overline{getquote}^{inv}\langle \text{“}provider\text{”}, req\rangle \\ & \parallel (getquote(\text{“}provider\text{”}, resp) \mid \overline{throw}^{throw}\langle\,\rangle) \\ & \parallel \overline{wantquote}^{rep}\langle \text{“}caller\text{”}, resp\rangle \end{aligned}$$

and

$$H = W_{FH}(\widehat{A_f}) \mid W_{EH}(\widehat{A_e})$$

Then, each handler is modelled as shown below:

- The **fault handler** deals with 3 kinds of faults: $S_f = \{f_{snv}, f_{qna}, f_{timeout}\}$, and their associated activities: $\widehat{A_f} = \{F_{snv}, F_{qna}, F_{timeout}\}$. It is modelled as:

$$W_{FH}(\widehat{A_f}) = en_{fh}().\Big(\Big(\Big(f^{catch}_{snv}(wantquote,\tilde{u}).(\overline{throw}^{throw}\langle\rangle \mid F_{snv}(wantquote))\Big) + \Big(f^{catch}_{qna}(wantquote,\tilde{u}).(\overline{throw}^{throw}\langle\rangle \mid F_{qna}(wantquote))\Big) + \Big(f^{catch}_{timeout}(wantquote,\tilde{u}).(\overline{throw}^{throw}\langle\rangle \mid F_{timeout}(wantquote))\Big)\Big) \parallel \Big(\overline{r}^{inv}\langle\rangle \mid \overline{y}^{inv}_{fh}\langle\rangle + dis_{fh}()\Big)\Big)$$

where

$F_{snv}(wantquote) = \overline{wantquote}\langle snv\rangle$ — to handle "symbol not valid" fault

$F_{qna}(wantquote) = \overline{wantquote}\langle qna\rangle$ — to handle "quote not available" fault

$F_{timeout}(wantquote) = \overline{wantquote}\langle timeout\rangle$ — to handle "timeout" fault

- The **event handler** waits for a unique event (timeout) and processes an activity ($A_{Timeout}$) associated with this event:

$$W_{EH}(\widehat{A_e}) = (\nu\ timeout)en_{eh}().\Big(\big(timeout^{onEvent}().\overline{z}\langle\rangle\ +\ dis_{eh}()\big) \mid z().A_{Timeout}\Big)$$

The ($A_{Timeout}$) activity consists of throwing a timeout error and can be modelled as:

$$A_{Timeout} = \overline{throw}^{throw}\langle timeout\rangle$$

- Finally, the whole service is modelled as:

$$AskQuote = (\nu\ throw, en_{eh}, en_{fh}, dis_{fh}, dis_{eh})\Big(W_{EH}(\widehat{A_e}) \mid W_{FH}(\widehat{A_f}) \mid \big(\overline{en_{eh}}\langle\rangle.\overline{en_{fh}}\langle\rangle\big) \mid \big(A \parallel \overline{t}\langle\ \rangle\big) \mid c().\Big(\overline{dis_{eh}}\langle\ \rangle.\overline{dis_{fh}}\langle\ \rangle \mid \big(x_z().(\overline{throw}\langle\ \rangle \mid \overline{dis_{fh}}\langle\ \rangle) + t().\overline{c}\langle\ \rangle\big)\Big)\Big)$$

The conversion of this BP-specification into a syntax that is compatible with the HAL Toolkit is shown in Appendix A.

Many functional properties that express desirable attributes of services and service-oriented computing applications have been defined so far: responsiveness, availability, reliability, fairness or non-ambiguity (see, e.g. [1] to get a complete list). One of the relevant abstract properties that help to illustrate our framework is *responsiveness*. A service is *responsive* if it guarantees a response to every received request. The *responsiveness* of the broker service for example, can be defined as being that whenever the broker service receives an ask (*req*) for a stock (*st*) rate, it delivers a response (*resp*) to its client. This property is formalized as follows:

```
P = AG([wantquote?(req,st)]EF<wantquote!  ⟨resp, st⟩ >true).
```

With the HAL toolkit, `P` is found true when one discounts fault and event handling, but false when one does not, as is to be expected given the occurence of faults.

5.2 Generation of BPEL code

When all desired properties are proven to be true, one can proceed to automatic generation of BPEL code. The translation into BPEL is done as follows. From the equation:

$$AskQuote = (\nu\ throw, en_{eh}, en_{fh}, dis_{fh}, dis_{eh})\{A, H\},\ H = W_{FH}(\widehat{A_f})|W_{EH}(\widehat{A_e})$$

containing a scope with a fault handler (W_{FH}), an event handler (W_{EH}), and a primary scope's activity A, the following template can be drawn:

```
<process name="AskQuote">
     <scope>
     <faultHandlers/> <eventHandlers/>
     <sequence><MainActivity></sequence>
     </scope>
</process>
```

Let us now detail each component:

Fault Handler:

The fault handler is composed of three activities, $\widehat{A_f} = \{F_{snv},\ F_{qna}, F_{timeout}\}$ (each one corresponding to a `catch`). The invoked PortType and operation are the same for the three faults. The only difference is the parameter for the name of the fault. The fault handler can be mapped as follows:

```
<faultHandlers>
      <catch faultName="SymbolNotValid"
          faultVariable="Fault">
      <sequence> <assign> <copy>
      <from exp="string('SymbolNotValid')"/>
          <to variable="Fault" part="error"/>
      </copy> </assign>
      <invoke partnerLink="Caller" operation="WantquoteFault"
      inputVariable="Fault"/>
      </sequence> </catch>
      <catch faultName="QuoteNotAvailable"
           faultVariable="Fault">
      <sequence> <assign> <copy>
      <from exp="string('QuoteNotAvailable')"/>
           <to variable="Fault" part="error" />
      </copy> </assign>
      <invoke partnerLink="Caller" oper="WantquoteFault"
          inputVar="Fault" />
      <sequence> </catch>
      <catch faultName="Timeout">
      <sequence> <assign> <copy>
```

```
      <from expression="string('Proc timeout')"/>
          <to variable="Fault" part="error"/>
      </copy> </assign>
      <invoke partnerLink="Caller" oper="WantquoteFault"
          inputVariable="Fault"/>
      </sequence>
      </catch>
<faultHandlers>
```

Event Handler

An event handler is identified by its name. In this case, it is composed of a single activity: throwing a timeout error. The input annotation indicates that it be translated by a `throw` construct.

```
<eventHandlers>
      <onAlarm for="Timeout"> <throw faultName="Timeout"
               faultVariable="Fault"/>
      </onAlarm>
</eventHandlers>
```

Main activity: The main activity is a sequence starting with a `<receive>` statement, followed by a synchronous `<invoke>` and terminated by a `<reply>`. The generated code is as follows:

```
<sequence>
      <receive partner = "caller" operation ="wantquote"
          variable = "req" />
      <invoke partner = "provider" operation ="getquote"
           InputVariable = "req" OutputVariable = "resp"/>
      <reply partner = "caller" operation ="wantquote"
          variable = "resp" />
</sequence>
```

6 Conclusion

Formal methods can contribute to solve the important and exacting problem of designing reliable and secure orchestrations of Web Services. Several approaches have been proposed. In order to integrate a well-established model-checking toolkit, and in order to generate BPEL code that has the same behavior as the model, we have proposed a π-calculus-like language (which we call BP-calculus) to formalize such specifications together with their BPEL-based semantics. Preliminary work strongly suggests that this mapping should be the reverse mapping of the π-calculus-based semantics of BPEL that was proposed by Lucchi and Mazzara [8]. Such an outcome would guarantee that the BPEL code thus generated will behave the same way as the BP-calculus specification. To demonstrate the relevance of our approach, we have implemented a prototype integrating the HAL Toolkit and illustrated it

with a representative example. Future work wil be in two directions. First we need to formally establish that the proposed BPEL-based semantics can be reversed by Lucchi and Mazzara's π-calculus-based semantics [8]. Secondly, we intend to integrate the tool into a specific environment designed to speed up the realization of formally verified Web Services orchestrations, while still allowing for the re-engineering of existing BPEL processes.

References

[1] Alonso, G., F. Casati, H. Kuno and V. Machiraju, "Web Services: Concepts, Architectures and Applications," Springer, 2004.

[2] Chirichiello, A. and G. Salan, *Encoding abstract descriptions into executable web services: Towards a formal development*, in: *2005 IEEE/WIC/ACM International Conference on Web Intelligence WI'2005*, Compine, France, 2005.

[3] Ferrara, A., *Web services: a process algebra approach,*, in: *Proceedings of the 2nd international conference on Service oriented computing*, New York, NY, USA, 2004, pp. 242–251.

[4] Ferrari, G., G. Ferro, S. Gnesi, U. Montanari, M. Pistore and G. Ristori, *An automata based verification environment for mobile processes*, in: E. Brinksma, editor, *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, LNCS **1217** (1997), pp. 275–289.

[5] Ferrari, G., S. Gnesi, U. Montanari and M. Pistore, *A model checking verification environment for mobile processes,*, Technical report, Consiglio Nazionale delle Ricerche, Istituto di Scienza e Tecnologie dell'Informazione 'A. Faedo' (2003.).

[6] Lapadula, A., R. Pugliese and F. Tiezzi, *A wsdl-based type system for ws-bpel.*, in: *Proc. of COORDINATION'06*, Lecture Notes in Computer Science **4038** (2006), pp. 145–163.

[7] Lapadula, A., R. Pugliese and F. Tiezzi, *A Calculus for Orchestration of Web Services.*, in: *Proc. of 16th European Symposium on Programming (ESOP'07)*, Lecture Notes in Computer Science **4421** (2007), pp. 33–47.

[8] Lucchi, R. and M. Mazzara, *A pi-calculus based semantics for ws-bpel*, Journal of Logic and Algebraic Programming, Elsevier press (2007).

[9] Merro, M. and D. Sangiorgi, *On asynchrony in name-passing calculi*, in: *ICALP*, 1998, pp. 856–867.

[10] Milner, R., J. Parrow and D. Walker, *Modal logics for mobile processes*, Theoretical Computer Science, (1993).

[11] Oasis, *Web service business process execution language version 2.0 specification, oasis standard,*, *http://docs.oasis − open.org/wsbpel/2.0/wsbpel − v2.0.pdf* (2007).

[12] van der Aalst, W. M. P. and K. B. Lassen, *Translating unstructured workflow processes to readable bpel: Theory and implementation.*, the International Journal of Information and Software Technology (INFSOF) (2006).

A HAL code of the example

Process

```
    define Process(wq, gq, throw) = (ene)(enf)(dise)(disf)
(timeout)(fsv)(fna) (fto)(z)(r)(yfh) (ene!z.enf!z.
(A(wq, gq, throw) | EH(ene, timeout, throw, dise) | FH(enf,
fsv, fna, fto, throw, wq, r, yfh, disf) ))

    define EH(ene, timeout, throw, dise) =(z) (ene?(z).
(timeout?(z).throw!timeout.nil + dise?(z).nil))

    define FH(enf, fsv, fna, fto, throw, wq, r, yfh, disf) =
enf?(z).F(wq, r, yfh, disf, throw)

    define F(wq, r, yfh, disf, throw) = (y)(x)((FV(wq, throw,x)
+ FA(wq, throw,x) + FT(wq, throw,x)) | G(r, yfh, disf,x))

    define G(r, yfh, disf ,x) =(z) (x?(z).
((r!z | yfh!z) + disf?(u)))

    define FV(wq, throw,x) = (z)(y)(x)(fsv!wq.x!y
| x?(y).(throw!v | wq!snv).x!v)

    define FT(wq, throw,x) = (z)(y)(x)(fto!wq.x!y
| x?(y).(throw!v | wq!to).x!v)

    define FA(wq, throw,x) = (z)(y)(x)(fna!wq.x!y
| x?(y).(throw!v | wq!qna).x!v)

    define A(wq, gq, throw) = (b)(req)(rep) wq?(req).
gq!b.gq?(b).wq!rep.Process(wq,gq,throw)

    build Process
```

Pi- Formula

```
     define formula1 = EF(<wq!rep>true | <throw!timeout>true)
```

B BPEL

In this section, we briefly present main BPEL activities. There are 2 kinds of activities: basic activities that describe elemental steps of the process behavior and structured activities that encode control-flow logic.

B.1 Basic activities

Main basic activities are: `Empty`, `invoke`, `receive`, `reply` , `throw` and `compensate`.

Empty

Empty process represents a terminated activity and is introduced by the `<empty>` element:

```
<empty standard-attributes>
      standard-elements
</empty>
```

Invoke

The invocation of a Web Service can be both synchronous and asynchronous according to the interaction modality used by the invoked service. The invoke activity is defined by the `<invoke>` element:

```
<invoke partnerLink="ncname" portType="qname"
          operation="ncname"
     inputVariable="ncname"?  outputVariable="ncname"?
     standard-attributes>
</invoke>
```

`Partner Link` indicates the partner supplying the operation, `portType` the access point of the invoked operation and the transmission protocol used to transmit SOAP requests. Variables are passed by means of `inputVariable` and `outputVariable`.

Receive

The `receive` construct represents an input on a specified channel. The specification does not accept the simultaneous enabling of more than one `receive` associated with the same partner, portType and operation. This activity is introduced by the `<receive>` element:

```
<receive partnerLink="ncname" portType="qname"
            operation="ncname"
    variable="ncname"?  createInstance="yes|no"?
    standard-attributes>
    standard-elements
</receive>
```

Reply

is used to generate a response. Therefore a `reply` activity must always be preceded by a receive activity for the same partner, port type and operation. This function is defined by the `<reply>` element:

```
<reply partnerLink="ncname" portType="qname"
          operation="ncname"
    variable="ncname"?  faultName="qname"?
    standard-attributes>
    standard-elements
</reply>
```

Assign

This activity allows for the updating of variables. The syntax is:

```
<assign validate="yes|no"?  standard-attributes>
    standard-elements
    (
    <copy keepSrcElementName="yes|no"?  >
    from-spec to-spec
    </copy>
    )+
</assign>
```

Throw

This activity is used to specifically signal an internal fault. When the `throw` is performed, the fault name has to be specified with some variables containing information about the faults. Faults are caught and processed by fault handlers:

```
<throw faultName="qname" faultVariable="ncname"?
    standard-attributes>
    standard-elements
</throw>
```

Compensate

A compensation handler is used in cases where exceptions occur, or when a partner requests reversal, and is invoked by the `compensate` activity. BPEL 2.0 has introduced the `<compensateScope>` activity that is used to start compensation on a specified inner scope that has already completed successfully. This activity should only be used from within a fault handler, another compensation handler, or a termination handler. If no compensation handler is defined, the default handler compensates all the children scopes. The syntax is:

```
<compensate scope="ncname"?  standard-attributes>
     standard-elements
</compensate>
```

Other basic activities are: `wait`, `exit`, `rethrow`.

B.2 Structured activities

Structured activities describe how a business process is created by composing basic activities into complex structures expressing workflow, control patterns, dataflow, faults handling, external events management and coordination of messages exchange between process instances involved in a business protocol. Main structured activities of BPEL include: sequential composition (`sequence`) , branching (`switch`), parallel composition and synchronization (`flow`), and nondeterministic choice (`pick`).

Sequence

A sequential activity contains one or more activities that are performed in the order they are listed within the `sequence` element. This activity is introduced by the `<sequence>` element:

```
<sequence standard-attributes>
    standard-elementsactivity+
</sequence>
```

Flow

This construct represents the concurrent execution of primitive activities. It is introduced by the `<flow>` element:

```
<flow standard-attributes>
    standard-elements
    <links>?
    <link name="ncname">+
    </links>
    activity+
</flow>
```

Links

A link represents a connection between two activities; one defined as the source and one defined as the target. Both the source and the target must define explicitly their role in the syntax. A link allows for the specification of some order in the execution of the parallel activities expressing the interdependencies among activities, thus allowing synchronization between some activities. Some restrictions hold for links (for example a link cannot cross a scope).

Conditional

In previous versions of the language, the `switch` activity consisted of an ordered list of one or more conditional branches defined by `case` elements and an optional `otherwise` branch. Conditions are expressed by boolean expressions. This construct has been changed to `if-elseif-else` in BPEL 2.0:

```
<if standard-attributes>
       standard-elements
       <condition expressionLanguage="anyURI"?>bool-expr
       </condition>
       activity
       <elseif>*
       <condition expressionLanguage="anyURI"?>bool-expr
       </condition>
       activity
       </elseif>
       <else>?
       activity
       </else>
       </if>
```

Repetitive Activities

While previous version of BPEL had a unique construct to express a repetition i.e the `<while>` construct, BPEL 2.0 has introduced 2 more activities: `<RepeatUntil>` and `<Foreach>`. The syntax for the `while` construct is:

```
<while standard-attributes>
    standard-elements
    <condition expressionLanguage="anyURI"?>bool-expr
    </condition>
    activity
</while>
```

Pick

This construct represents the nondeterministic execution of one of several paths depending on an external event. Possible events are the arrival of some message, or an alarm clock based on a timer. The activity awaits the occurrence of one of the defined events and performs the associated activity. If more than one of the events occur then the selection depends on which one occurred first. It is introduced by the `<pick>` element:

```
<pick createInstance="yes|no"?  standard-attributes>
     standard-elements
     <onMessage partnerLink="ncname" portType="qname"
     operation="ncname" variable="ncname"?>+
     <correlations>?
     <correlation set="ncname" initiate="yes|no"?>+
     </correlations>
     activity
     </onMessage>
</pick>
```

Scope

A scope (introduced by the `<scope>` element) provides the behavior context for each activity, and can provide a nested activity with its own associated `<partnerLinks>`, `<messageExchanges>`, `<variables>`, `<correlation Sets>`, `<faultHandlers>`, `<compensationHandler>`, `<terminationHandler>` and `<eventHandlers>`. Variables exist only within the scope (as local variables). Compensation is used after the successful termination of the scope where it is defined. The syntax is:

```
<scope variableAccessSerializable="yes|no"
                    standard-attributes>
      standard-elements
      <variables>?
      ...
      </variables>
      <correlationSets>?
      ...
      </correlationSets>
      <faultHandlers>?
      ...
      </faultHandlers>
      <compensationHandler>?
      ...
      </compensationHandler>
      <eventHandlers>?
      ...
      </eventHandlers>
      <TerminationHandlers>?
      ...
      </TerminationHandlers>
      activity
</scope>
```

Compensation handler

A `compensation handler` can be performed when the scope terminates in a successful way. It represents a part of the process that is reversible and acts as a wrapper for these activities. It is defined by the `<compensationHandler>` element:

```
<compensationHandler>?
      activity
</compensationHandler>
```

Fault handler

A `fault handler`, attached to a `scope`, provides a way to define a set of customised fault-handling activities that are syntactically defined as `catch` activities. Each catch activity is defined to intercept a specific kind of fault, defined by a globally unique fault name and a variable for the data associated with the fault. `catchAll` is able to capture any fault that is not specifically handled. It is possible for a fault to match more than one fault handler. This function is defined by the `<faultHandlers>` element:

```
<faultHandlers>?
<!-- there must be at least one fault handler or default -->
      <catch faultName="qname"?  faultVariable="ncname"?>*
      activity
      </catch>
      <catchAll>?
      activity
      </catchAll>
</faultHandlers>
```

Event handler

Any scope, as well as the whole business process, can be associated with event handlers. A handler is associated to a particular event (an incoming message or a timeout) and defines the activities to be performed if this event occurs. event handlers are defined using the `<eventHandlers>` element:

```
<eventHandlers>?
      <onEvent partnerLink="NCName"
      portType="QName"?
      operation="NCName"
      ( messageType="QName" | element="QName")?
      variable="BPELVariableName"?
      messageExchange="NCName"?>*
      <correlations>?
      <correlation set="NCName" initiate="yes|join|no"?/>+
      </correlations>
      <fromParts>?
      <fromPart part="NCName" toVariable="BPELVariableName"/>+
      </fromParts>
      <scope ...>...</scope>
```

```
      </onEvent>
      <onAlarm>*
      (
      <for expressionLanguage="anyURI"?>duration-expr</for>
      |
      <until expressionLanguage="anyURI"?>deadline-expr
      </until>
      )?
      <repeatEvery expressionLanguage="anyURI"?>?
      duration-expr
      </repeatEvery>
      <scope ...>...</scope>
      </onAlarm>
</eventHandlers>
```

Termination handler (BPEL 2.0)

The scope's (custom or default) `terminationHandler` is executed after the scope's primary activity and all running event handler instances have terminated. The syntax is:

```
<terminationHandler>
     activity
</terminationHandler>
```