International Journal on Web Service Computing (IJWSC), Vol.11, No.1, March 2020

Toward a Formalization of BPEL 2.0 : An Algebra Approach

L. Boumlik[1], M. Mejri[1], and H. Boucheneb[2]

[1] Laval University, QC, Canada
[2] Polytechnique Montreal University, QC, Canada

Abstract. The WS-BPEL 2.0 (Web Service Business Process Execution Language) has been the dominant standard to describe Web Services (WS) orchestration approach. It is a rich and expressive language that provides interesting features, among them we find four mechanisms to deal with abnormal situations (Event, Fault, Compensation and Termination) handlers, EFCT- handlers. However, WS-BPEL is not rigorously defined as a formal language making EFCT- handlers complicated and ambiguous. This paper aims to remove ambiguities from EFCT-handlers by formalizing their semantics.

Keywords: Web Services orchestration, WS-BPEL 2.0, Formal methods, Process Algebra, Operational Semantics.

1 Introduction

Web services are a successful instantiation of SOC [1, 2, 3], they are used by a wide array of companies and governments because of their autonomy, reusability, flexibility, and platform-independence. Web services aim to achieve safe and transparent interoperability, using a framework based primarily on XML (eXtensible Markup Language) [4], SOAP [5], WSDL [6] and UDDI [7]. The WSDL (Web Service Description Language) provides an abstract language to describe messages to be exchanged between services. The SOAP (Simple Object Access Protocol) is a protocol for exchanging structured information between the involved parties and the UDDI (Universal Description Discovery and Integration) is used to publish and discover web services.

Having the possibility of constructing new web services by composing existing ones has opened a new interesting perspective and has significantly influenced the way industrial applications are developed. The Web service orchestration approach is now an integral part of the modern Web, such as cloud computing environments [8, 9, 10], Internet of Things (IoT)[11] , social networks and Web technologies [12]. A variety of languages have been introduced to address WS composition, such as WS-BPEL [13], WS-CDL [14], and WSCI [15]. In this paper, we focus on WS-BPEL 2.0, the dominant standard to express orchestration process, later called BPEL, for short. It is a rich and expressive language that provides interesting features. Among them, we find four mechanisms to deal with special or abnormal situations: event, fault, compensation and termination handlers, referred to us (EFCT) handlers. However, the semantics of BPEL is not formally defined and can be very confusing. For instance, even if the global ideas of the fault handler and the compensation handler are intuitive, it remains that there are a large number of particular cases that need to be considered: Which fault handler catches which error? Which compensation handler needs to be called after a given error? Which activities need to be stopped when an error occurs? What happens if an error occurs during a compensation? What happens if an error occurs in a fault handler itself? These are only a part of the particular situations that need to be clarified. Combined with other features, like the event handler and the termination handler, the number of these kinds of questions explodes justifying the urgent need for a suitable formalization for BPEL.

This paper provides a detailed formalization of the EFCT of BPEL, that address the above questions, through a process algebra endowed with a small step operational semantics. The formalization of BPEL opens the door to the use of tools and techniques provided by the formal methods community to address many interesting problems including the following: Does a service S satisfies a security policy Φ (i.e., $S \models \Phi$)? Is the behavior of a service S equivalent to the one of a service S' (i.e., $S \approx S'$)? Is the behavior of a service S included in the one of a service S' (i.e., $S \sqsubseteq S'$)?

The formalization is also a great tool for improving the language itself in different directions. In fact, it allows not only to remove its ambiguities, but also to detect the incompleteness of its specification (behaviors that are not explicitly described by the specification) as it was stated in [16]. Furthermore, it allows to detect the redundancy of some operators and gives the possibility to simplify the language by substituting a group of constructors or concepts with better new ones as advocated in [17].

The remainder of this paper is organized as follows. Section 2 gives the syntax of an abbreviated version of BPEL called AV-BPEL. Section 3 formalizes the semantics of AV-BPEL. Some examples to show the practical use of our approach are given in the section 4. Some related works are introduced in Section 5. Finally, a conclusion and future works are given in Section 6.

2 Syntax of AV-BPEL: Abbreviated Version of BPEL

BPEL is an XML-based language which is very verbose. To abbreviate the presentation of its syntax and semantics, we introduce AV-BPEL, which is not a new language but just a concise representation of BPEL that is more suitable for the definition of an operational semantics. Many tools use graphical notation, including Jdevelopper[18] and OpenESB[19], to simplify the use of BPEL. But these graphical notations are also not very convenient for the operational style semantics. The EBNF grammar of AV-BPEL is as shown by Table 1.

2.0.1 Process, handlers, environment and scope

- **Business Process**: A business process is composed of an environment $\mathcal{E}_p$, handlers (gathered in $\mathcal{H}_p$) and an activity A.
- **Environment**: An environment defines declarations (variables, partner links, message exchanges, correlation sets, etc.). However, for the sake of simplicity, we assume that it is a mapping from variables to values.
- **Handlers**: A handler is a piece of code responsible of dealing with a particular situation such that errors, compensations, coming events and termination. It is activated only when some conditions hold. For example, event handler is activated when an event (i.e, some particular incoming inputs), error handler is activated when errors occurs, compensation handler is activated to compensate (reverse or cancel) the effect of successful accomplished activities of a transaction, when a part of it fails. Termination handler gives the ability to force the termination of some activities when an error occurs. These handlers give a clean separation of different aspect of the process, like in aspect-oriented paradigm, which is very useful, especially, for maintenance. A process can have only a fault handler and event handler. However, its scopes could have the other handlers. Even handlers can have their own handlers.
- **Scope**: A scope is an activity with a context that influences its execution. The context contains an environment and handlers that are available only to the activity of the scope. The scope can be nested hierarchically and the context of a father is always accessible to its children. The root context is provided by the process itself.

2.0.2 Basic and structured activities

- **Basic Activities**: Beside *exit* and *empty* activities, we mainly find communication (invoke, receive, etc.), errors (throw and catch), compensation , assignment and waiting. Some of these activities are reserved to a particular handler. For instance, the error catching activities could be found only in the fault handler.
  - $c?m$ (receive): It allows receiving a message on the channel c. A channel name informs about a partner link, a port type and an operation. The message m is the variable that received that data. The receive activity is mostly used to instance a business process.
  - $c \hookleftarrow m$ (replay) : It allows respond to a request previously accepted through a receive.
  - $c!m$ (asynchronous invoke): It invokes, asynchronously, a service available at the channel c with a message m.
  - $c!(m, m')$ (synchronous invoke): invokes, synchronously, a service available at the channel c with a message m. The remaining part of the invoking activity will be blocked until a message is received on c and its value is stored in m'.
  - 0 (exit): It immediately ends a process.
  - 1 (empty): It does nothing.
  - $\Rsh_{er}$ (throw): It is used to report a fault named er.
  - $assign$ (Assign): It is used to update variables.
  - $waitFor/waitUntil$ (wait): This activities delay a task for a period of time or until a fixed date.
  - $\$$ and $\$_n$ (compensate): These activities are used to compensate all the scope ($\$$) or a particular one named n ($\$_n$).
- **Structured Activities:** BPEL provides a various control-flow patterns for business processes. Besides basic activities, it encompasses:
  - $A_1.A_2$ (*sequence*). First A_1 is executed, then it followed by A_2.
  - $A_1 \lhd b \rhd A_2$ (conditional behaviors- *if*): *if* (b) A_1 *else* A_2.
  - b^*A (*while*): *while b do A*.
  - A^*b (*repeat until*): *repeat A until b*.
  - E (*pickup*): A pickup activity E waits for an event (onMessage: "receive" ($c?x.A$) or on Alarm: *waitFor* or *waitUntil*) from a set of events, then it executes the activity associated with that event. Once one event is fired, the others are ignored. A *pickup* can also create instance $((\sum_{i=1}^{n} c_i?m_i.A_i)^+$: an instance containing A_i is created when a message is received on c_i and stored in m_i).
  - $A_1||A_2$ (*flow:*) The flow activity provides parallel composition: A_1 and A_2 run in concurrency.
- *Composition:* A *Composition* can be a business process P, a message on transit $c!m$ or a parallel composition of two *Composition*. The notion of *Composition* does not belong to the syntax of BPEL, but it is useful to formalize the interaction between processes.

Table 1: Syntax of AV-BPEL Process.

Basic Activities		
Communication Activities		
A_{com} ::=	$c?m$	(receive)
$\mid$	$c!m$	(invoke)
$\mid$	$c!(m, m')$	(synchronous invoke)
$\mid$	$c \hookleftarrow m$	(reply)
Simple Activities		
A_b ::=	0	(exit)
$\mid$	1	(empty)
$\mid$	$\Rsh_{er}$	(throw)
$\mid$	$x := exp$	(assign)
$\mid$	$waitFor(n)$	(wait for)
$\mid$	$waitUntil(t)$	(wait until)
$\mid$	A_{com}	(Communication Activities)
Compensation Handler Basic Activities		
A_c ::=	A_b	(Basic Activities)
$\mid$	$\$_n$	(compensate n)
$\mid$	$\$$	(compensate all)
Termination Handler Basic Activities		
A_t ::=	A_c	(Termination Handler Activities)

Structured Activities		
Scope's Activities		
A, A_1, A_2 ::=	A_b	
$\mid$	$A_1.A_2$	(**sequence**)
$\mid$	$A_1 \lhd b \rhd A_2$	(**if:** if (b) A_1 else A_2)
$\mid$	b^*A	(**while:** while(b) do A)
$\mid$	A^*b	(**repeat:** repeat A until b)
$\mid$	E	(**pickup**)
$\mid$	$(\sum_{i=1}^{n} c_i?m_i.A_i)^+$	(**pickup with instance**)
$\mid$	$A_1 \| A_2$	(**flow:** parallel composition)
$\mid$	$< \mathcal{E}, \mathcal{H}_s, A >$	(**scope**)
Compensation-Handler's Activities		
A_{ch} ::=	$A[A_c/A_b, \mathcal{H}_h/\mathcal{H}_s]$	
The definition of A_{ch} is same as A except that A_b is replaced by A_c and $\mathcal{H}_s$ by $\mathcal{H}_h$		
Termination-Handler's Activities		
A_{th} ::=	$A[A_t/A_b, \mathcal{H}_h/\mathcal{H}_s]$	
The definition of A_{th} is same as A except that A_b is replaced by A_t and $\mathcal{H}_s$ by $\mathcal{H}_h$		
Fault-Handler's Activities		
A_{fh} ::=	$\Lsh_{er} A_{th}$	(catch er)
$\mid$	$\Lsh A_{th}$	(catch all)
$\mid$	$A_{fh} + A_{fh}$	(choice of catches)

Handlers		
$\mathcal{H}$ ::=	$\mathcal{H}_s \mid \mathcal{H}_h \mid \mathcal{H}_p$	Handler
$\mathcal{H}_s$ ::=	$< F, C, T, E >$	Handler for Scope
$\mathcal{H}_h$ ::=	$< F, T, E >$	Handler for Handler
$\mathcal{H}_p$ ::=	$< F, E >$	Handler for process
Fault Handler		
F ::=	A_{fh}	(Fault Handler)
Compensation Handler		
C ::=	A_{ch}	(Compensation Handler)
Termination Handler		
T ::=	A_{ch}	(Termination Handler)
Event Handler		
E ::=		Event Handler
$\mid$	$c?m.A$	OnEvent (OnMessage/receive)
$\mid$	$waitFor(n).A$	OnEvent For (duration)
$\mid$	$waitUntil(t).A$	OnEvent Until (timeout)
$\mid$	$E + E$	Selective Event
$\mid$	$< \eta, E >$	(Event with attributes)

(Business Process)		
P ::=	$< \mathcal{E}_p, \mathcal{H}_p, A >$	(Business Process)

Composition		
D ::=		(Composition)
$\mid$	1	(Empty Composition)
$\mid$	$c!m$	(Message in Transit)
$\mid$	P	(Process)
$\mid$	$D \| D$	(Parallel Composition)

2.1 From AV-BPEL to BPEL

Hereafter, we give the relation between AV-BPEL and BPEL. As stated before, AV-BPEL is not a new language, it is just a concise representation of BPEL that is suitable for operational semantics. Therefore, the transformation here consists mainly on giving to any activity in AV-BPEL , its XML version in BPEL

The transformation from AV-BPEL to BPEL of simple activities is given in the table 2.

Table 2: Transformation of simple activities

AV-BPEL	**BPEL**
1	`<empty ></empty>`
0	`<exit></exit>`
$\Rsh_e$	`<throw faultName=`e`></throw>`
waitFor(d)	`<wait> <for>` d `</for></wait>`
waitUntil(t)	`<wait> <until>` t `</until></wait>`

The transformation from AV-BPEL to BPEL of communication activities is shown by the table 3.

Table 3: Transformation of communication activities

AV-BPEL	**BPEL**

$c!m$
$c = l.t.o$: channel name related to :
`partnerLink.portType.operation`
$m = (x_i^?, x_o)$
x_i is optional
	`<invoke partnerLink="`l`"`
	`portType="`t`"`
	`operation ="`o`"`
	`inputVariable="`x_i`" ?`
	`outputVariable="`x_o`">`

$(c?m)$
$c = l.t.o$: channel name
`partnerLink.portType.operation`
	`<receive partnerLink=`l
	`portType=`t
	`operation =`o
	`createInstance="no"`
	`variable="`m`"/>`

$(c?m)^+$
$c = l.t.o$: channel name
`partnerLink. portType. operation`
	`<receive partnerLink=`l
	`portType=`t
	`operation =`o
	`createInstance="yes"`
	`variable="`m`"/>`

$a \hookleftarrow x$ $c = l.t.o$: channel name `partnerLink.portType.operation`	`<replay partnerLink=`l `portType=`t `operation =`o `variable="`x`"/>`

The transformations from AV-BPEL to BPEL of structured activities is given in the table 4

The table 5 shows the transformation from AV-BPEL to BPEL of the Compensate Handlers Basic activities.

Finally, the transformation from AV-BPEL to BPEL of Scope, Fault Handlers, Compensation Handlers, Termination Handlers and Event Handlers is presented in the table 6.

Table 4: Transformation of structured activities

AV-BPEL	**BPEL**

$A_1.\ldots.A_n$	`<sequence>`
	A_1
	$\vdots$
	A_n
	`</sequence>`

$A_1\|\|\ldots\|\|A_n$	`<flow>`
	A_1
	$\vdots$
	A_n
	`</flow>`

$A \lhd b \rhd B$	`<if>`
	`<condition>` b `</condition>`
	A
	`<else>`
	B
	`</else>`
	`</if>`

b^*A	`<while>`
	`<condition>` b `</condition>`
	A
	`</while>`

A^*b	`<repeatUntil>`
	A
	`<condition>` b `</condition>`
	`</repeatUntil>`

$\sum_{i=1..n} c_i?x_i.A_i + \sum_{i=1..m} \mathsf{waitfor}(\mathsf{n_i}).B_i$ $+ \sum_{i=1..k} \mathsf{waitUntil}(\mathsf{T_i}).D_i$
 	
$c_i = l_i.t_i.o_i$: channel name related to `partnerLink.portType.operation`
	`<pick>`
	`<onMessage partnerLink=`l_1
	`portType=`t_1
	`operation =`o_1
	`variable="`x_1`"/>`
	A_1
	`</onMessage>`
	`<onMessage partnerLink=`l_n
	`portType=`t_n
	`operation =`o_n
	`variable="`x_n`"/>`
	A_n
	`</onMessage>`
	
	`<onAlarm>`
	`<for >`n_1`</for>`
	B_1
	`</onAlarm>`
	$\vdots$
	`<onAlarm>`
	`<for >`n_m`</for>`
	B_m
	`</onAlarm>`
	
	`<onAlarm>`
	`<until>`T_1`</until>`
	D_1
	`</onAlarm>`
	$\vdots$
	`<onAlarm>`
	`<until>`T_k`</until>`
	D_k
	`</onAlarm>`
	`</pick>`

Table 5: Transformation of Compensate Handlers Basic activities

AV-BPEL	**BPEL**
$\$$	`<compensate/>`
$\$_n$	`<compensateScope target="n"/>`
$\Lsh_e .S$	`<catch faultName=`e`>`S`</catch>`
$\Lsh .S$	`<catchall>`S`</catchall>`

Table 6: Transformation of Scope and Handlers

AV-BPEL	**BPEL**

$< \mathcal{E}, < F, C^?, T^?, E >, A >$
C and T are optional
	`<scope>` $\mathcal{E}$,
	`<faultHandlers>` F `</faultHandlers>`
	`<compensationHandler>` C `</compensationHandler>?`
	`<terminationHandler>` T `</terminationHandler>?`
	`<eventHandlers>` E `</eventHandlers>`
	A
	`</scope>`

3 Semantics

For the definition of AV-BPEL semantics, we adopt the following notations.

- Let $\mathcal{A}$ be the set of all activities. We use $A, A_1, A_2, \ldots$, as metavariables that range over $\mathcal{A}$.
- Let $\mathcal{D}$ be set of all possible compositions. We use $D, D_1, D_2 \ldots$ as metavariables that range over $\mathcal{D}$.
- We use M, N, R, M_1, N_1, R_1, $\ldots$ as metavariable to range over $\mathcal{D} \cup \mathcal{A}$.
- Let $\mathcal{L} = \mathcal{L}_N \cup \mathcal{L}_H$ be a set of labels used to show the evolution of activities and composition. It contains labels showing normal evolution of activities $\mathcal{L}_N = \{c!m, c?m, (c?m)^+, c!(m_1, m_2), c \hookleftarrow m, wait(t), x := v, \tau\}$ and others special ones used by handlers $\mathcal{L}_H = \{\$, \$_n, \Rsh_{er}, \Lsh, \Lsh_{er}\}$. We use α, α_1, α_2, $\ldots$ as metavariable to range over $\mathcal{L}$, α^N, α_1^N, α_2^N, $\ldots$ to range over $\mathcal{L}_N$ and α^H, α_1^H, α_2^H, $\ldots$ to range over $\mathcal{L}_H$.
- The definition of $\mathcal{E}$ is extended to a list of environments $\mathcal{E}_1 : \ldots : \mathcal{E}_n$, denoted by $\overrightarrow{\mathcal{E}}$. Also, we assume that $(\overrightarrow{\mathcal{E}} : \mathcal{E}')(x) = \mathcal{E}'(x)$ if x belongs to the domain of $\mathcal{E}'$, otherwise $(\overrightarrow{\mathcal{E}} : \mathcal{E}')(x) = \overrightarrow{\mathcal{E}}(x)$.

The operational semantics of AV-BPEL is defined by the transition relation $\longrightarrow \in (\mathcal{A} \cup \mathcal{D}) \times \mathcal{L} \times (\mathcal{A} \cup \mathcal{D})$. Table 7 gives some standard rules showing the evolution of activities. Also, it uses the equivalent relation $\equiv \in (\mathcal{A} \cup \mathcal{D}) \times (\mathcal{A} \cup \mathcal{D})$ defined by Table 8. The rules related to the evolution of processes and handlers are detailed separately later.

3.1 Semantics of activities

Once started, an activity needs only its environment $\mathcal{E}$ or its parent one, but there is no need for any handler until it finishes or an error occurs. The evolution of an activity A in a given environment $\mathcal{E}$ is specified by Table 7.

- **Equivalence rule:** The $(R_{\equiv})$ of Table 7 combined with the axioms of Table 8 simplify the presentation of the semantics. For instance, instead of writing a rule for $(M + N)$ and another for $(N + M)$, we can provide only one and implicitly get the other from the equivalence rules.

Table 7: Classical Rules for $\longrightarrow$.

Equivalence rule	Assignement rule	
$(R_{\equiv})\ \dfrac{P \equiv P_1 \quad P_1 \xrightarrow{\alpha} P_2 \quad P_2 \equiv P'}{P \xrightarrow{\alpha} P'}$	$(R_{:=})\ \dfrac{[\![exp]\!]_{\mathcal{E}} = v}{< \mathcal{E}, x := e > \xrightarrow{x:=v} < \mathcal{E}[x \mapsto v], 1 >}$	
Delay rules		
$(R_T^f)\ \dfrac{t_c = \mathsf{Clock}()}{< \mathcal{E}, \mathsf{waitFor}(n) > \xrightarrow{\tau} < \mathcal{E}, \mathsf{waitUntil}(n + t_c) >}$	$(R_T^u)\ \dfrac{t \geq \mathsf{Clock}()}{< \mathcal{E}, \mathsf{waitUntil}(t) > \xrightarrow{wait(t)} < \mathcal{E}, 1 >}$	
Communication related rules		
$(R_{\hookleftarrow})\ \dfrac{\Box}{< \mathcal{E}, c \hookleftarrow m > \xrightarrow{c \hookleftarrow m} < \mathcal{E}', 1 >}$	$(R_!)\ \dfrac{\Box}{< \mathcal{E}, c!m > \xrightarrow{c!m} < \mathcal{E}', 1 >}$	
$(R_?)\ \dfrac{< \mathcal{E}, A > \overset{c'?x}{\twoheadrightarrow} < \mathcal{E}, A' >}{< \mathcal{E}, A > \Vert c!m \xrightarrow{c?m} < \mathcal{E}'[x \mapsto m], A' >}$	$(R_?)^+\ \dfrac{< \mathcal{E}, A > \overset{(c'?x)^+}{\twoheadrightarrow} < \mathcal{E}, A' >}{< \mathcal{E}, A >) \Vert c!m \xrightarrow{(c?m)^+} < \mathcal{E}'[x \mapsto m], 1 >}$	
Decompostion rules		
$(R_+)\ \dfrac{< \mathcal{E}, A_1 > \xrightarrow{\alpha} < \mathcal{E}', A_1' >}{< \mathcal{E}, A_1 + A_2 > \xrightarrow{\alpha} < \mathcal{E}', A_1' >}$	$(R_.)\ \dfrac{< \mathcal{E}, A_1 > \xrightarrow{\alpha} < \mathcal{E}', A_1' >}{< \mathcal{E}, A_1.A_2 > \xrightarrow{\alpha} < \mathcal{E}', A_1'.A_2 >}$	
$(R_*^l)\ \dfrac{< \mathcal{E}, A > \xrightarrow{\alpha} < \mathcal{E}', A' >}{< \mathcal{E}, b^*A > \xrightarrow{\alpha} < \mathcal{E}', A'.b^*A >} [\![b]\!]_{\mathcal{E}} = \mathsf{tt}$	$(R_*^d)\ \dfrac{< \mathcal{E}, A > \xrightarrow{\alpha} < \mathcal{E}', A' >}{< \mathcal{E}, A^*b > \xrightarrow{\alpha} < \mathcal{E}', A'.b^*A >}$	
$(R_{\triangleleft})\ \dfrac{< \mathcal{E}, A_1 > \xrightarrow{\alpha} < \mathcal{E}', A_1' >}{< \mathcal{E}, A_1 \triangleleft b \triangleright A_2 > \xrightarrow{\alpha} < \mathcal{E}', A_1' >} [\![b]\!]_{\mathcal{E}} = \mathsf{tt}$	$(R_{\triangleright})\ \dfrac{< \mathcal{E}, A_2 > \xrightarrow{\alpha} < \mathcal{E}', A_2' >}{< \mathcal{E}, A_1 \triangleleft b \triangleright A_2 > \xrightarrow{\alpha} < \mathcal{E}', A_2' >} [\![b]\!]_{\mathcal{E}} = \mathsf{ff}$	
$(R_{\Vert})\ \dfrac{< \mathcal{E}, A_1 > \xrightarrow{\alpha} < \mathcal{E}', A_1' >}{< \mathcal{E}, A_1 \Vert A_2 > \xrightarrow{a} < \mathcal{E}', A_1' \Vert A_2 >}$	$(R_!^s)\ \dfrac{< \mathcal{E}, c!m_1 > \xrightarrow{c!m_1} < \mathcal{E}', 1 >}{< \mathcal{E}, c!(m_1, m_2) > \xrightarrow{c!m_1} < \mathcal{E}', c?m_2 >}$	
Environement propagation rule	**Compensation rules**	
$(R_{\mathcal{E}})\ \dfrac{<< \overrightarrow{\mathcal{E}_1} : \mathcal{E}_2, A > \xrightarrow{\alpha} << \overrightarrow{\mathcal{E}_1}' : \mathcal{E}_2', A' >}{< \overrightarrow{\mathcal{E}_1}, < \mathcal{E}_2, A >> \xrightarrow{\alpha} << \overrightarrow{\mathcal{E}_1}', < \mathcal{E}_2', A' >>}$	$(R_{\$_n})\ \dfrac{\Box}{< \mathcal{E}, \$_n > \xrightarrow{\$_n} < \mathcal{E}, 1 >}$	$(R_{\$})\ \dfrac{\Box}{< \mathcal{E}, \$ > \xrightarrow{\$} < \mathcal{E}, 1 >}$
Error related rules		
$(R_{\Rsh_{er}})\ \dfrac{\Box}{< \mathcal{E}, \Rsh_{er} > \xrightarrow{\Rsh_{er}} < \mathcal{E}, 1 >}$	$(R_{\Lsh_{er}})\ \dfrac{\Box}{< \mathcal{E}, \Lsh_{er}\ S > \xrightarrow{\Lsh_{er}} < \mathcal{E}, S >}$	$(R_{\Lsh})\ \dfrac{\Box}{< \mathcal{E}, \Lsh\ S > \xrightarrow{\Lsh} < \mathcal{E}, S >}$

Table 8: Axioms of BPEL.

$$\begin{array}{rll}
M + N \equiv & N + M & (AX_1) \\
M \| N \equiv & N \| M & (AX_2) \\
(M + N) + R \equiv & M + (N + R) & (AX_3) \\
(M \| N) \| R \equiv & M \| (N \| R) & (AX_4) \\
0 + M \equiv & M & (AX_5) \\
1.M \equiv & M & (AX_6) \\
0 \| M \equiv & M & (AX_7) \\
1 \| M \equiv & M & (AX_8) \\
A_1 \equiv A_2 \Rightarrow & < \mathcal{E}, A_1 > \equiv < \mathcal{E}, A_2 > & (AX_9) \\
< \mathcal{E}, < F, E >, 1 > \equiv & 1 & (AX_{10}) \\
\sum_{i=1}^{n} (a_i?m_i.A_i)^+ \equiv & \sum_{i=1}^{n} (a_i?m_i)^+.A_i & (AX_{11})
\end{array}$$

- **Assignment rules:** The rule $(R_{:=})$ of Table 7 gives the semantics of an assignment. We assume that $[\![exp]\!]_{\mathcal{E}}$ is an existing function that evaluates exp in $\mathcal{E}$ and returns a value.
- **Delay rules:** We suppose that we have an external clock, denoted by $\mathsf{clock}()$, that return current time. The rule (R_T^f) of Table 7 allows to transforming $waitFor(n)$ into $waitUntil(n + t_c)$, where the time t_c is the content of the $Clock()$. The rule (R_T^u) of Table 7 allows to wait until the time t arrives.
- **Communication rules:** Communication is handled by rules $(R_{\hookleftarrow})$, $(R_!)$, $(R_?)$ and $(R_?)^+$ of Table 7. Sending a message m on a channel c (rules $(R_{\hookleftarrow})$ and $(R_!)$) is not a blocking action and it produces a floating message (meaning that

the message has not been received yet). Receiving a message m on a channel c (rules $(R_?)$ and $(R_?^+)$), is, however, always blocking until seeing a floating message on the appropriate channel. To know if a process is ready to receive a message, we introduce the relation $\twoheadrightarrow$ defined by Table 9. Basically, only process that are ready to receive can evolve with $\twoheadrightarrow$.

Table 9: Rules for $\twoheadrightarrow$.

Definition of $\twoheadrightarrow$
$(R_{\equiv}^{\twoheadrightarrow})\ \dfrac{A \equiv A_1 \quad A_1 \overset{\alpha}{\twoheadrightarrow} A_2 \quad A_2 \equiv A'}{A \overset{\alpha}{\twoheadrightarrow} A'}$
$(R_?^{+\twoheadrightarrow})\ \dfrac{\square}{<\mathcal{E},(c_i?m_i)^+> \overset{(c_i?m_i)^+}{\twoheadrightarrow} <\mathcal{E},1>}$
$(R_?^{\twoheadrightarrow})\ \dfrac{\square}{<\mathcal{E},c?m> \overset{c?m}{\twoheadrightarrow} <\mathcal{E},1>}$
$(R_{\|}^{\twoheadrightarrow})\ \dfrac{<\mathcal{E},A_1> \overset{\alpha}{\twoheadrightarrow} <\mathcal{E}',A_1'>}{<\mathcal{E},A_1\|A_2> \overset{\alpha}{\twoheadrightarrow} <\mathcal{E},A_1'\|A_2>}$
$(R_*^{l\twoheadrightarrow})\ \dfrac{<\mathcal{E},A> \overset{\alpha}{\twoheadrightarrow} <\mathcal{E},A'>}{<\mathcal{E},b^*A> \overset{\alpha}{\twoheadrightarrow} <\mathcal{E},A'.b^*A>} [\![b]\!]_{\mathcal{E}} = \mathsf{tt}$
$(R_*^{d\twoheadrightarrow})\ \dfrac{<\mathcal{E},A> \overset{\alpha}{\twoheadrightarrow} <\mathcal{E},A'>}{<\mathcal{E},A^*b> \overset{\alpha}{\twoheadrightarrow} <\mathcal{E},A'.b^*A>}$
$(R_{\triangleleft}^{\twoheadrightarrow})\ \dfrac{<\mathcal{E},A_1> \overset{\alpha}{\twoheadrightarrow} <\mathcal{E},A_1'>}{<\mathcal{E},A_1 \triangleleft b \triangleright A_2> \overset{\alpha}{\twoheadrightarrow} <\mathcal{E},A_1'>} [\![b]\!]_{\mathcal{E}} = \mathsf{tt}$
$(R_{\triangleright}^{\twoheadrightarrow})\ \dfrac{<\mathcal{E},A_2> \overset{\alpha}{\twoheadrightarrow} <\mathcal{E},A_2'>}{<\mathcal{E},A_1 \triangleleft b \triangleright A_2> \overset{a}{\twoheadrightarrow} <\mathcal{E},A_2'>} [\![b]\!]_{\mathcal{E}} = \mathsf{ff}$
$(R_{+}^{\twoheadrightarrow})\ \dfrac{<\mathcal{E},A_1> \overset{\alpha}{\twoheadrightarrow} <\mathcal{E},A_1'>}{<\mathcal{E},A_1+A_2> \overset{\alpha}{\twoheadrightarrow} <\mathcal{E},A_1'>}$
$(R_{.}^{\twoheadrightarrow})\ \dfrac{<\mathcal{E},A_1> \overset{\alpha}{\twoheadrightarrow} <\mathcal{E},A_1'>}{<\mathcal{E},A_1.A_2> \overset{\alpha}{\twoheadrightarrow} <\mathcal{E},A_1'.A_2>}$

- **Decomposition rules:** The rule (R_+) is for the choice (+), the rule $(R_.)$ is for the sequential composition. The two rules (R_*^l) and (R_*^d) are for $''while''$ and $''do\ until''$ respectively. The rules $(R_{\triangleleft})$ and $(R_{\triangleright})$ respectively present the two cases of $''If\ then\ else''$. $(R_{\|})$ is the rule of the flow (||) operator and finally the rule $(R_!^s)$ is for processing a communication.

- **Environment propagation rule:** $(R_{\mathcal{E}})$ shows how environments are inherited (for read and write) by child scopes from their parents.

3.2 Scope's Semantics

A process or a scope has its principal activity and different handlers. To know the part of the process (the main activity or a handler) is being executed, we introduce a new notion called mode. Basically, we have a default mode and three other special ones denoted by $\mathbb{F}$, $\mathbb{C}$ or $\mathbb{T}$ as following:

- A scope $<\mathcal{E},\mathcal{H},A>$ without any special annotation is in its default mode, meaning that it is running its principal activity A.
- A scope in mode $\mathbb{F}$, denoted by $<\mathcal{E},\mathcal{H},A>^{\mathbb{F}}$, means that it is running the code of its fault handler.
- A scope in mode $\mathbb{C}$, denoted by $<\mathcal{E},\mathcal{H},A>^{\mathbb{C}}$, means that it is running the code of its compensation handler.
- A scope in mode $\mathbb{T}$, denoted by $<\mathcal{E},\mathcal{H},A>^{\mathbb{T}}$, means that it is running the code of its termination handler.
- The event handler does not need any special mode since its code is executed within the principal activity.

There are some actions or situations that allow us to switch from one mode to another. For example, when a *throw* action is executed, the process goes to the fault mode.

To simplify the presentation of the remaining rules, we denote by $<\mathcal{E},<F,C^?,T^?,E>,A>$ a scope or a process, where the termination and the compensation handlers are optional (indicated by the character "?"). But only the following possibilities exist:

- $<\mathcal{E},<F,E>,A>$: a process has a scope with only two handlers (Fault and Event handlers).
- $<\mathcal{E},<F,T,E>,A>$: a scope in FCT-handlers could not have a compensation handler.
- $<\mathcal{E},<F,C,T,E>,A>$: a scope of a normal activity has all the handlers.

Fault and compensation handlers have their implicit values when they have not been explicitly specified:

- the default fault handler is: $\Lsh(\$.\Rsh_{\mathsf{er}})$
- the default compensate and fault handlers is: $

3.2.0.1 • *Normal execution of a scope* During the normal execution of a scope, we need only its activity and its environment and there is no need for its handlers. This fact is formalized by the rule (R_S^1) of Table 10.

Table 10: Rules for scope and process

Rules for Scope	
$(R^1_S)\ \dfrac{<\mathcal{E},A>\xrightarrow{\alpha^N}<\mathcal{E}',A'>}{<\mathcal{E},<F,C^?,T,E>,A>\xrightarrow{\alpha^N}<\mathcal{E}',<F,C^?,T,E>,A'>}$	$(R^2_S)\ \dfrac{\square}{<\mathcal{E},<F/\Delta,C^?,T,E>,<\mathcal{E}',F',C',T',E',1>>\xrightarrow{\tau}<\mathcal{E},<F/(<\mathcal{E}',C'>.\Delta),C^?,T,E>,1>}$
$(R^3_S)\ \dfrac{\square}{<\mathcal{E},<F/\Delta,C^?,T,E>,<\mathcal{E}',F',C',T',E',1>.Q>\xrightarrow{\tau}<\mathcal{E},<F/(<\mathcal{E}',C'>.\Delta),C^?,T,E>,Q>}$	
$(R^4_S)\ \dfrac{\square}{<\mathcal{E},<F/\Delta,C^?,T,E>,<\mathcal{E}',F',C',T',E',1>.Q\parallel R>\xrightarrow{\tau}<\mathcal{E},<F/(<\mathcal{E}',C'>.\Delta),C^?,T,E>,Q\parallel R>}$	
$(R^5_S)\ \dfrac{\square}{<\mathcal{E},<F/\Delta,C^?,T,E>,<\mathcal{E}',F',C',T',E',1>.Q+R>\xrightarrow{\tau}<\mathcal{E},<F/(<\mathcal{E}',C'>.\Delta),C^?,T,E>,Q>}$	
Rules for Process	
$(R_P)\ \dfrac{<\mathcal{E},A>\xrightarrow{\alpha}<\mathcal{E}',A'>}{<\mathcal{E},<F,E>,A>\ \xrightarrow{\alpha}<\mathcal{E}',<F,E>,A'>}\quad \alpha\neq c!m \text{ and } \alpha\neq c\hookleftarrow m$	$(R^{\parallel}_P)\ \dfrac{D_1\xrightarrow{\alpha}D_1'}{D_1\parallel D_2\ \xrightarrow{\alpha}D_1'\parallel D_2}$
$(R^+_P)\ \dfrac{<\mathcal{E},A>\parallel c!m\xrightarrow{(c?m)^+}<\mathcal{E}',A'>}{<\mathcal{E},<F,E>,A>\parallel c!m\ \xrightarrow{(c?m)^+}\ <\mathcal{E}',<F,E>,A'>\parallel<\mathcal{E},<F,E>,A>}$	$(R^?_P)\ \dfrac{<\mathcal{E},A>\parallel c!m\xrightarrow{c?m}<\mathcal{E}',A'>}{<\mathcal{E},<F,E>,A>\parallel c!m\ \xrightarrow{c?m}\ <\mathcal{E}',<F,E>,A'>}$
$(R^!_P)\ \dfrac{<\mathcal{E},A>\xrightarrow{c!m}<\mathcal{E}',A'>}{<\mathcal{E},<F,E>,A>\ \xrightarrow{c!m}<\mathcal{E}',<F,E>,A'>\parallel c!m}$	$(R^{\hookleftarrow}_P)\ \dfrac{<\mathcal{E},A>\xrightarrow{c\hookleftarrow m}<\mathcal{E}',A'>}{<\mathcal{E},<F,E>,A>\ \xrightarrow{c\hookleftarrow m}<\mathcal{E}',<F,E>,A'>\parallel c!m}$

3.2.0.2 • *Termination of a scope* When a child scope terminates with success, its compensation activity is stored for an eventual future compensation by its parents. Storing this scope is done simply by placing it within the fault handler of its parent using the notation F/Δ. The compensation backup is managed by the rules (R^2_S), (R^3_S), (R^4_S), and (R^5_S) of Table 10 according to the following policy.

- We assume that the initial value of Δ is 1 and $F/1$ is abbreviated by F.
- A compensation handler can use the context (environment) of its associated scope as it has been left. For this reason, the environment of the scope is also stored.
- When a scope terminates, its event handler is uninstalled.
- The termination handler and the fault handler are useful only during the execution of a scope (in case that an error occurs). But, when the scope terminates successfully, these handlers are no more utile.
- Compensation handlers are stored in the reverse order since, once needed, they will be executed in LIFO (Last In First Out) to reverse the effects of their corresponding activities.

3.2.0.3 • *Simple evolution of a process* A process $<\mathcal{E},<F,E>,A>$ evolves with a non-sending action using the rule (R_P) of Table 11.

3.2.0.4 • *Communication between Process* The syntax of BPEL gives the possibility of creating an instance of a process when a message is received as shown by the rule (R^+_P) of Table 11. The rules (R^+_P) and $(R^?_P)$ of Table 11 show how a floating message is consumed by an inbound action. The rules $(R^{\hookleftarrow}_P)$ and $(R^!_P)$ of Table 11 show how a floating message is produced by an outbound action.

3.2.0.5 • *Termination of a process* When the principal activity of a process terminates, the process itself terminates and its fault handler and event handler are uninstalled. This property is captured by axiom AX_{10} of Table 8.

3.2.0.6 • *Composition Rule* The others parallel compositions of processes are handled by the rule $(R^{\parallel}_P)$.

3.3 Handler's Semantics

To define the semantics of a handler, we need to clarify, when it is activated, how it runs its activity, what happens when an error occurs during its execution and when it terminates. Rules of Table 11 capture these situations.

3.3.0.1 • *Fault handler*

- **Starting a fault mode**: Each error thrown by a principal activity should be caught by the fault handler of the same scope, then we switch to the fault mode $\mathbb{F}$. This fact is captured by the rule (R^s_F) of Table 11. If a throw $\Rsh_{er}$ doesn't have a dedicated catch in the fault handler, then it should be handled by the "catch all" activity as shown by the rule $(R^{s'}_F)$ of Table 11.
- **Running the activities of a fault handler**: A fault handler can run its activity only after the termination handler has finished (or if the termination handler is not present for the case of a process). Executing the fault handler action is done like any other activity as shown by rule (R_F) of Table 11.

Table 11: Rules for Handlers.

Fault Handler Rules
$(R_F^s)\ \dfrac{<\mathcal{E},A>\overset{\Rsh_{er}}{\longrightarrow}<\mathcal{E}',A'> \quad <\mathcal{E},F>\overset{\Lsh_{er}}{\longrightarrow}<\mathcal{E},F'>}{<\mathcal{E},<F,C^?,T^?,E>,A>\overset{\Rsh_{er}}{\longrightarrow}<\mathcal{E}[\mathsf{err}\mapsto er],<F',C^?,T^?,1>,1>^{\mathbb{F}}}$ $(R_F^{s'})\ \dfrac{<\mathcal{E},A>\overset{\Rsh_{er}}{\longrightarrow}<\mathcal{E}',A'> \quad <\mathcal{E},F>\overset{\Lsh_{er}}{\not\longrightarrow} \quad <\mathcal{E},F>\overset{\Lsh}{\longrightarrow}<\mathcal{E},F'>}{<\mathcal{E},<F,C^?,T^?,E>,A>\overset{\Rsh_{er}}{\longrightarrow}<\mathcal{E}'[\mathsf{err}\mapsto er],<F',C^?,T^?,1>,1>^{\mathbb{F}}}$
$(R_F)\ \dfrac{<\mathcal{E},F>\overset{\alpha^N}{\longrightarrow}<\mathcal{E}',F'>}{<\mathcal{E},<F,C^?,1^?,1>,1>^{\mathbb{F}}\overset{\alpha^N}{\longrightarrow}<\mathcal{E}',<F',C^?,1^?,1>,1>^{\mathbb{F}}}$ $(R_F^t)\ \dfrac{\Box}{<\mathcal{E},<<\mathcal{E}_f,\mathcal{H}_f,1>,C^?,1^?,1>,1>^{\mathbb{F}}\overset{\tau}{\longrightarrow}1}$ $(R_F^e)\ \dfrac{<\mathcal{E},F>\overset{\Rsh_{er}}{\longrightarrow}<\mathcal{E}',F'>}{<\mathcal{E},<F,C^?,1^?,1>,1>^{\mathbb{F}}\overset{\Rsh_{er}}{\longrightarrow}1}$
Compensation Handler Rules
$(R_C^s)\ \dfrac{<\mathcal{E},F>\overset{\$}{\longrightarrow}<\mathcal{E},F'>}{<\mathcal{E},<F,C^?,1^?,1>,1>^{\mathbb{F}}\overset{\$}{\longrightarrow}<\mathcal{E},<F',C^?,1^?,1>,1>^{\mathbb{C}}}$ $(R_C^{s'})\ \dfrac{<\mathcal{E},T>\overset{\$}{\longrightarrow}<\mathcal{E},T'>}{<\mathcal{E},<F,C^?,T,1>,1>^{\mathbb{T}}\overset{\$}{\longrightarrow}<\mathcal{E},<F,C^?,T',1>,1>^{\mathbb{C}}}$
$(R_C^{s_n})\ \dfrac{<\mathcal{E},F>\overset{\$_n}{\longrightarrow}<\mathcal{E},F'>}{<\mathcal{E},<F,C^?,1^?,1>,1>^{\mathbb{F}}\overset{\$_n}{\longrightarrow}<\mathcal{E},<F',C^?,1^?,1>,1>^{\mathbb{C}}}$ $(R_C^{s'_n})\ \dfrac{<\mathcal{E},T>\overset{\$_n}{\longrightarrow}<\mathcal{E},T'>}{<\mathcal{E},<F,C^?,T,1>,1>^{\mathbb{T}}\overset{\$_n}{\longrightarrow}<\mathcal{E},<F,C^?,T',1>,1>^{\mathbb{C}}}$
$(R_C)\ \dfrac{<\mathcal{E},\Delta>\overset{\alpha^N}{\longrightarrow}<\mathcal{E}',\Delta'>}{<\mathcal{E},<F/\Delta,C^?,T^?,1>,1>^{\mathbb{C}}\overset{\alpha^N}{\longrightarrow}<\mathcal{E}',<F/\Delta',C^?,T^?,1>,1>^{\mathbb{C}}}$ $(R'_C)\ \dfrac{\Box}{<\mathcal{E},<F/<\mathcal{E}',<\mathcal{E}_c,\mathcal{H}_c,1>>.\Delta,C^?,T^?,1>,1>^{\mathbb{C}}\overset{\tau}{\longrightarrow}<\mathcal{E},<F/\Delta,C^?,T^?,1>,1>^{\mathbb{C}}}$
$(R_C^t)\ \dfrac{\Box}{<\mathcal{E},<F/<\mathcal{E}',<\mathcal{E}_c,\mathcal{H}_c,1>>,C^?,T^?,1>1>^{\mathbb{C}}\overset{\tau}{\longrightarrow}<\mathcal{E},<F/1,C^?,T^?,1>,1>^{\mathbb{F}}}$ $(R_C^e)\ \dfrac{<\mathcal{E},\Delta>\overset{\Rsh_{er}}{\longrightarrow}<\mathcal{E}',\Delta'>}{<\mathcal{E},<F/\Delta,C^?,T^?,1,1>^{\mathbb{C}}\overset{\tau}{\longrightarrow}<\mathcal{E}',<F/1,C^?,T^?,1>,1>^{\mathbb{F}}}$
Termination Handler Rules
$(R_T^s)\ \dfrac{\Box}{<\mathcal{E},<F,C^1,T,1>,1>^{\mathbb{F}}\overset{\tau}{\longrightarrow}<\mathcal{E},<F,C^1,T,1>,1>^{\mathbb{T}}}\ T\neq 1$ $(R_T)\ \dfrac{<\mathcal{E},T>\overset{a}{\longrightarrow}<\mathcal{E}',T'>}{<\mathcal{E},<F,C^1,T,1>,1>^{\mathbb{T}}\overset{a}{\longrightarrow}<\mathcal{E}',<F,C^1,T',1>,1>^{\mathbb{T}}}$
$(R_T^t)\ \dfrac{\Box}{<\mathcal{E},<F,C^1,1,1>,1>^{\mathbb{T}}\overset{\tau}{\longrightarrow}<\mathcal{E},<F,C^1,1,1>,1>^{\mathbb{F}}}\ T\neq 1$ $(R_T^e)\ \dfrac{<\mathcal{E},T>\overset{\Rsh_{er}}{\longrightarrow}<\mathcal{E}',T'>}{<\mathcal{E},F,C^1,T,1,1>^{\mathbb{T}}\overset{\tau}{\longrightarrow}<\mathcal{E}',F,C^1,1,1,1>^{\mathbb{F}}}$
Event Handler Rules
$(R_E^s)\ \dfrac{<\mathcal{E},E>\overset{wait(t)}{\longrightarrow}<\mathcal{E}',E'>}{<\mathcal{E},F,C^?,T^?,E,A>\overset{wait(t)}{\longrightarrow}<\mathcal{E}',F,C^?,T^?,E,E'\Vert A>}$ $(R_E^{s'})\ \dfrac{<\mathcal{E},E>\Vert D\overset{c?m}{\longrightarrow}<\mathcal{E}',E'>\Vert D'}{<\mathcal{E},<F,C^?,T^?,E>,A>\Vert D\overset{c?m}{\longrightarrow}<\mathcal{E}',<F,C^?,T^?,E>,E'\Vert A>\Vert D'}$
$(R_E^e)\ \dfrac{<\mathcal{E},E>\Vert c!m\overset{\Rsh_{er}}{\longrightarrow}A' \quad <\mathcal{E},F>\overset{\Lsh_{er}}{\longrightarrow}<\mathcal{E},F'>}{<\mathcal{E},<F,C^?,T^?,E>,A>\Vert c!m\overset{\Rsh_{er}}{\longrightarrow}<\mathcal{E}[\mathsf{err}\mapsto e],<F',C^?,T^?,1>,1>^{\mathbb{F}}}$ $(R_E^{e'})\ \dfrac{<\mathcal{E},E>\Vert c!m\overset{\Rsh_{er}}{\longrightarrow}A' \quad <\mathcal{E},F>\overset{\Lsh_{er}}{\not\longrightarrow} \quad <\mathcal{E},F>\overset{\Lsh}{\longrightarrow}<\mathcal{E},F'>}{<\mathcal{E},<F,C^?,T^?,E>,A>\Vert c!m\overset{\Rsh_{er}}{\longrightarrow}<\mathcal{E}[\mathsf{err}\mapsto e],<F',C^?,T^?,1>,1>^{\mathbb{F}}}$

- **Finishing a fault handler**: If a fault handler terminates normally, the scope is considered to be ended abnormally and we don't install its compensation handler in its parent scope as shown by rule (R_F^t) of Table 11.
- **Handling errors inside a fault handler** : When an error occurs in the fault handler, the scope is considered to have ended abnormally and we don't install its compensation handler in its parent scope. The fault is, however, propagated to the parent scope (enclosing scope) as shown by rule (R_F^e) of Table 11.

3.3.0.2 • Compensation handler execution

- **Starting a compensation handler**: Both fault handler and termination handler can activate the compensation handler by execution the activity \$ or $\$_n$ as shown by the rules (R_C^s), $(R_C^{s'})$, $(R_C^{s_n})$ and $(R_C^{s'_n})$ of Table 11.
- **Running the activities of a compensation handler**: The activity of a compensation handler is executed in a compensation mode according to the rule (R_C) of Table 11. Once a compensation finishes its activity, we move to the next one as shown by (R'_C) of Table 11.
- **Finishing a compensation handler**: Once the compensation has successfully terminated, the control is turned back to the calling activity (fault or termination handler). The calling activity is the termination handler if $T \neq 1$, otherwise it is the fault handler. Even, if we turn the control back to the fault handler all the time, we get the right execution. In fact, the fault handler will automatically transfer the execution to the termination handler if this second hasn't terminate yet as shown by the rule (R_C^t) of Table 11.
- **Handling errors inside a compensation handler** : If an error occurs during the execution of a compensation handler, then it will not be reported, its remaining activity is ignored and the control is turned back to the fault handler as shown by (R_C^e) of Table 11.

3.3.0.3 • Termination handler execution

- **Starting a termination handler**: If we are in a fault mode, and the termination activity is not empty ($T \neq 1$), then the termination handler is automatically activated as shown by the rule (R_T^s) of Table 11.
- **Running the activities of a termination handler**: The activity of a termination handler is executed in a termination mode as illustrated by (R_T) of Table 11.
- **Finishing a termination handler**: Once the termination handler has successfully finished, the control is turned back to the calling activity (the fault handler) as shown by the rule (R_T^t) of Table 11.
- **Handling errors inside a termination handler** : Any error occurs during the execution of a termination handler will not be reported. Also, the remaining activity of this termination handler is ignored and the control is turned back to the calling activity as shown by the rule (R_T^e) of Table 11.

3.3.0.4 • Event Handler

- **Starting an event handler**: An event handler can start either after a time event ((R^s_E) of Table 11) or after receiving a message ($(R^{s'}_E)$ of Table 11). The residual part of the event handler is executed in parallel with the principal activity of the scope or the process.
- **Running the activities of an event handler**: We don't need any special rule, since, unlike FCT-handlers, the activities of an event handler are executed in parallel with the principal activity of the current scope.
- **Finishing the activities of an event handler**: We don't need any special rule, since, unlike FCT-handlers, the activities of an event handler are executed in parallel with the principal activity of the current scope.
- **Handling error inside an event handler**: When a fault occurs during `OnEvent`, it will be handled by the fault handler of the scope as shown by the rules (R^3_E) and (R^4_E) of Table 11.

4 Applications

In this section, we give two examples to better understand the operational rules of the semantics. The first one is a real-life case study to showcase the practical use of our approach, when the second one is related to error handling and compensation.

4.1 Travel Agency

We have chosen a simple example of a Travel Agency as shown by Figure 1 to explain and advocate the use of AV-BPEL to describe and formalize Travel Agency BPEL process.

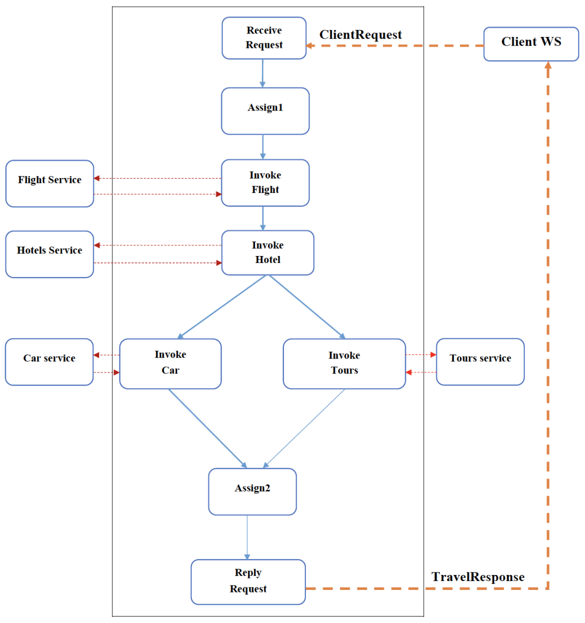

Fig. 1: Travel Agency BPEL Process

$TravelProcess$, this is the process framed in black in the figure 1, is a composition of services that offers a trip planning depending upon the needs of the client. On the reception of a request, initialized by the client of the service, a response is returned.

A typical use case could be the booking of a trip with flight tickets together with a hotel, a car for the whole stay or just famous sightseeing trips. These are the services (partners) that $TravelProcess$ interacts with. Depending upon the availability, a suitable travel is returned to the client.

Here is the part of the BPEL code corresponding to the example of Travel Agency. For the sake of simplicity, the correlation mechanism will not be dressed in this example.

```
<?xml version="1.0" encoding="UTF-8"?>
<process name="TravelProcess" >
      <partnerLinks>
            <partnerLink name="client"
                  partnerLinkType="clientPLT"/>
            <partnerLink name="flight"
                  partnerLinkType="flightPLT"/>
            <partnerLink name="car"
                   partnerLinkType="carPLT"/>
            <partnerLink name="hotel"
                  partnerLinkType="hotelPLT"/>
            <partnerLink name="tours"
                  partnerLinkType="toursPLT"/>
      </partnerLinks>
      <partners name="client">
      </partners>
      <variables>
            <variable messageType="clientReqType" name="clientReq"/>
            <variable messageType="clientRespType" name="clientResp"/>
            <variable messageType="lightReqType" name="flightReq"/>
            <variable messageType="flightResType" name="flightRes"/>
            <variable messageType="hotelReqType" name="hotelReq"/>
            <variable messageType="hotelRespType" name="hotelResp"/>
            <variable messageType="carReqType" name="carReq"/>
            <variable messageType="carRespType" name="carResp"/>
            <variable messageType="toursReqType" name="toursReq"/>
            <variable messageType="toursRespType" name="toursResp"/>
      </variables>
<!-- Activities -->
      <sequence>
            <receive name="Receive␣Request"
                        partnerLink="client"
                        portType="travel"
                        operation="search"
                        variable="clientReq"
                        createInstance="yes"/>
            <assign name="Assign␣Requests">
                  <copy>
                        <from> clientReq.itinerary </from>
                        <to> flightReq.itinerary </to>
                  </copy>
                  <copy>
                        <from> clientReq.dates </from>
                        <to> carReq.dates </to>
                  </copy>
                  <copy>
                        <from> clientReq.city </from>
                        <to> carReq.city </to>
                  </copy>
                  <copy>
                        <from> clientReq.dates </from>
                        <to> hotelReq.dates </to>
                  </copy>
                  <copy>
                        <from> clientReq.city </from>
                        <to> hotelReq.city </to>
                  </copy>
                  <copy>
                        <from> clientReq.dates </from>
                        <to> toursReq.dates </to>
                  </copy>
                  <copy>
                        <from> clientReq.city </from>
                        <to> toursReq.city </to>
                  </copy>
            </assign>
      <!-- Invoke Flight -->
            <invoke>
                  partnerLink="flight"
                  portType="book"
                  operation ="search"
                  inputVariable= "flightResp"
                  outputVariable= "flightReq"
            </invoke>
      <!-- Invoke Hotel -->
            <invoke>
                  partnerLink="hotel"
                  portType="book"
                  operation ="search"
                  inputVariable="hotelResp"
                  outputVariable="hotelReq"
            </invoke>
            <flow>
            <!-- Invoke Rent a car -->
                  <invoke>
```

```
                        partnerLink="car"
                        portType="rent"
                        operation ="search"
                        inputVariable="carResp"
                        outputVariable="carReq"
                  </invoke>
            <!-- Invoke Tours -->
                  <invoke>
                        partnerLink="toor"
                        portType="services"
                        operation ="getInfo"
                        inputVariable= "toursResp"
                        outputVariable= "toursReq"
                  </invoke>
            </flow>
      <assign name="Assign␣Responses">
                  <copy>
                        <from> flightResp </from>
                        <to> clientResp.flightPrice </to>
                  </copy>
                  <copy>
                        <from> hotelResp </from>
                        <to> clientResp.hotelPrice</to>
                  </copy>
                  <copy>
                        <from> carResp </from>
                        <to> clientResp.carPrice </to>
                  </copy>
                  <copy>
                        <from> toursResp </from>
                        <to> clientResp.toursPrice </to>
                  </copy>
      </assign>
      <!-- Replay Request -->
            <reply name ="Replay␣Request"
                  partnerLink="client"
                  portType="travel"
                  operation ="results"
                  variable="clientResp"/>
      </sequence>
</process>
```

To keep the Travel Agency example simple, we removed some information like the WSDL files (containing messages type, partner links, correlations, etc.), the handlers and the correlation data. Now, we give its AV-BPEL form denoted by $TravelProcess = <\mathcal{E}, \mathcal{H}, A>$. If we ignore the handlers H as for the XML form, we obtain:

$\mathcal{E}$ <

pLinks

$$\begin{aligned} client &: \text{clientPLT}, \\ flight &: \text{flightPLT}, \\ car &: \text{carPLT}, \\ hotel &: \text{hotelPLT}, \\ toors &: \text{toorsPLT} \end{aligned}$$

Var

$$\begin{aligned} clientReq &: \text{clientReqType}, \\ clientResp &: \text{clientRespType}, \\ flightReq &: \text{flightReqType}, \\ flightResp &: \text{flightRespType}, \\ hotelReq &: \text{hotelReqType}, \\ hotelResp &: \text{hotelRespType} \\ carReq &: \text{carReqType}, \\ carResp &: \text{carRespType}, \\ toursReq &: \text{toursReqType}, \\ toursResp &: \text{toursRespType} \end{aligned}$$

>

A: <

$<$ **name** $\mapsto$ **Receive Request**, $((client.travel.search)?clientReq)^{+} >$

. $<$ **name** $\mapsto$ **Assign Requests**, $flightReq.itinerary := clientReq.itinerary, carReq.dates := clientReq.dates,$
$carReq.city := clientReq.city, hotelReq.dates := clientReq.dates, hotelReq.city := clientReq.city$
$toursReq.dates := clientReq.dates, toursReq.city := clientReq.city >$

. $<$ **name** $\mapsto$ **Invoke Flight**, $(flight.book.search)!(flightReq, flightResp) >$

. $<$ **name** $\mapsto$ **Invoke Hotel**, $(hotel.book.search)!(hotelReq, hotelResp) >$

. $<$ **name** $\mapsto$ **Invoke Car**, $(car.rent.search)!(carReq, carResp) >$

|| $<$ **name** $\mapsto$ **Invoke Tours**, $(tours.services.getInfo)!(toursReq, toursResp >$

. $<$ **name** $\mapsto$ **Assign Responses**, $clientResp.flightPrice := flightResp, clientResp.hotelPrice := hotelResp,$
$clientResp.carPrice := carResp, clientResp.toursPrice := toursReps >$

. $<$ **name** $\mapsto$ **Replay Request**, $client.travel.results \hookleftarrow clientResp >$

$>$

It is clear, from the previous example, that AV-BPEL is less cumbersome and more suitable to give the semantic rules.

4.2 Bank transfer transaction

We choose a realistic bank transfer transaction scenario, to show how semantics deals with error handling and compensation. We consider the example of Fig. 2.

The basic idea is that the process P realizes a bank transfer transaction between two accounts. Assume that the aim of $S1$ is to credit the account c_1 of the receiver with 5 dollars, where $S2$ aims to debit the account of the sender by the same amount. Assume also that an error occurs during the execution of $S2$ and the transaction needs to be cancelled. The semantics of AV-BPEL will show the sequence of activities triggered by the error. Basically, the fault handler of the process P terminates all the activities in $S2$ and then compensates $S1$ by executing its compensation handler $C1$.

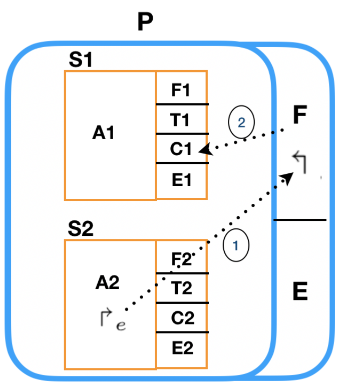

Fig. 2: Error handling and compensation

Formally, let $P =< \mathcal{E}, \mathcal{H}, S1.S2 >$, where $S1 =< \mathcal{E}_1, \mathcal{H}_1, c_1 := c_1 + 5 >$, $S2 =< \mathcal{E}_2, \mathcal{H}_2, \Rsh_e>$, where e is the name of an error . Suppose also that:

$$\begin{array}{ll}\mathcal{E}(c_1) & = 2\\ \mathcal{H} & = < F, E >\\ F & = < \mathcal{E}_F, \mathcal{H}_F, \Lsh\$ >\\ \mathcal{H}_1 & = < F_1, T_1, C_1, E_1 >\\ C_1 & = < \mathcal{E}_{C_1}, \mathcal{H}_{C_1}, c_1 := c_1 - 5 >\end{array}$$

The parameters that are not completely defined will not be relevant for this example. The compensation handler of the scope $S1$ debits the account c_1 if the rest of transaction fails. We assume that c_1 is defined in $\mathcal{E}$ and not redefined in $\mathcal{E}_1$.

In the following, we explain the evolution of the process P step by step.

1. P starts by executing the $S1$:
$$P \xrightarrow{c_1:=7} P_1$$
where $P_1 =< \mathcal{E}[c_1 \mapsto 7], \mathcal{H}, < \mathcal{E}_1, \mathcal{H}_1, 1 > .S2 >$. The proof is in Table 12.
2. Since $S1$ completes successfully, then its compensation handler is stored for possible future use.
$$P_1 \xrightarrow{\tau} P_2$$
where $P_2 =< \mathcal{E}[c_1 \mapsto 7], < F/ < \mathcal{E}_1, C_1 >, E >, S2 >$. The proof is in Table 12.
3. $S2$ can now start its execution. But it will generate an error that will be caught by the event handler of the process (event (1) in Fig. 2) and the execution of residual part of the process switch to the fault mode.
$$P_2 \xrightarrow{\Rsh_e} P_3$$
where $P_3 =< \mathcal{E}[c_1 \mapsto 7][\text{err} \mapsto e], << \mathcal{E}_F, \mathcal{H}_F, \$ > / < \mathcal{E}_1, C_1 >, 1 >, 1 >^{\mathbb{F}}$. The proof is in Table 12.

4. The fault handler of the process starts the compensation mode.

$$P_3 \xrightarrow{\$} P_4$$

where $P_4 =< \mathcal{E}' \mapsto e], << \mathcal{E}_F, \mathcal{H}_F, 1 > / < \mathcal{E}_1, C_1 >, 1 >, 1 >^{\mathbb{C}}$ and $\mathcal{E}' = \mathcal{E}[c_1 \mapsto 7][\mathsf{err} \mapsto e]$. The proof is in Table 12.

5. The compensation handler starts:

$P_4 \xrightarrow{c_1:=2} P_5$ where $P_5 =< \mathcal{E}'[c_1 \mapsto 2], << \mathcal{E}_F, \mathcal{H}_F, 1 > / < \mathcal{E}_1, < \mathcal{E}_{C_1}, \mathcal{H}_{C_1}, 1 >>, 1 >, 1 >^{\mathbb{C}}$. The proof is in Table 12.

6. Since compensation handler has finished, we go back to the fault handler:

$$P_5 \xrightarrow{\tau} P_6$$

where $P_6 =< \mathcal{E}'[c_1 \mapsto 2], << \mathcal{E}_F, \mathcal{H}_F, 1 >, 1 >, 1 >^{\mathbb{F}}$. The proof is in Table 12.

7. Finally, since the fault handler has finished as well as the compensation handler, and since the process does not have other activities, then it terminates.

$$P_6 \xrightarrow{\tau} P_7$$

where $P_7 = 1$. The proof is in Table 12.

Table 12: Proofs related to the Example

Proof of $P \xrightarrow{c_1:=7} P_1$
$(R^1_s)\,\dfrac{(R_.)\,\dfrac{(R_{\mathcal{E}})\,\dfrac{(R_{:=})\,\dfrac{[\![c_1+5]\!]_{\mathcal{E}:\mathcal{E}_1}=7}{<\mathcal{E}:\mathcal{E}_1,\mathcal{H}_1,c_1:=c_1+5>\xrightarrow{c_1:=7}<\mathcal{E}[c_1\mapsto 7]:\mathcal{E}_1,\mathcal{H}_1,1>}}{<\mathcal{E},<\mathcal{E}_1,\mathcal{H}_1,c_1:=c_1+5>>\xrightarrow{c_1:=7}<\mathcal{E}[c_1\mapsto 7],<\mathcal{E}_1,\mathcal{H}_1,1>>}}{<\mathcal{E},<\mathcal{E}_1,\mathcal{H}_1,c_1:=c_1+5>.S2>\xrightarrow{c_1:=7}<\mathcal{E}[c_1\mapsto 7],<\mathcal{E}_1,\mathcal{H}_1,1>.S2>}}{<\mathcal{E},\mathcal{H},<\mathcal{E}_1,\mathcal{H}_1,c_1:=c_1+5>.S2>\xrightarrow{c_1:=7}<\mathcal{E}[c_1\mapsto 7],\mathcal{H},<\mathcal{E}_1,\mathcal{H}_1,1>.S2>}$ Recall that $\mathcal{E}:\mathcal{E}_1(c_1)=\mathcal{E}(c_1)=2$ since c_1 is not defined in $\mathcal{E}_1$.
Proof of $P_1 \xrightarrow{\tau} P_2$
$(R^3_S)\,\dfrac{\Box}{<\mathcal{E}[c_1\mapsto 7],\mathcal{H},<\mathcal{E}_1,\mathcal{H}_1,1>.S2>\xrightarrow{\tau}<\mathcal{E}[c_1\mapsto 7],<F/<\mathcal{E}_1,C_1>,E>,S2>}$
Proof of $P_2 \xrightarrow{\tau} P_3$
$(R^{s'}_F)\,\dfrac{(R_{\mathcal{E}})\,\dfrac{(R_{\Rsh_e})\,\dfrac{\Box}{<\mathcal{E}[c_1\mapsto 7]:\mathcal{E}_2,\mathcal{H}_2,\Rsh_e>\xrightarrow{\Rsh_e}<\mathcal{E}[c_1\mapsto 7]:\mathcal{E}_2,\mathcal{H}_2,1>}}{<\mathcal{E}[c_1\mapsto 7]<\mathcal{E}_2,\mathcal{H}_2,\Rsh_e>>\xrightarrow{\Rsh_e}<\mathcal{E}[c_1\mapsto 7]<\mathcal{E}_2,\mathcal{H}_2,1>>}\quad <\mathcal{E},F>\not\xrightarrow{\Lsh_e}\quad (R_{\mathcal{E}})\,\dfrac{(R^1_s)\,\dfrac{(R_{\Lsh})\,\dfrac{\Box}{<\mathcal{E}[c_1\mapsto 7]:\mathcal{E}_F,\Lsh\$>\xrightarrow{\Lsh}<\mathcal{E}[c_1\mapsto 7]:\mathcal{E}_F,\$>}}{<\mathcal{E}[c_1\mapsto 7]:\mathcal{E}_F,\mathcal{H}_F,\Lsh\$>\xrightarrow{\Lsh}<\mathcal{E}[c_1\mapsto 7]:\mathcal{E}_F,\mathcal{H}_F,\$>}}{<\mathcal{E}[c_1\mapsto 7],F>\xrightarrow{\Lsh}<\mathcal{E}[c_1\mapsto 7],<\mathcal{E}_F,\mathcal{H}_F,\$>}}{<\mathcal{E}[c_1\mapsto 7],,<F/<\mathcal{E}_1C_1>,E>,<\mathcal{E}_2,\mathcal{H}_2,\Rsh_e>>\xrightarrow{\Rsh_e}<\mathcal{E}[c_1\mapsto 7][\mathsf{err}\mapsto e],<<\mathcal{E}_F,\mathcal{H}_F,\$>/<\mathcal{E}_1,C_1>,1>,1>^{\mathbb{F}}}$
Proof of $P_3 \xrightarrow{\tau} P_4$
$(R^s_C)\,\dfrac{(R_{\mathcal{E}})\,\dfrac{(R_{\$})\,\dfrac{\Box}{<\mathcal{E}':\mathcal{E}_F,\mathcal{H}_F,\$>\xrightarrow{\$}<\mathcal{E}':\mathcal{E}_F,\mathcal{H}_F,1>}}{<\mathcal{E}',<\mathcal{E}_F,\mathcal{H}_F,\$>>\xrightarrow{\$}<\mathcal{E}',<\mathcal{E}_F,\mathcal{H}_F,1>>}}{<\mathcal{E}',<<\mathcal{E}_F,\mathcal{H}_F,\$>/<\mathcal{E}_1,C_1>,1>,1>^{\mathbb{F}}\xrightarrow{\$}<\mathcal{E}',<<\mathcal{E}_F,\mathcal{H}_F,1>/<\mathcal{E}_1,C_1>,1>,1>^{\mathbb{C}}}$
Proof of $P_4 \xrightarrow{\tau} P_5$
$(R_C)\,\dfrac{(R_{\mathcal{E}})\,\dfrac{(R_{\mathcal{E}})\,\dfrac{(R^1_S)\,\dfrac{(R_{:=})\,\dfrac{[\![c_1-5]\!]_{\mathcal{E}':\mathcal{E}_1:\mathcal{E}_{C_1}}=2}{<\mathcal{E}':\mathcal{E}_1:\mathcal{E}_{C_1},c_1:=c_1-5>^{\mathbb{C}}\xrightarrow{c_1:=2}<\mathcal{E}'[c_1\mapsto 2]:\mathcal{E}_1:\mathcal{E}_{C_1},1>^{\mathbb{C}}}}{<\mathcal{E}':\mathcal{E}_1:\mathcal{E}_{C_1},\mathcal{H}_{C_1},c_1:=c_1-5>^{\mathbb{C}}\xrightarrow{c_1:=2}<\mathcal{E}'[c_1\mapsto 2]:\mathcal{E}_1:\mathcal{E}_{C_1},\mathcal{H}_{C_1},1>^{\mathbb{C}}}}{<\mathcal{E}':\mathcal{E}_1,<\mathcal{E}_{C_1},\mathcal{H}_{C_1},c_1:=c_1-5>>^{\mathbb{C}}\xrightarrow{c_1:=2}<\mathcal{E}'[c_1\mapsto 2]:\mathcal{E}_1,<\mathcal{E}_{C_1},\mathcal{H}_{C_1},1>>^{\mathbb{C}}}}{<\mathcal{E}',<\mathcal{E}_1,<\mathcal{E}_{C_1},\mathcal{H}_{C_1},c_1:=c_1-5>>>^{\mathbb{C}}\xrightarrow{c_1:=2}<\mathcal{E}'[c_1\mapsto 2],<\mathcal{E}_1,<\mathcal{E}_{C_1},\mathcal{H}_{C_1},1>>>^{\mathbb{C}}}}{<\mathcal{E}',<<\mathcal{E}_F,\mathcal{H}_F,1>/<\mathcal{E}_1,<\mathcal{E}_{C_1},\mathcal{H}_{C_1},c_1:=c_1-5>>,1>,1>^{\mathbb{C}}\xrightarrow{c_1:=2}<\mathcal{E}'[c_1\mapsto 2],<<\mathcal{E}_F,\mathcal{H}_F,1>/<\mathcal{E}_1,<\mathcal{E}_{C_1},\mathcal{H}_{C_1},1>>,1>,1>^{\mathbb{C}}}$
Proof of $P_5 \xrightarrow{\tau} P_6$
$(R^t_C)\,\dfrac{\Box}{<\mathcal{E}'[c_1\mapsto 2],<<\mathcal{E}_F,\mathcal{H}_F,1>/<\mathcal{E}_1,<\mathcal{E}_{C_1},\mathcal{H}_{C_1},1>>,1>,1>^{\mathbb{C}}\xrightarrow{\tau}<\mathcal{E}'[c_1\mapsto 2],<<\mathcal{E}_F,\mathcal{H}_F,1>,1>,1>^{\mathbb{F}}}$
Proof of $P_6 \xrightarrow{\tau} P_7$
$(R_Ft)\,\dfrac{\Box}{<\mathcal{E}'[c_1\mapsto 2],<<\mathcal{E}_F,\mathcal{H}_F,1>,1>,1>^{\mathbb{F}}\xrightarrow{\tau}1}$

5 Related Work

A rich variety of excellent theoretical works have been proposed to formalize different subsets of BPEL and the research related to this topic is still ongoing looking for the perfect (precise, complete, simple, supported by tool, etc.) formalization. Most of them have formalized the semantics of BPEL by defining a mapping from the syntax of BPEL to an existing formal language. In particular, the use of Petri net and existing process calculi (LOTOS, CCS, TSTP, π-calculus, Promela etc.) to formalize BPEL have been widely investigated. In fact, these formal language give the advantage of benefiting from their accompanying tools (CWB-NC, CADP, MWB, etc.) that are very useful for designing and verifying web service compositions. They usually provide the possibility of specifying web service liveness (something good happens) and safety (bad events do not happens) properties using temporal logic (LTL, CTL, μ-calculus) and automatically verifying them. Many interesting properties related to bad behaviors such as deadlock or mutual exclusion can be early detected before even deploying the service.

Petri nets provide a very expressive language that is frequently used to model parallel systems and could be suitable to formalize the composition of Web Services. For that reason, many attempts of using Petri nets to formalize different subsets of BPEL have been introduced in [20], [21], [22], [23], [24], [25], [26], [27], [28], [29] and [30] . A formalization of a more complete subset of BPEL, including exception handling and compensations, using Petri net could be find in [25]. A more recent Petri net based approach to formally model and verify BPEL services can be found in [31]. The authors present an approach for transforming BPEL specifications to Workflow WF-nets models, then verifying behavioral properties on IOWF (Inter-Organizational Workflow) processes.

Process algebra provides also an elegant and powerful formalism for specifying and verifying composition of web services, either by defining a new one or by one or two-way mapping of BPEL syntax into an existing calculus. For example, Lucchi and Mazzara have proposed in [17] a mapping from a BPEL process to a π-based calculus, named web$\pi\infty$. The authors advocated that the different mechanisms (the fault handler, the termination handler and the compensation handler) for error handling are not needed and they proposed the idea of event notification to substitute them. Yang and Zhong presented in [32] a formalization of a subset of BPEL 2.0 based also on the π-calculus. Another approach proposed in [33] maps SCA/BPEL to the Wright/CSP ADL in order to verify the architectural properties of an SCA software architecture equipped with its BPEL behavioral aspects. In [34] authors presented a formal verification of the behavioral properties of SCDL/WS-BPEL service-component architectures.

In [35, 36], a two-way mapping between a fragment of BPEL and LOTOS was presented by Ferrara et *al.* They have considered most of BPEL's activities including fault handlers, compensation handlers and event handlers. Another two-way mapping approach between the π-based orchestration calculus and a subset of BPEL was proposed in [37]. In [38], Zhu et *al.* also propose a two-way transformation from a subset of BPEL to CSP. After that the generated CSP model of the BPEL process together with some CSP assertions can be sent to FDR (Failures-Divergences Refinement), a refinement checker, to verify some properties. If the verification fails a counter-example is given, then the CSP model could be modified to meet the requirement and the new version could be transferred back to get a reliable BPEL process. In [39, 40], Pu et *al.* introduced BPEL0, a lightened version of BPEL, where the complicated XML style is abandoned and by focusing on fault and compensation handling. The formal semantics of the language has been given as well as a new bisimulation, called n-bisimulation, to compare BPEL0 process. In [41], Tu et *al.* proposed a Priced Probabilistic Process Algebra, a probabilistic process algebra extended with cost, to take into consideration this important factor, especially when different scenarios of service composition fit with requirements. The authors demonstrate that WS composition can be controlled by Markov decision process with target function of optimal cost. In [42], Rai et *al.* use recursive composition technique to specify and verify interactions between web services. They propose a recursive composition interaction graph (RCIG) to specify service interaction and a recursive composition specification language (RCSL) to capture the properties that we want to verify. Another approach based RECATNet to model and verify a fragment of BPEL processes has been also proposed in [43]. In [44] , Pugliesea and Tiezzi proposed a process calculus called COWS to specify web services compositions and model their life cycle (publication, discovery, negotiation, orchestration, deployment, reconfiguration and execution). In [45], Spieler provided a formal semantics for the FCT-handling mechanisms to capture the FCT-part of the WS-BPEL 2.0 specification in full detail.

Classical State Transitions Systems (STS) have also been proposed to model and specify BPEL processes. Marconi et *al.* presented, in [46, 47], an approach to generate new web services from other ones and a requirement. Given a set services $S = \{S_1, \ldots, S_n\}$ in BPEL and a requirement Φ, the idea is to check if we can generate, by composing a subset of S, another service S_Φ that respects the requirement Φ. First the service in S are transformed to STS, then synthesis technique are used to produce the STS of S_Φ which will be translated later to BPEL. The approach is very interesting but it consider only a subset of BPEL that does not takes handlers into consideration. Nakajima proposed in [48, 49] a mapping from a subset of BPEL activities to a Extended Finite-state Automaton (EFA). After that the EFA is translated to a Promela process which is the specification language of the model checker SPIN to analyze behavioral aspects and to detect potential deadlocks that may occur in the Promela description or any property specified using the LTL logic.

Using Event-B as a formal method supported by tools has been also explored to specify and verify web services composition. For instance, in [50], Ait-Sadoune et *al.* describe a stepwise refinement approach for a structure-preserving modeling of BPEL using Event-B method. The proposed approach formalizes the static and the dynamic parts of BPEL and is supported by a translation tool and considers both generic and custom correctness properties for verification. A novel correct-by-construction approach based on refinement using the Event-B method has been presented in [51].

Translating rules from BPEL language to a low-level real-time model (DATA) have been proposed in [52]. They present an operational semantics of BPEL based on a real-time model that suupports true-concurrency semantics, timing constraints and actions durations.

Several interesting formal contributions have been proposed in the literature to specify the WS composition in a formal and expressive language. Most of them have formalized the semantics of BPEL by defining a mapping from BPEL to an existing formal language.

Table 13 provides a more structured overview of related work. We use the next symbols to indicate whether the formalization covers the constructs of BPEL fully (✓), partially (+/-) or not at all ( ).

Table 13: A comparative summary of related work on formalization of BPEL

	Approach			**Type of mapping**		**Translation vs dedicated sematics**		**Supported by tools**		**Features covered in BPEL**			
Paper	**Petri Net**	**Automaton**	**Process Algebra**	**One-way**	**Two-ways**	**Translation**	**Dedicated**	**Yes**	**No**	**FCT-handlers**	**Correlation**	**Links**	**Attributes**
[21]	✓			✓		✓			✓	✓		✓	✓
[17]			✓	✓		✓			✓	✓		✓	+/-
[47]		✓		✓			✓	Astro		✓		✓	+/-
[53]	✓			✓			✓	WofBPEL		+/-		✓	+/-
[49]		✓		✓		✓		SPIN				✓	+/-
[35]			✓		✓	✓			✓	✓			+/-
[54]	✓			✓		✓		Woflan				✓	+/-
[37]			✓		✓	✓		HAL		✓			✓
[32]	✓			✓		✓			✓				✓
[33]			✓	✓		✓		Wr2fdr					+/-
[52]	✓			✓		✓		UPPAAL		✓			✓
[38]			✓	✓		✓		FDR					✓
[40]			✓	✓			✓		✓	✓			✓
[41]			✓	✓		✓			✓				+/-
[43]	✓			✓		✓		Maude					+/-
[51]		✓		✓		✓		Rodin		+/-			✓
[50]		✓		✓		✓		Rodin					✓
[55]		✓		✓		✓			✓	✓		✓	✓
Our work			✓		✓		✓		✓	✓	✓	✓	✓

As shown by Table 13, several features can be used to compare existing works such that :

- Translation vs. dedicated semantics : A common way to formalize a language is to map its syntax to the one of another formal existing language with well understood semantics like pi-calculus or Petri-Net. The advantage of this approach is that the original language can benefit from the techniques and tools developed for the language in which it is translated. However, a common critique of this approach is, when the two languages do not have the same level of abstraction, the map could be challenging and leads to a complicated structure with non-intuitive meaning. Also, when the translated version of the program is executed and errors occurs, it will be difficult to track it back to the original language.
  The second option is to develop a dedicated semantics to the language from the scratch. This approach usually gives a clearer and more intuitive semantics. In facts, the semantic rules are designed from the beginning to fit exactly with the intuitive meaning of the constructs of the language. Also, within this approach the debugging a program is usually simpler than the case of translation semantics.
- One-way vs Two-way Translation semantics : A tow-way mapping approach has the advantage of tackling problem such as service optimization and refinement that cannot be easily handle by a one-way translation approach. In fact, once a service is translated to another language in which some optimizations or refinements are accomplished, it will be difficult to inherit these improvements if we cannot translate the modified version back to the original language.
- Completeness of the formalization : Most of the proposed formalization focus on the common activities that BPEL shares with other programming languages such that sequence, conditional, looping and parallel. Few of these for-

malization take into consideration handlers and very rare are the works that take into consideration attributes, links, correlation, type of data. In our work, we targeted BPEL in almost all its details by taking into consideration links correlation, most relevant attributes as well as types of data and query languages.

– Accuracy of the formalization : Sometimes the proposed formalization of some concepts does not fit exactly with the reality of BPEL. For instance, according to the semantics of BPEL, compensations can be executed in parallel when errors occur in two parallel scopes. This is the case for our semantics but it is not the case for other ones such as [56, 45]. Also, in BPEL a receive action is blocking while send action is not. Most of the proposed calculi use synchronous communication. But the semantics of our formalization uses an asynchronous communication based on floating messages. Another example is related to assignment. This explanation of this activity takes 15 pages in the specification of BPEL[13] showing that it is a complex structure that can contain many sub assignments that should be executed in atomic way. In our formalization we take into consideration this detail but they are neglected by others.

However, the main drawback of our formalization is that it is not yet supported by any tools (such as a model checker) to take benefit from it that goes beyond a better understanding of the meaning of BPEL constructs.

6 Conclusion and future work

This paper provides a formal operational semantics to AV-BPEL (an abbreviated version of WS-BPEL) that takes into consideration almost all details of BPEL, in order to disambiguate some important concepts including (EFCT)-handlers. We give a bank transfer transaction scenario to show that our approach is relevant in reality, this example illustrates how the operational rules are put in action to appropriately run (EFCT)-handlers to cancel a transaction when it partially fails. Another detailed example of Travel Agency is given to better clarify the relation between AV-BPEL and BPEL. We also provided some recommendations to improve BPEL and a comparative study of related works.

As a future work, we want to connect our formalization with some existing tools (model checkers, deductive program verifiers, test case generators, monitors, etc.) so that we can take more benefit from it to reach a variety of goals. An easy way to make this kind of connection is to translate our formalization to K-Framework [57] which provides these tools without additional efforts.

Another future direction that we are exploring now is the automatic formal enforcement of security policies on web service compositions. The proposed approach is also an important step towards many formal analyses, as to ensure that the composite web service can be executed correctly. Given a security policy Φ, a web service composition S, we want use rewriting techniques to automatically generate a new version S' that behaves like S whenever Φ is respected and stops or produces a new behavior when Φ is violated.

References

[1] Q. Yu, X. Liu, A. Bouguettaya, and B. Medjahed, "Deploying and managing web services: issues, solutions, and directions," *VLDB J.*, vol. 17, pp. 537–572, 2008.

[2] M. P. Papazoglou and W. J. Heuvel, "Service oriented architectures: approaches, technologies and research issues," in *VLDB J*, pp. 389–415, 2007.

[3] A. Bouguettaya, Q. Z. Sheng, and F. Daniel, "Web services foundations," Springer, 2013.

[4] N. Bradley, "The xml companion," 2002. Addison Wesley.

[5] N. Mitra and Y. Lafon, "Soap version 1.2 part 0: Primer (second edition)." `http://www.w3.org/TR/2007/REC-soap12-part0-20070427/`, April 2007. Online; accessed 9-November-2019.

[6] E. Christenses, F. Curbera, G. Meredith, and S. Weerawarana, "Web services description language (wsdl 1.1)." `http://www.w3.org/TR/wsdl`, March 2003. Online; accessed 9-November-2019.

[7] OASIS, "Uddi version 2.04 api specification." `http://www.uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm`, July 2002. Online; accessed 9-November-2019.

[8] S. Mishra, D. Puthal, B. Sahoo, S. Jena, and M. Obaidat, "An adaptive task allocation technique for green cloud computin," *The Journal of Supercomput*, vol. 74, pp. 370–385, january 2018.

[9] A. Vakili and N. Navimipou, "Comprehensive and systematic review of the service composition mechanisms in the cloud environments.," *Journal of Network and Computer Applications*, vol. 81, pp. 24–36, March 2017.

[10] A. AlSedrani and A. Touir, "Web service composition processes: A comparative study," *International Journal on Web Service Computing (IJWSC)*, vol. 7, March 2016.

[11] L.Chen, S.Thombre, K. Jarvinen, and al., "Robustness, security and privacy in locationâĂŘbased services for future iot: a survey.," in *IEEE Access*, pp. 8956–8977, April 2017.

[12] C.Jatoth, G. Gangadharan, and R. Buyya, "Computational intelligence based qosâĂŘaware web service composition: a systematic literature review.," in *IEEE Transactions on Services Computing*, pp. 475–492, 2017.

[13] OASIS, “Web services business process execution language version 2.0.” http://docs.oasis-open.org/wsbpel/2.0/, April 2007.
[14] N. Kavantzas, “Web services choreography description language (ws-cdl) version 1.0.” http://www.w3.org/TR/ws-cdl-10/, 2004. Online; accessed 9-November-2019.
[15] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P. Takacsi-Nagy, and Stand., “Web service choreography interface (wsci),” 2002. Propos. by BEA Syst. Intalio, S., Sun Microsystems.
[16] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana, “The next step in web services.,” *ACM Tansaction on Information and System Security*, vol. 46, pp. 29–34, October 2003.
[17] R. Lucchi and M. Mazzara, “A pi-calculus based semantics for ws-bpel,” *Journal of Logic and Algebraic Programming*, vol. 70, pp. 96–118, 2007.
[18] D. Mills, P. Koletzke, and A. Roy-Faderman, *Oracle JDeveloper 11G Handbook*. New York, NY, USA: McGraw-Hill, Inc., 1 ed., 2010.
[19] O. Community, “Openesb.” https://www.open-esb.net, 2019. Online; accessed 9-November-2019.
[20] S. Narayanan and S. McIlraith, “Simulation, verification and automated composition of web service,” in *in Proc. Int. 11th Int. Conf. World Wide Web*, pp. 77–88, 2002.
[21] N. Lohmann, “A feature-complete petri net semantics for ws- bpel 2.0,” in *in Proceedings of the Workshop on Formal Approaches to Business Processes and Web Services (FABPWS’07), University of Podlasie*, pp. 21–35, 2007.
[22] J. Zhang, J. Y. Chung, C. K. Chang, and S. W. Kim, “Ws-net: A petri-net based specification model for web services,” in *in Proc. IEEE Int. Conf. Web Services*, pp. 420–427, 2008.
[23] W. V. Aalst, A. Mooij, C. Stahl, and K. Wolf, “Service interaction: Patterns, formalization, and analysis,” in *in 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, (SFM’09), vol. 5569, Bertinoro, Italy*, pp. 42–88, 2009.
[24] Y. S. W. Tan and M. Zhou, “A petri net-based method for compatibility analysis and composition of web services in business process execution language,” *IEEE TRANSACTIONS ON AUTOMATION SCIENCE AND ENGINEERING*, vol. 6, 2009.
[25] C. Stahl, “A Petri Net Semantics for BPEL,” Informatik-Berichte 188, Humboldt-Universität zu Berlin, July 2005.
[26] W. Tan, Y. S. Fan, M. C. Zhou, and Z. Tian, “Data-driven service composition in enterprise soa solutions: A petri net approach,” in *IEEE TRANSACTIONS ON AUTOMATION SCIENCE AND ENGINEERING*, vol. 7, pp. 686–694, 2010.
[27] N. Q. Wu and M. C. Zhou, “System modeling and control with resource-oriented petri nets,” CRC Press, 2010.
[28] N. Q. Wu, M. C. Zhou, and G. Hu, “Petri net modeling and one-step look-ahead maximally permissive deadlock control of automated manufacturing systems,” in *ACM Trans. Embedded Comput. Syst*, vol. 12, pp. 10:1–10:23, 2013.
[29] A. Bourouis, K. Klai, N. Ben-Hadj-Alouane, and Y. El-Touati, “On the verification of opacity in web services and their composition,” in *IEEE Transactions on Services Computing*, vol. 4421, pp. 66–79, IEEE, 2017.
[30] N. Parimala, “Modelling co-occurring changes in a BPEL process with petri nets,” in *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2019, Heraklion, Crete, Greece, May 4-5, 2019* (E. Damiani, G. Spanoudakis, and L. A. Maciaszek, eds.), pp. 410–416, SciTePress, 2019.
[31] S. Boukhedouma and Z. Alimazighi, “An approach based on hierarchical petri nets for the verification of interconnected bpel processes,” *Journal of software: practice and experienceInternational Journal of Information System Modeling and Design*, vol. 9, pp. 44–78, 2018.
[32] C. Yang and F. Zhong, “Towards the formal foundation of orchestration process,” in *5th ICCCNT- 2014*, 2014.
[33] T. Rouis, M. Bhiri, L. Sliman, and M.Kmimech, “Towards a formal approach for the verification of sca/bpel software architectures,” in *in 2017 8th International Conference on Information, Intelligence, Systems & Applications (IISA), Larnaca, Cyprus*, 2017.
[34] T. Rouis, M. Bhiri, and M.Kmimech, “Mapping scdl/bpel to ada for formal verification of the behavioral properties of service-component architecturemapping,” *International Journal on Web Service Computing (IJWSC)*, vol. 9, March 2018.
[35] A. Ferrara, “Web services: a process algebra approach,” in *Proceedings of 2nd ACM International Conference on Service Oriented Computing*, pp. 242–255, November 2004.
[36] G. Salaün, A. Ferrara, and A. Chirichiello, “Negotiation among web services using lotos/cadp,” in *Proceedings of the European Conference on Web Services*, vol. 3250 of Lecture Notes in Computer Science, pp. 198–212, 2004.
[37] F. Abouzaid and J. Mullins, “A calculus for generation, verification and refinement of bpel specifications,” in *Electronic Notes in Theoretical Computer Science*, vol. 4421, pp. 43–65, 2008.
[38] Y. Zhu, Z. Huang, and H. Zhou, “Modeling and verification of web services composition based on model transformation,” *Journal of software: practice and experience International Journal of Information System Modeling and Design*, vol. 47, pp. 709–730, 2017.
[39] R. Mateescu, P. Poizat, and G. Salaün, “Adaptation of service protocols using process algebra and on-the-fly reduction techniques,” in *IEEE Transactions on Software Engineering, Institute of Electrical and Electronics Engineers (IEEE)*, 2012.
[40] G. Pu, H. Zhu, Z. Qiu, S. Wang, X. Zhao, and J. He, “Theoretical foundation of scope-based compensable fow language for web service,” in *Proceedings of the 8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, vol. 4037 of Lecture Notes in Computer Science, pp. 251–266, June 2006.
[41] L. Tu, F. Xiao, and Z. Huang, “Modeling service composition using priced probabilistic process algebra,” in *in Proc. IEEE 5th Int. Symp. Service Oriented Syst. Eng*, pp. 35–38, 2010.
[42] G. N. Rai, G. R. Gangadharan, V. Padmanabhan, and R. Buyya, “Web service interaction modeling and verification using recursive composition algebra,” in *in IEEE Transactions on Services Computing*, IEEE, 2018.
[43] A. Kheldoun and M. Ioualalen, “Transformation bpel processes to recatnet for analysing web services compositions,” in *Model-Driven Engineering and Software Development (MODELSWARD), 2014 2nd International Conference on*, pp. 425–430, 2014.

[44] R. Pugliesea and F. Tiezzi, "A calculus for orchestration of web services," in *Journal of Applied Logic*, vol. 10-1, pp. 2–31, 2012.
[45] D. Spieler, "Scope-based fct-handling in ws-bpel 2.0," Master's thesis, Saarland University, 2008.
[46] A. Marconi, "Automated process-level composition of web services: from requirements specification to process run," Master's thesis, University of Trento, Italy, 2008.
[47] A. Marconi and M. Pistore, "Synthesis and composition of web services," *Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM*, vol. 5569, pp. 89–157, 2009.
[48] S. Nakajima, "Lightweight formal analysis of web service flows," in *Progress in Informatics*, pp. 57–76, 2005.
[49] S. Nakajima, "Model-checking behavioral specification of bpel applications," in *Electronic Notes in Theoretical Computer Science*, vol. 151, pp. 89–105, 2006.
[50] I. Ait-Sadoune and Y. Ait-Ameur, "Stepwise development of formal models for web services compositions: Modelling and property verification," *Transactions on Large- Scale Data- and Knowledge-Centered Systems*, pp. 1–33, 2013.
[51] Y. A. A. G. Babin and M. Pantel, "Formal verification of runtime compensation of web service compositions: A refinement and proof based proposal with event-b," in *2015 IEEE International conference on services computing*, vol. 978, pp. 7–15, 2015.
[52] N. B. I. Chama and D. Saadouni, "Formalization and analysis of timed bpel," in *IEEE IRI 2014*, vol. 978, pp. 4799–5880, 2014.
[53] C. Ouyang, E. Verbeek, W. V. der Aalst, S. Breutel, and M. Dumas, "Formal semantics and analysis of control flow in ws-bpel," *Sci. Comput. Program*, vol. 67, pp. 162–198, 2007.
[54] H. Verbeek and W. V. der Aalst, "Analyzing bpel processes using petri nets," in *2nd International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*, 2005.
[55] E. Stachtiari and P. Katsaros, "Compositional execution semantics for business process verification," *The Journal of Systems and Software*, vol. 137, pp. 217–238, 2018.
[56] A. Lapadula, R. Pugliese, and F. Tiezzi, "A calculus for orchestration of web services," in *In Proc. 16th European Symposium on Programming*, pp. 33–47, Springer, 2007.
[57] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.