



The United Nations
University

UNU-IIST

International Institute for
Software Technology

A Survey of Business Process Execution Language (BPEL)

Chris Ma, Qiwen Xu and J. W. Sanders

September 2009

UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macao, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the government of Macao and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macao authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on **formal methods** for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macao or visit UNU-IIST's home page: <http://www.iist.unu.edu>, if you would like to know more about UNU-IIST and its report series.

Govindan Parayil, Director a.i.



The United Nations
University

UNU-IIST

**International Institute for
Software Technology**

P.O. Box 3058
Macao

A Survey of Business Process Execution Language (BPEL)

Chris Ma, Qiwen Xu and J. W. Sanders

Abstract

This survey introduces a link between BPEL and a more abstract BPEL-like language. It is tutorial in nature, focusing on features particular to BPEL, like scope-based compensation and fault-handling. Such features are demonstrated here with a number of executable BPEL programs. Although BPEL also provides features common to other programming languages, it contains three important components: the BPEL program (program algebra), XSD (Data types) and WSDL (Input/Output). This survey concentrates on clarifying program logic and algebra instead of syntax. In doing so an undesirable feature of compensation in BPEL is observed and a correction for it proposed. It is argued that such observations justify the more abstract view of BPEL taken here which facilitates accountable correctness.

Chris Ma Chris Ma is a Principal Technical Supervisor of Computer Service Centre, Macao Polytechnic Institute. His interests in applying Formal Methods to industry software.

Xu Qiwen is an Assistant Professor at Faculty of Sciences and Technology, University of Macau. His interests lie largely in Formal Methods.

Jeff Sanders is Principal Research Fellow at UNU-IIST. His interests lie largely in Formal Methods.

The authors acknowledge assistance from the Macao Science and Technology Development Fund under the PEARL project, grant number 041/2007/A3.

Contents

1. Introduction	7
2. An abstract BPEL-like language	7
2.1 Lifecycle of BPEL instance	7
2.2 Testing environment of this survey	7
2.3 Abstract language	8
3. BPEL activities	15
3.1 Basic activity	15
3.2 Structured activity	17
4. Case study – Hong Kong SunFlower Travel	20
5. Discussion	23
6. Conclusion	24
References	24
Appendix: Table of acronyms	25

1. Introduction

BPEL [4,5] is an XML-based programming language, standardised by the OASIS WS-BPEL technical committee with latest version 2.0 on Sep 2004. Its purpose is to describe high-level inter and intra business communication. An example is a travel agency which reserves rooms from a hotel and air-tickets from an airline: the hotel and airline independently provide web services for reservation. But the travel agency can orchestrate them to provide a new web service offering a minimum of incompatibility only if it can rely on compensation when necessary.

However programming in BPEL is as difficult as programming any distributed system, and hence considerably more difficult than sequential programming. Accountably accurate BPEL programs thus require tools from Software Engineering. In this paper a more abstract but still procedural language is considered that encapsulates BPEL's characteristic features like scope-based compensation and fault handling. With it, a start is made to a study of the laws of BPEL programs. This work is thus seen as complementing other more abstract approaches to orchestration, like Orc [7] which is a functional language abstracting those features. Formal treatments in a different direction but more extensive than that begun here are given in [1], [2] and [3].

A table of acronyms ---like BPEL itself--- is given in the Appendix.

2. An abstract BPEL-like language

2.1 Lifecycle of BPEL instance

In most cases, a BPEL program serves as a server-side service that is invoked by a client request. When a new request to a BPEL service arrives, a new instance of a BPEL program is created and further client interactions with the BPEL server are assigned to the created BPEL process until all interactions are complete. Then the BPEL process exits and disappears from the BPEL server.

2.2 Testing environment of this survey

In the commercial world, it is easy to find a product to support BPEL (see Table 1.1). However, not all products support the latest version of BPEL. We choose Apache ODE[5] as the testing environment in this survey because it supports the latest version of BPEL and is free.

Table 1.1 A comparison of [BPEL](#) engines (Source: Wikipedia [8])

Product	Vendor	Release Date	Framework	Compatibility	License
ActiveVOS	Active Endpoints	Nov 2008	Servlet or J2EE	BPEL4People / WS-HumanTask: standards	Commercial
Apache Agil	ASF	25 March 2005	Servlet	WS-BPEL	Apache v2.0
Apache ODE (was PXE)	ASF (donated by Intalio)	14 August 2006 7 June 2006	Apache Axis J2EE	BPEL4WS 1.1, WS-BPEL 2.0	Apache
Appian Enterprise	Appian	-	J2EE	BPEL, BPMN	commercial
bexee	-	December 15, 2004	<i>Apache Axis</i>	BPEL4WS 1.1 (incomplete)	open source
Business Integration Server	SEEBURGER		J2EE	BPEL	commercial
BizTalk Server	Microsoft	April 3, 2006	.NET	BPEL, BPMN, RFID	commercial
BPMS	Intalio	8 October 2006	J2EE	BPMN, BPEL, BPEL4People, XForms	freeware and commercial
Digité Enterprise	Digité, Inc.		J2EE		commercial
Fiorano BPEL	Fiorano Software Inc.		J2EE		commercial
Oracle BPM Suite	Oracle Corporation		J2EE		commercial

iBolt Server	Magic Software Enterprises		J2EE	BPEL4WS	commercial
jBPM	jBoss	12 January 2009	J2EE	WS-BPEL	LGPL
Orchestra	Groupe Bull		JOnAS	BPEL4WS 1.1	open source
Orchestrator	Cape Clear	October 2006	J2EE	BPEL 1.1	commercial
Orchestrator Server	OpenStorm		.NET or J2EE	BPEL4WS 1.1	commercial
Oracle BPEL Process Manager (formerly Collaxa BPEL Orchestration Server)	Oracle Corporation	28 Jan 2006	J2EE	WSBPEL 1.1	commercial
OSWorkflow	OpenSymphony	1 August 2006	J2EE, OSCoreJ2EE,	?	Apache
Parasoft BPEL Maestro	Parasoft		?	BPEL	commercial
PolarLake Integration Suite	PolarLake		J2EE	BPEL, FpML , XBRL	commercial
SAP Exchange Infrastructure	SAP AG			BPEL	commercial
TeamWorks Enterprise Edition	Lombardi Software		J2EE	BPMN, BPEL, BPDM, WSRP	commercial
Virtuoso Universal Server	OpenLink Software	2006		UDDI, WS-BPEL, WS-*	GPL and commercial
WebSphere Process Server	IBM	29 September 2006	J2EE	WS-BPEL	commercial

2.3 Abstract language

In order to understand the algebraic and logical properties of BPEL programs, this survey introduces an abstract language to describe its characteristic features. Mappings of BPEL activities to the higher-level language are shown in Table 1.2.

Notation	Description	BPEL Activity
skip	Do nothing. Sometimes it will be used to consume a fault in the fault handler.	<code><empty name="Empty"/></code>
$x:=e$	Assignment; e is an expression of state.	<pre> <assign> <copy> <from variable="e"/> <to variable="x"/> </copy> </assign> </pre>
inv a x y	Call a web service provided by a partner through partner link "a" with input parameter x and output variable y. (An inv operation becomes asynchronous if the output variable is omitted). A partner link can be considered as a channel to communicate with the web service described in the WSDL file.	<pre> <invoke name="InvokeF" operation="bookFlight" partnerLink="a" inputVariable="x" outputVariable="y" /> </pre>
rec a x	Hold the process and wait for a message x from partner link a. (We can consider a partner link to be a particular communication channel between the service requester and provider.)	<pre> <receive name="receiveInput" partnerLink="a" operation="Booking" variable="x" /> </pre>

rep a x	Reply result message x to partner link a.	<pre><reply partnerLink="a" operation="Receipt" variable="x" /></pre>
throw	Throw a specific fault to the fault handler.	<pre><throw faultName="Flight is full" name="Throw"/></pre>
A ; A	BPEL allows a collection of activities to be executed in sequential order through activity <i>sequence</i> . In simplified BPEL, a semi-colon (;) is used to separate the different activities that will be executed sequentially from the left.	<pre><sequence name = "SQ"> <receive name="Order"../> (A) <invoke name="chk_room"../> (A) <invoke name="book_room"../> <reply name="Confirm"../> </sequence></pre>
A ◁ b ▷ B	The basic conditional construct: A if b is true, else B.	<pre><if name="condition name"> <condition>y<x+1(b)</con...> x=x+1 (A) <else> x=x-1 (B) </else> </if></pre>
A A	The parallel notation " " is used for the concurrent execution of activities inside the flow activity.	<pre><flow> <links> .. </links> <invoke name="check_flight"../> (A) <invoke name="check hotel"../> (A) ... </flow></pre>
b * A	There are three activities in BPEL to repeat execution of a program,	<u>While activity</u>

	<p>but all abstract to iterating A if Boolean condition b holds (and otherwise skip).</p>	<pre> <while> <condition>x>0 (b)</condi..> <invoke...> (A) </while> RepeatUntil activity <repeatUntil> Activity A <condition>b</condition> </repeatUntil> ForEach activity {b is (1<=i<=5)} <forEach parallel="no" counterName="i"..> <startCounterValue>1</start..> <finalCounterValue>5</final..> Activity A(i) </forEach> </pre>
<p>$g \rightarrow A$</p>	<p>Parallel processes typically need synchronization, provided by “Link” and “joinCondition”. A Boolean condition ‘g’ guards execution of an activity inside the flow so that Activity A is enabled only when g is true (\$link1 or \$link2).</p>	<pre> <flow> ... <assign name="A"> (g) <targets> <joinCondition> \$link1 or \$link2 </joinCondition> <target linkName="link1"/> <target linkName="link2"/> </targets> (A) <copy> <from variable="e"/> <to variable="x"/> </copy> </assign> ... </flow> </pre>

<p>{A ? C, F}n</p>	<p>In BPEL, all activities are scoped, and scopes can be nested to arbitrary depth. Each scope has its variables, part links and handlers, etc. The fault handler and compensation handler are the main components that help a process to refine a specific scope when necessary. In this abstract BPEL, {A ? C, F}n is a scope with name n; it contains Activity A, compensation statements C and Fault handler F. A is basic activity or a structural activity, C is a piece of program that is used to refine A if scope n can be completed successfully. In some case, a fault may happen within A, fault handler F is used to handle the partial refinement of A.</p>	<pre> <scope name="n"> (F) <faultHandlers> <catch faultName="noHotel"> <invoke name="cancelFlight" inputVariable=Fcode /> </catch> </faultHandlers> (C) <compensationHandler> <invoke name="cancelFlight" inputVariable=Fcode /> <invoke name="cancelHotel" inputVariable=Rinfo /> </compensationHandler> (A) <sequence> <invoke name="bookFlight" inputVariable=Fcode outputVariable=Res /> <invoke name="bookhotel" inputVariable=Rinfo outputVariable=ResH /> <!-- If ResH = 0, a throw noHotel is thrown to cancel the booked flight > </sequence> </scope> </pre>
<p>↶</p>	<p>Invoke all compensation handler instances which have been installed within current scope, e.g. all compensation handler instances for s1,s2,s3 will be</p>	<pre> <scope name="N"> <faultHandlers> ... <compensate/> ... </faultHandlers> </pre>

	invoked.	<pre> <scope name="s1"> ... <scope name="s2"> ... <scope name="s3"> ... </scope> </pre>
←	Invoke a compensation handler instance of a specific scope n within current scope.	<pre> <scope name="S0"> <faultHandlers> ... <compensateScope target="n"/> ... </faultHandlers> <scope name="s1"> ... <scope name="s2"> ... <scope name="n"> ... </scope> </pre>

3. BPEL activities

A BPEL business process contains two kinds of activity: a basic activity and a structured activity. A basic activity performs its intended purpose without containing further activities. (E.g., assign, empty, receive, reply, invoke, etc.) However, a structured activity contains other activities. (E.g. flow, sequence, if, while, etc.) The following sections analyse features of different activities in BPEL according to the BPEL specification 2.0 [4].

$A = A \text{ or } A ; A$ if A is a structural activity

In the following we use, as is usual, annotations to summarise program logic: if P is a program and q is a predicate on state, then $P \{q\}$ means that execution of P terminates in a state satisfying q (from any initial state).

3.1 Basic activity

3.1.1 Do nothing <empty> (skip)

The simplest basic activity in BPEL does nothing and so acts as an identity element in the program algebra. Sometimes it is used to consume a fault of a fault handler if there is no action for the fault. It is both a left and right identity for sequential composition:

$A ; \text{skip} = \text{skip} ; A = A$.

3.1.2 Assignment <assign> (x:=e)

Another basic activity is assignment: it copies the value of an expression to a variable. In some case, a shared variable between parallel \parallel processes is protected if the assignment activity is inside a scope with attribute *isolated* = true.

For example, in that case

$x:=1; y:=x \parallel x:=2; z:=x \{y=1 \text{ and } z=2\}$

whilst without isolation the postcondition is $\{y=1,2 \text{ and } z=1,2\}$.

3.1.3 Receive activity <receive> (rec a x)

The purpose of a *receive* activity is to hold a business process waiting to receive messages from a communication channel (a partner link). A new message received from the channel will invoke the BPEL server to create a process instance of the BPEL program. In the following example if a message *m* is received from channel *a*, then the value of *x* becomes *m* after execution of *rec a x*.

```
rec a x {x=m}
```

3.1.4 Reply activity <reply> (rep a x)

The *reply* activity is paired with the *receive* activity; a reply activity without a corresponding *receive* activity will make the process throw a fault on reaching the end of the business process [4]. A reply will not affect the state of a business process. Continuing the example in 2.1.3, execution of *rep a x* means that a message *x* is sent to channel *a* and stored in *m*.

```
rec a x {x=m}; x:=x+1 {x=m+1}; rep a x {m=x+1}
```

3.1.5 Invoke activity <invoke> (inv a x y)

The *invoke* activity is used to call a web service provided by a partner through a partner link. A partner link can be considered a channel to communicate with the web service described in the WSDL file. The type of the link is also defined in the WSDL file. As a web service can perform different kinds of operation defined in the WSDL file, the name of the operation must be declared when a service is called. An input variable is required to be passed to the service as parameter of the operation. Another variable that can be passed to the service is the output variable. It is an optional variable, if an output variable *y* is defined in the service call, the program counter will stop and wait for the answer *y* before going to the next counter. On the contrary, if the output variable of a web service call is omitted, the call is asynchronous (the program counter will go to the next counter immediately after the service is called).

(Asynchronous)e.g. *inv a y; m=y+1 || rec a x; p=x+n; rep a p*

Global PC	WS 1	WS 2
1)	<i>inv a y</i>	<i>rec a x {x=y}</i>
2)	<i>m=y+1 {m=y+1}</i>	<i>p=x+n {p=y+n}</i>

(Synchronous) e.g. `inv a y,z; m=z+y || rec a x; p=x+n; rep a p`

Global PC	WS 1	WS 2
1)	<code>inv a y,z</code>	<code>rec a x {x=y}</code>
2)		<code>p=x+n {p=y+n}</code>
3)	<code>{z=y+n}</code>	<code>rep a p</code>
4)	<code>m=z+y {m=2y+n}</code>	

3.1.6 Throw activity <throw/> (throw)

A *throw* activity is used to throw a specific fault in an immediately enclosing scope.

If a throw happens in a scope, the remaining activities of the scope are not executed and the throw will be handled by the fault handler, if the fault handler can handle the fault; otherwise the scope is completed unsuccessfully, and the compensation handler instance of the scope will not be installed. The fault propagates to the outer scope if the fault handler of the current scope cannot catch the throw or a *rethrow* activity is executed.

`{throw; A} = throw`

3.2 Structured activity

3.2.1 Sequence activity <sequence> (A;A)

BPEL allows a collection of activities to be executed in sequential order through activity *sequence*. In this paper, a semi-colon (;) is used to separate the different activities that will be executed sequentially from the left. The operator is associative, so parentheses can be omitted, as in

`A1 ; A2 ; A3.`

3.2.2 Flow activity <flow> (A||A)

For improving performance, processes are allowed to present in parallel if there is no interference between them. The parallel processes inside a *Flow* activity can be basic activities or structured activities.

E.g. $\text{inv } x \ y \parallel A2 \parallel \text{inv } a \ n$

3.2.3 Scope-based compensation statement (A ? C, F)

In BPEL scope, a compensation handler is a piece of program to undo a completed process step (scope). A compensation handler instance will be created and installed after a scope has been completed successfully[4]. When a completed scope is to be undone, the installed instance of the compensation handler of the scope can be invoked by using *compensateScope* <name of scope>: a piece of program is executed to compensate the undoing scope and the instance of the compensation handler of the scope is uninstalled. In some cases, all completed scopes are needed to rollback; then *compensate* is used to invoke all installed compensation handler instances.

In simplified BPEL

$\{A, \dots\}n : \{..\}n$ is a non-empty scope n which contains at least one activity A .

$\{A? C, \dots\}n$: C is the compensation of scope n , which is used to undo A if A is completed successfully. In fact, compensation in BPEL is just a methodology to let programmer to refine the activity A after it is completed, it may not be a fully undo to A and it depends on the programmer how to design the C here.

$A; C = \text{skip}$ /*may not be true, it depends on the coding of C */

$\{A? C, F\}n$: F is the fault handler, it can be considered a partial compensation of A . If a fault is thrown within A , F is a way for programmer to undo the partially completed tasks of A .

$A; \{A1?C,F\} = A; \{A11;fault;A13?C,F\} = A$ /* if F has the capability to undo $A11$ */

Restrictions of compensation in BPEL syntax

$\{ \{ \{ A1 ? C1 \} x ; \{ A2 ? C2 \} y ? C3, F3 \} z ? C, F \} w$

1) *compensateScope* n (undo scope n) or *compensate* (undo all scopes) in the handler can invoke ONLY the compensation handler instances of the immediately enclosing scope.

E.g. $F3$ or $C3$ can invoke $C1, C2$ only. □

2) As the compensation handler refines only the current scope, if there are any scopes that are enclosed in the refining current scope, the compensation handler of the current

scope can be used to invoke instances of the compensation handler of the enclosed scopes to prevent rewriting the compensation logic.

E.g. $\neg z$ in C invokes C3, $\neg x$ and $\neg y$ in C3 invokes C1, C2....

3) $\neg N(n)$ or $\neg N$ can be used **ONLY** within the fault handler, compensation handler or termination handler.

3.2.4 Compensation within repeatable constructs

In some cases, a scope with associated compensation handler is enclosed in a repeatable construct, e.g. `<while>`, `<repeatUntil>`. The result is called a Compensation Handler Instance Group. A compensation handler instance group contains the same number of compensation handlers instances as the number of successfully completed scopes in the repeatable construct. "If an uncaught fault occurs while executing the compensation handler instance within the instance group, all running instance will be stopped and the remaining handler instances will be uninstalled." [4]

The following is a simple example of creating a compensation handler instance group. A while loop, initialised with $X=0$, contains a scope: it increases X by 1, and loops five times. The compensation of the scope decreases X by 1. Assume a user-defined-fault is thrown after the while loop is complete; all compensation instances are invoked and X is reset to 0.

A fault is thrown after the repeatable constructs is completed successfully	Post Condition
$X:=0; X<5 * (\{X:=X+1 ? X=X-1, \neg N\}); \text{throw}$	$\{X=0\}$

In BPEL

$X:=0$

`<While>`

`<condition> X < 5 </condition><scope n>`

user-defined-fault: { compensate }

compensation_handler_n {X:=X-1}

$X:=X+1$

`</scope n>`

```
</While> {X=5}  
throw user-defined-fault  
{X=0}
```

4. Case study – Hong Kong SunFlower Travel

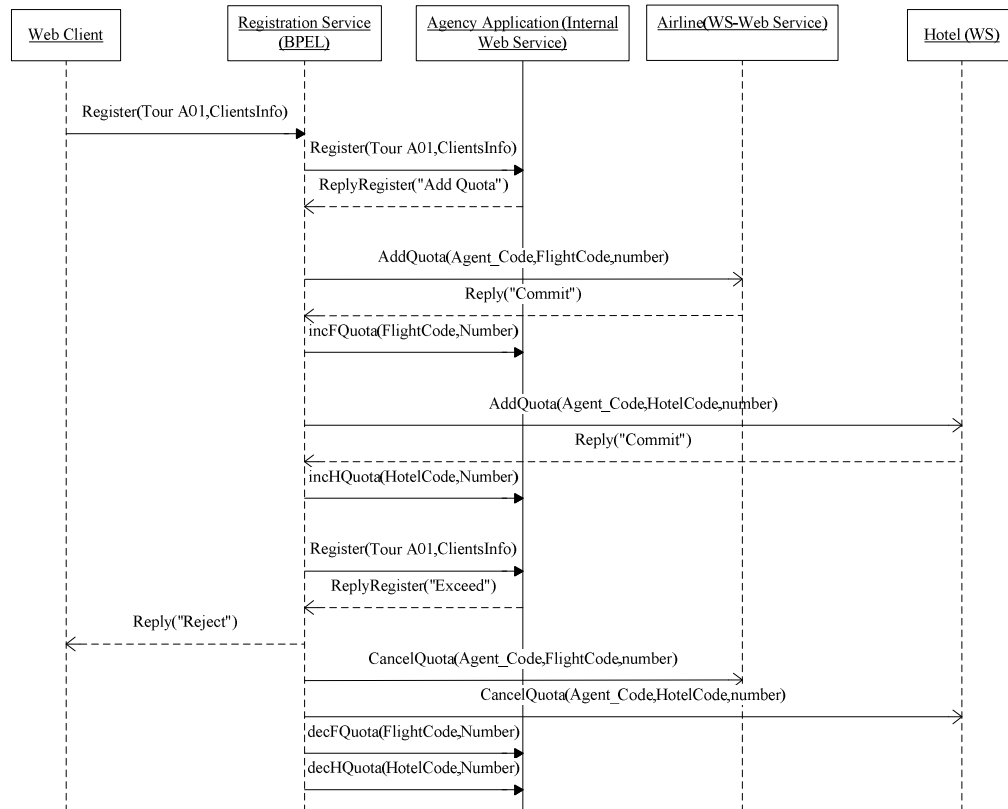
With the extension of business in SunFlower Travel, the CEO decides to automate. In Hong Kong, travel agencies have an agreement with airlines that they can reserve tickets with low prices in different seasons. According to different situations, agencies may get different numbers of seats for a specific flight. But all the travel agencies have to confirm the number of seats three days before the flight, when ticket payment is processed and tickets confirmed.

The process of preparing a tour starts by getting the seat quota for a specific city. SunFlower designs different tours according to the seat quota and different standards of hotel. A tour code is used to identify tours and every tour requires a minimum of ten members. Anyway, the total number of clients on tour to the same city can not exceed the quota assigned by the airline. Clients interested in the tour can be registered in its branches or other cooperating agencies. Cooperating agencies receive payment from clients and transfer money to the host agent.

In General, SunFlower expects to receive many clients on every tour, but it is restricted by the airline's seat quota. SunFlower can apply to the airline for a higher allocation if the number of clients to a city exceeds its quota. Since the number of clients in a tour is restricted by space, a tour can not have more than 30 clients. In that case, compensation should occur if the maximum number of clients is reached after increasing the airline quota.

We have designed a service in simplified BPEL to handle the following scenario. A client registers a tour on the web (BPEL service); BPEL forwards the request to the internal web service of SunFlower; finds from the internal service of SunFlower that the flight quota is insufficient; sends a web service request to the airline and hotel agency to increase the quotas and sends a request to the internal web service again. Unfortunately, the maximum number of clients of the tour is reached while executing the quota application, so the registration is rejected and compensation to the airline and hotel agency is invoked.

Compensation of registration when maximum number of a tour is reached.



Registration Service (BPEL)

rec w x /* receiving request x from web client */

inv i, x, y /* call internal service through channel i to register customer described in x and wait for reply info. in y */

rep_res:= "Accept" ◁ y="success" ▷

{A ? C, F}n; /* try to increase quota from airline and hotel agency */

inv i, x, z ; /* invoke request to interanal service again */

```

        /* if the second apply is not success, send a throw to invoke the
        fault handler of current scope ("not scope n") and compensate A of scope
        n */
    ( rep_res:= "Accept" <z="success" >rep_res:= "Reject"; throw ))

/* if AddQuota is reply from internal application, execute above statements */
<y="AddQuota" > rep_res:="Reject"

rep w, rep_res /* reply result to web client */

```

where

Fault handler of current scope is: $\leftarrow n$ /* invoke C */

A of scope n is :

```

    inv a, f_info, fres; /* request flight quota in airline */
    inv i, inc_fq < fres="Commit" > throw /* increase flight quota in internal
    application if the flight quota request from airline is committed */
||
    inv h, h_info, hres; /* request hotel quota in hotel agency */
    inv i, inc_hq < hres="Commit" > throw /* increase hotel quota in internal application if
    the hotel quota request from hotel agency is committed */

```

C of scope n is :

```

    inv a, f_info; /* cancel flight quota in airline*/
    inv h, h_info; /* cancel hotel quota in hotel agency*/
    inv i, dec_fq /* dec flight quota in internal application*/
    inv i, dec_hq /* dec hotel quota in internal application*/

```

F of scope n is:

```

/* cancel flight quota in airline*/
    inv a, f_info; inv i, dec_fq < fres="Commit" > skip
/* cancel hotel quota in hotel agency*/
    inv h, h_info; inv i, dec_hq < hres="Commit" > skip

```

5. Discussion

During this investigation of BPEL from the viewpoint of program algebra, a feature not specifically covered by the BPEL standard [4] has been observed which may generate unpredictable, and hence undesirable, results. It happens when a fault is thrown in the process of installing a compensation handler instance group.

To demonstrate it, we modify the example in 2.2.3 (Compensation within repeatable constructs) to throw a user-defined-fault inside the while loop when $X > 2$. While $X = 3$, there should be three compensation handler instances installed. If the fault handler execute *compensate* to invoke all installed compensation handler instances while $X=3$, the value of X should reset to 0. By inspection of the real example, only the last compensation instance is executed, returning 2.

In Simplified BPEL

A fault is thrown inside the loop	Post Condition
$X:=0; X<5 * (\{X:=X+1 ? X=X-1, \neg N\}; \text{throw } \langle X > 2 \rangle)$	$\{X=2\}$

In BPEL

$X:=0$

<While>

<condition> $X < 5$ </condition><scope n> *user-defined-fault: { compensate }*

compensation_handler_n {X:=X-1}

$X:=X+1$

</scope n>

if $x > 2$ then

throw user-defined-fault</While>

$\{X=2\}$

To solve this problem, it suffices for the compensation handler of the instance group to have the capability of handling the situation while the iteration construct is not complete.

6. Conclusion

In this paper, a high level language has been used to describe BPEL programs, concentrating on characteristic features of BPEL, like scope-based compensation and fault handling. It is hoped that this approach will form the basis for a future study of the laws of BPEL programming. One immediate benefit has been the identification of an undesirable feature not address in the BPEL standard [4]. An approach to its solution has been offered.

It is concluded that further study of the abstract language introduced here is justified and that the use of program logic and algebra provide useful techniques for the accountable correctness of BPEL code.

References

1. Qiu Zongyan, Wang Shuling, Pu Geguang, and Zhao Xiangpeng, Semantics of BPEL4WS-like Fault and Compensation Handling, LMAM and Department of Informatics, School of Math., Peking University, Beijing 100871, China
2. Huibiao Zhu, Jifeng He, Jing Li, Geguang Pu, Jonathan P.Bowen, Linking Denotational Semantics with Operational Semantics for Web Services, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
3. Chenguang Luo, Shengchao Qin, and Zongyan Qiu, Verifying BPEL-like Programs with Hoare Logic, LMAM and Department of Informatics, School of Math., Peking University
4. Web Services Business Process Execution Language Version 2.0, OASIS, Public Review Draft, 23th August, 2006
5. Web Services Business Process Execution Language Version 2.0 (Primer), OASIS, 9 May 2007.
6. Antony Miguel, WS-BPEL 2.0 Tutorial, http://www.eclipse.org/tptp/platform/documents/design/choreography_html/tutorials/wsbpel_tut.html, 13th October, 2005.
7. Computation orchestration: A basis for wide-area computing. In M.Broy, J.Gruenbauer, D.Harel and C.A.R.Hoare, editors, Proceedings of the NATO Advanced Study Institute on Engineering Theories of Software Intensive Systems, NATO ASI Series, Marktoberdorf, 195, 2004.
8. Wikipedia survey.

Appendix: Table of acronyms

BPEL: Business Process Execution Language

WSDL: Web Services Definition Language

XML: Extensible Markup Language

XSD: XML Schema Definition