Compensations in Orchestration Languages

Gianluigi Zavattaro
Joint work with C. Guidi, I. Lanese, F. Montesi

Joint FOCUS Research Team
INRIA / University of Bologna

Figure 10.32 provides an example of a Sub-Process that includes three Event Sub-Processes. The first Event Sub-Process is triggered by a Message, does not interrupt the Sub-Process, and can occur multiple times. The second Event Sub-Process is used for compensation and will only occur after the Sub-Process has completed. The third Event Sub-Process handles errors that occur while the Sub-Process is active and will stop (interrupt) the Sub-Process if triggered.

Figure 10.32 - An example that includes Event Sub-Processes
Plan of the Talk

- Long-Running Transactions (LRTs) [NestedSagas]
- A renewed interest in LRTs [BPMN, WS-BPEL]
- The JOLIE orchestration language
- Dynamic compensations in JOLIE
Plan of the Talk

- **Long-Running Transactions (LRTs)** [NestedSagas]
- A renewed interest in LRTs [BPMN, WS-BPEL]
- The JOLIE orchestration language
- Dynamic compensations in JOLIE
Data Processing Application: an example

- **Purchase order**
  - A transaction composed of sub-transactions
A First Solution

- Use of nested (standard) transactions
- Standard transactions are ACID
  - A = atomic (all or nothing)
  - C = consistent (w.r.t. the application logic)
  - I = isolated (unobservable)
  - D = durable (persistent)
- ACIDity implies a perfect roll-back
- Not satisfactory
  - The whole transaction may require a long period: resources may be locked for the whole transaction
Nested Sagas (1)

- Compensations are provided
  - No perfect roll-back
  - No isolation
Nested Sagas (2)

- Sagas can be nested

DIAGRAM: Sagas can be nested within other sagas, creating a hierarchy of processes.
An exception handler can be associated to each Saga
Plan of the Talk

- Long-Running Transactions (LRTs) [NestedSagas]
- A renewed interest in LRTs [BPMN, WS-BPEL]
- The JOLIE orchestration language
- Dynamic compensations in JOLIE
Internet Technologies

Connectivity

Presentation

Programmability

TCP/IP

HTML

XML

Web Pages

Web Services

Browse the Web

Program the Web

Technology

Innovation

Browse the Web
Web Service Orchestration

- **WS-BPEL** [OASIS standard]:
  Language for Web Service Orchestration
  - Description of the message exchanged among Web Services that cooperate in a business process

- **BPMN** [OMG standard]:
  - Graphical notation for business procedures
BPMN: Business Process Modeling Notation

- Selection of a Nobel Prize laureate

Diagram:
- Nominate: Identify Potential Nominee(s) → Send Nominee Completed Form(s) → Nomination Form(s) Sent → Nominator may nominate one or more Nominees
- Field Assembly: Disclose Nominations (Meeting 1) → Select Laureates (Meeting 2) → Announce Nobel Prize Laureates → Announcement Made
- Process: Around 3000 invitations/ confidential nomination forms are sent to selected Nominators
- Nominators: Send Nomination Form → Collect Completed Form(s) → Screen & Select Preliminary Candidates → Determine Need for Expert Assistance → Send List of Selected Preliminary Candidates → Collect Candidates Work Assessment Reports → Select Final Candidates and their works → Write Recommendations Report → Submit Report with Recommendations → Hold Nobel Prize Award Ceremony
- Report with Recommendations: No
- Expert Assistance Required? Yes
BPMN: Long Running Transactions

- An activity can have a corresponding compensation activity
- This is triggered by a “compensate” event
BPMN: Long Running Transactions

- Also user defined compensation handlers can be programmed
LRTs in WS-BPEL

```xml
<scope name="mainScope">
  <faultHandlers>
    <catchAll>
      <compensateScope target="invoiceSubmissionScope" />
    </catchAll>
  </faultHandlers>
  <sequence>
    ...<br />
  </sequence>
  <scope name="invoiceSubmissionScope">
    ...
    <compensationHandler>
      <invoke name="withdrawInvoiceSubmission" ... />
    </compensationHandler>
    <invoke name="submitInvoice" ... />
  </scope>
  ...
  <!-- do additional work -->
  <!-- a fault is thrown here;
       results of invoiceSubmissionScope must be undone -->
</scope>
```

REVER - Paris 19/1/2012
Plan of the Talk

- Long-Running Transactions (LRTs) [Nested Sagas]
- A renewed interest in LRTs [BPMN, WS-BPEL]
- The JOLIE orchestration language
- Dynamic compensations in JOLIE
JOLIE: programming orchestrators with a C / Java like syntax

```java
execution { concurrent }

cset { request.id }

interface myInterface {
    OneWay: login
    RequestResponse: get_data
}

inputPort myPort {
    Protocol: http
    Location: "socket://localhost:2000"
    Interfaces: myInterface
}

main {
    login( request ) ;
    get_data( request )( response ) {
        response.data = "your data" + request.id
    }
}
```

REVER - Paris 19/1/2012
Data are exchanged by means of operations

Two types of operations:
- One-Way: receives a message;
- Request-Response: receives a message and sends a response to the caller.

A: main
   { sendNumber@B( 5 ) }

B: main
   { sendNumber( x ) }

A sends 5 to B through the sendNumber operation.
JOLIE: basic communication primitives

Data are exchanged by means of operations

Two types of operations:
One-Way: receives a message;
Request-Response: receives a message and sends a response to the caller.

A:
main{
    twice@B(5)(x)
}

B:
main{
twice(x)(result){
    result = x * 2
}
}

A sends 5 to B;
B doubles the received value;
B sends the result back to A.
JOLIE: communication ports

A should know how to contact B
B should expose the operation “twice”

Two types of ports:

- Input ports: expose operations
- Output ports: bind output operations to input operations

A:

```
main
{
    twice@B( 5 )( x )
}
```

B:

```
main
{
    twice( x )( result )
        {result = x * 2}
}
```
JOLIE: communication ports

**A should know how to contact B**

**B should expose the operation “twice”**

**Two types of ports:**

- **Input ports:** expose operations
- **Output ports:** bind output operations to input operations

A:

```java
inputPort MyInput {
    Location: "socket://localhost:8000/"
    Protocol: soap
    RequestResponse:
        twice(int)(int)
}
```

```
main
{
    twice@B( 5 )( x )
}
```

B:

```java
main
{
    twice( x )( result )
    { result = x * 2 }
}
```
JOLIE: communication ports

A should know how to contact B
B should expose the operation “twice”

Two types of ports:

*Input ports: expose operations*

*Output ports: bind output operations to input operations*

```java
outputPort B {  
  Location:  
    "socket://192.168.1.2:8000/"  
  Protocol:  
    soap  
  RequestResponse:  
    twice(int)(int)  
}

main  
{
  twice@B( 5 )( x )
}
```

```java
inputPort MyInput {  
  Location:  
    "socket://localhost:8000/"  
  Protocol:  
    soap  
  RequestResponse:  
    twice(int)(int)  
}

main  
{
  twice( x )( result )  
    {result = x * 2}
}
```

REVER - Paris 19/1/2012
JOLIE: work- and control-flow

Basic activities can be combined with sequence, parallel and choice constructs...

sequence: \( \text{send}@\text{S}(x) \); \text{receive}(\text{msg}) \)

parallel: \( \text{send}@\text{S}(x) \) | \text{receive}(\text{msg}) \)

choice: \[
\begin{array}{l}
[\text{recv1}(x)] \{ \ldots \} \\
[\text{recv2}(x)] \{ \ldots \}
\end{array}
\]

... as well as the usual control flow constructs

if then else: \( \text{if}(x > 1) \{ \ldots \} \) \text{else} \{ \ldots \} 

for: \( \text{for}(i = 0, i < n, i++) \{ \ldots \} \)

while: \( \text{while}(i < 0) \{ \ldots \} \)
Plan of the Talk

- Long-Running Transactions (LRTs) [NestedSagas]
- A renewed interest in LRTs [BPMN, WS-BPEL]
- The JOLIE orchestration language
- Dynamic compensations in JOLIE
Statically Defined Hierarchy of Scopes

```plaintext
main
{
    scope(carRepair){
        { scope(carRental){
            ...
        } |
        scope(garage){
            ...
        }
    } ;
    scope(towingTrack){
        ...
    }
}
```

```
```

```
```
Fault handling

```plaintext
main {
    scope(carRepair){
        { scope(carRental){
            ...
        } |
        scope(garage){
            ...
        }
    } ;
    scope(towingTrack){
        ...
        throw(noTowTrack);
    }
}
```
Fault handling

- Scopes have a name $q$, an activity $P$, and a set of fault handlers $H$
- They are organized in a hierarchy
- When a fault is raised, it goes up in the hierarchy until it reaches a handler
- While going up, parallel scopes are interrupted
Fault handling

- Scopes have a name $q$, an activity $P$, and a set of fault handlers $H$
- They are organized in a hierarchy
- When a fault is raised, it goes up in the hierarchy until it reaches a handler
- While going up, parallel scopes are interrupted
Fault handling

- Scopes have a name $q$, an activity $P$, and a set of fault handlers $H$
- They are organized in a hierarchy
- When a fault is raised, it goes up in the hierarchy until it reaches a handler
- While going up, parallel scopes are interrupted

```
throw(f)
```

```
(q_1, T_1)
```

```
q_1
```

```
(f, Q)
```

```
q_2
```

```
(q_2, T_2)
```
Fault handling

- Scopes have a name q, an activity P, and a set of fault handlers H
- They are organized in a hierarchy
- When a fault is raised, it goes up in the hierarchy until it reaches a handler
- While going up, parallel scopes are interrupted
Fault handling

- Scopes have a name q, an activity P, and a set of fault handlers H
- They are organized in a hierarchy
- When a fault is raised, it goes up in the hierarchy until it reaches a handler
- While going up, parallel scopes are interrupted
Fault handling

- Scopes have a name $q$, an activity $P$, and a set of fault handlers $H$
- They are organized in a hierarchy
- When a fault is raised, it goes up in the hierarchy until it reaches a handler
- While going up, parallel scopes are interrupted
Fault handling

- Scopes have a name \( q \), an activity \( P \), and a set of fault handlers \( H \)
- They are organized in a hierarchy
- When a fault is raised, it goes up in the hierarchy until it reaches a handler
- While going up, parallel scopes are interrupted
Dynamic fault handling

- In Nested SAGAS, WS-BPEL, BPMN, etc. the fault handlers are statically defined
- In JOLIE fault handlers can be dynamically modified
  - We use an installation primitive that explicitly installs the handlers
  - The new handlers can be defined as modifications of the previous ones
Dynamic installation of handlers

\[ \text{inst}(f, Q) \rightarrow (f, Q) \]

\[ \text{inst}(f, R; cH) \rightarrow (f, R; Q) \]
Example

- Consider:

\[
\{\text{throw}(f) \mid \text{while } (i < 100) \text{ if } i \% 2 = 0 \text{ then } P \text{ else } Q, H\}_q
\]

- When \( f \) is thrown, execute \( P' \) and \( Q' \) to undo the instances of \( P \) and \( Q \) in the order in which they have been executed

\[
\{ \text{throw}(f) \mid \text{while } (i < 100) \text{ if } i \% 2 = 0 \\
\quad \text{then } P;\text{inst}(cH;P') \\
\quad \text{else } Q;\text{inst}(cH;Q'), H\}_q
\]
Compensation handler

- When a scope terminates, its last termination handler becomes its compensation handler.

When a scope terminates, its last termination handler becomes its compensation handler.
Example

- Reserve a hotel and a public transportation
  - Take the train, or in case of failure (notified with $fT$) take a bus

\[
\{ \text{inst}([fT \mapsto \text{Bus}; \text{inst}([q \mapsto cH; \text{revBus}]()])); \text{Hotel; inst}([q \mapsto \text{revHotel}])); \text{Train; inst}([q \mapsto cH; \text{revTrain}])\}_{q}
\]
Faults and Request-responses

- The JOLIE fault handling mechanism does not spoil request-responses
- In this way non-trivial distributed fault handling policies can be programmed
Faults on server side

- A client asks a payment to the bank, the bank fails
- In ActiveBPEL (a largely used BPEL engine) the client receives a generic “missing-reply” exception
- Our approach
  - The exact fault is notified to the client
  - The notification acts as a fault for the client
  - Suitable actions can be taken to manage the remote fault
Faults on client side

- A client asks a payment to the bank, then fails before the answer
- In BPEL the return message is discarded
- Our approach
  - The return message is waited for
  - The handlers can be updated according to whether or not a non-faulty message is received
  - The remote activity can be compensated if necessary
Conclusion and Future work:

- We have seen some model for compensation
- Future work:
  - How to combine reversibility and compensation?...