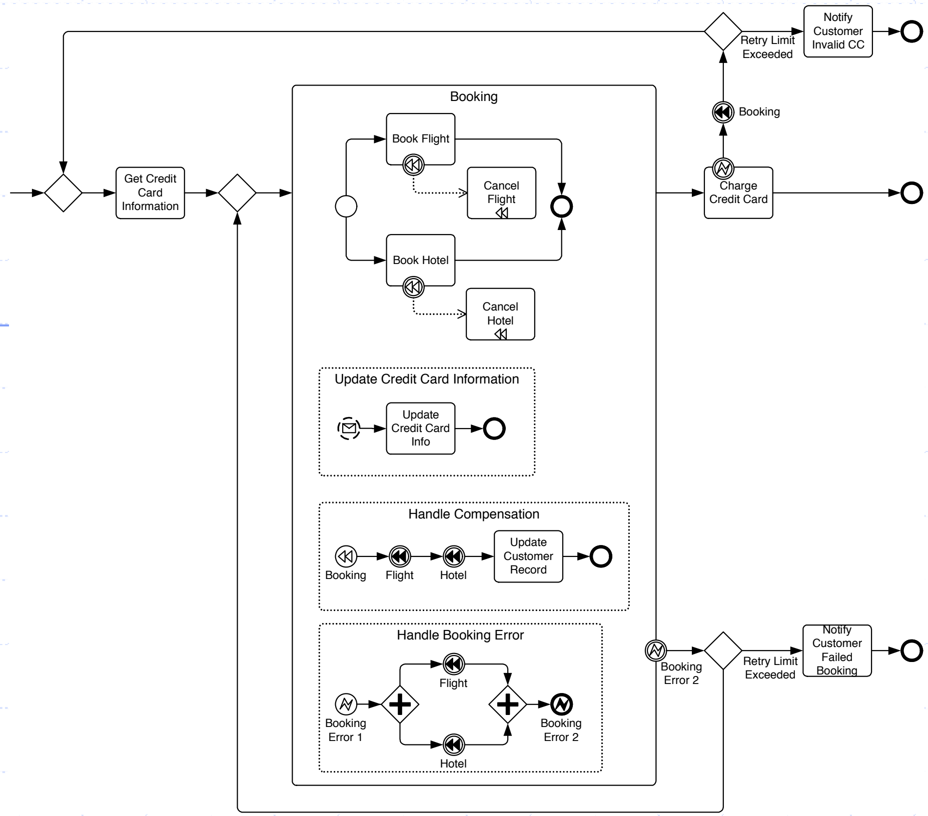
Compensations in Orchestration Languages



Gianluigi Zavattaro

Joint FOCUS Research Team
INRIA / University of Bologna

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

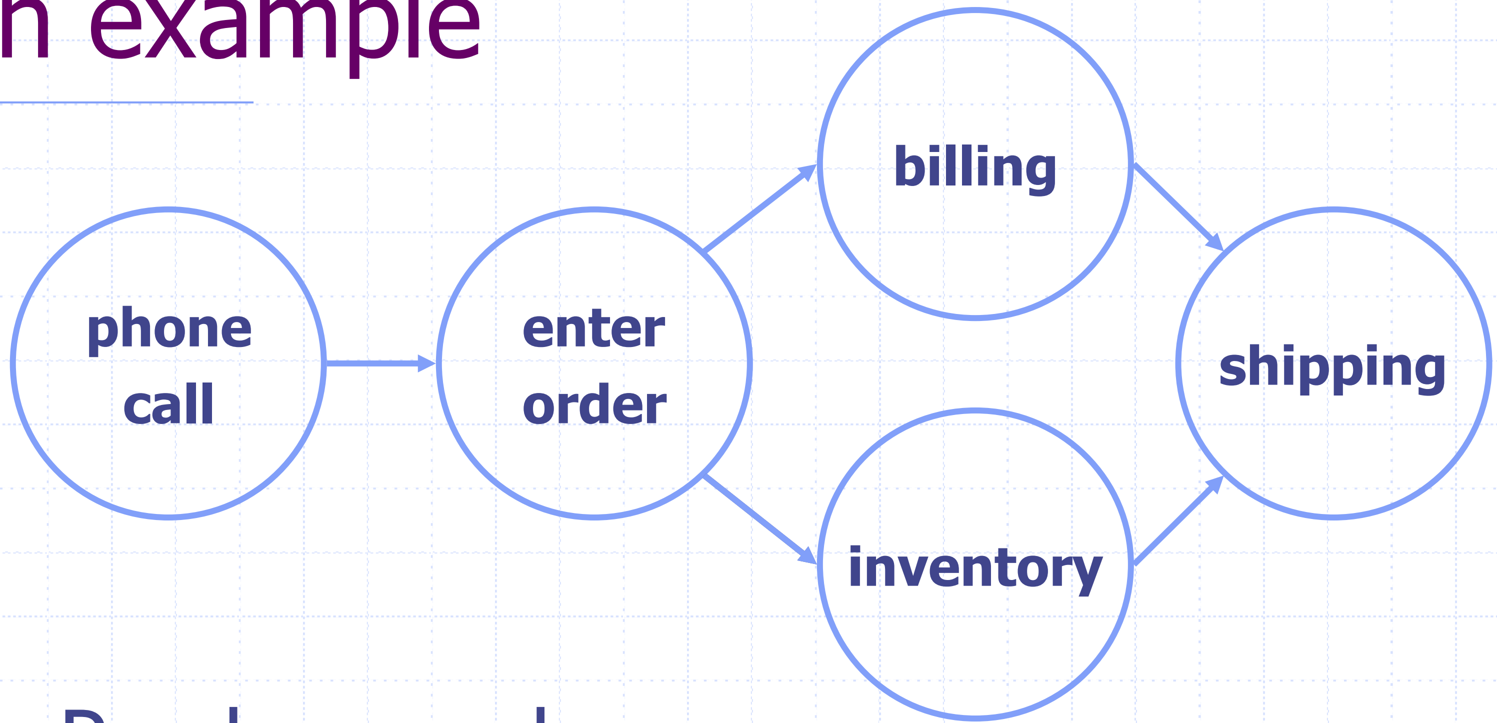
Plan of the Talk

- 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

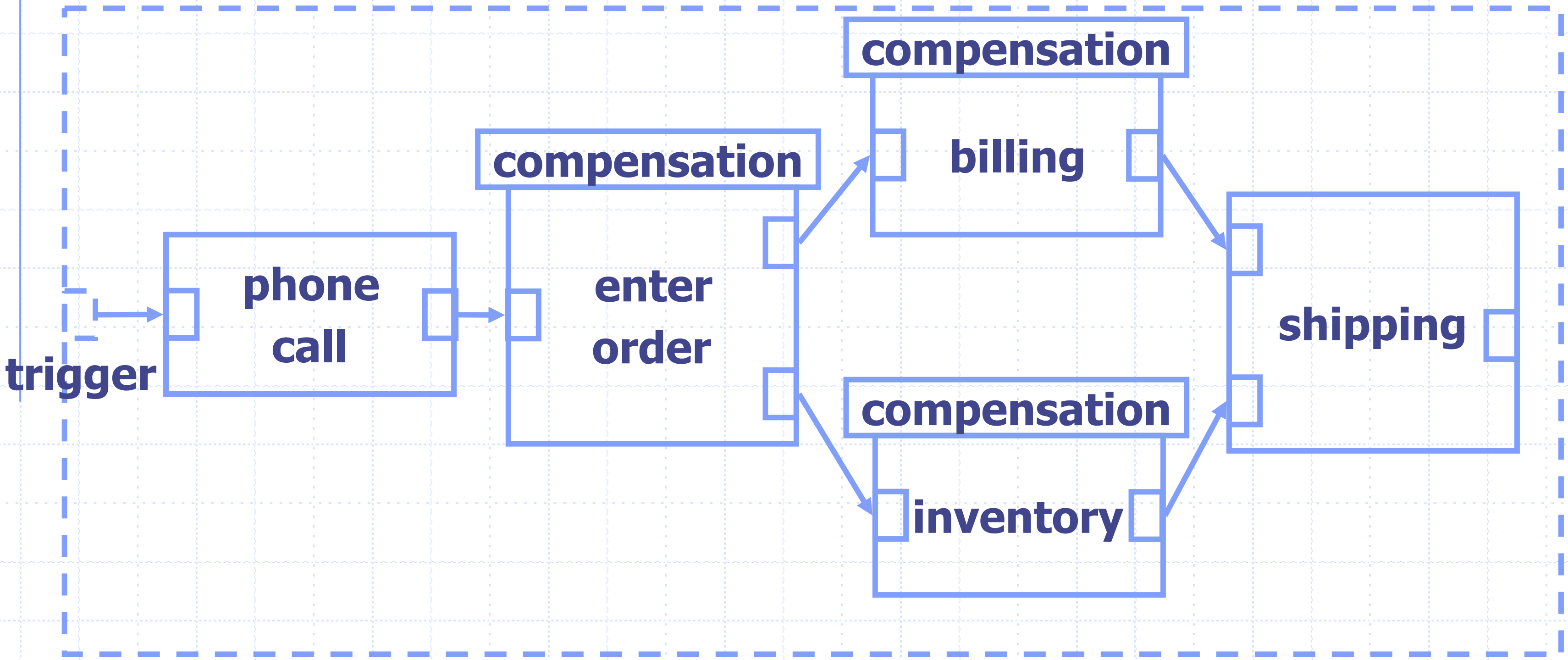


- Purchase order
  - A transaction composed of sub-transactions

A First Solution

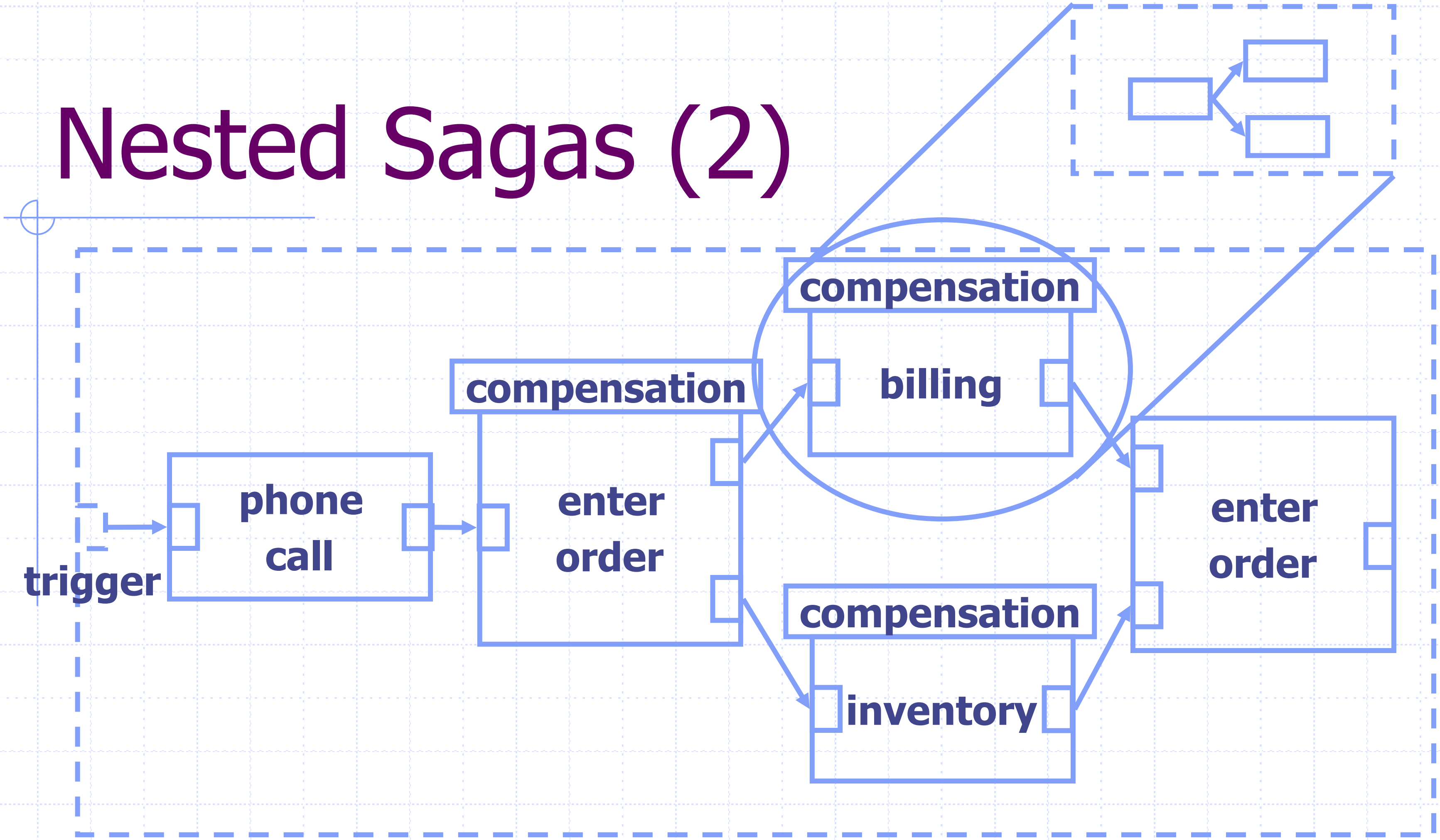
- Use of nested (standard) transactions
- Standard transactions are ACID
  - A = atomic (all or nothing)
  - C = consistent (w.r.t. the application logic)
  - I = isolated (unobservable)
  - D = durable (persistent)
- ACIDity implies a perfect roll-back
- Not satisfactory
  - The whole transaction may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

Nested Sagas (1)



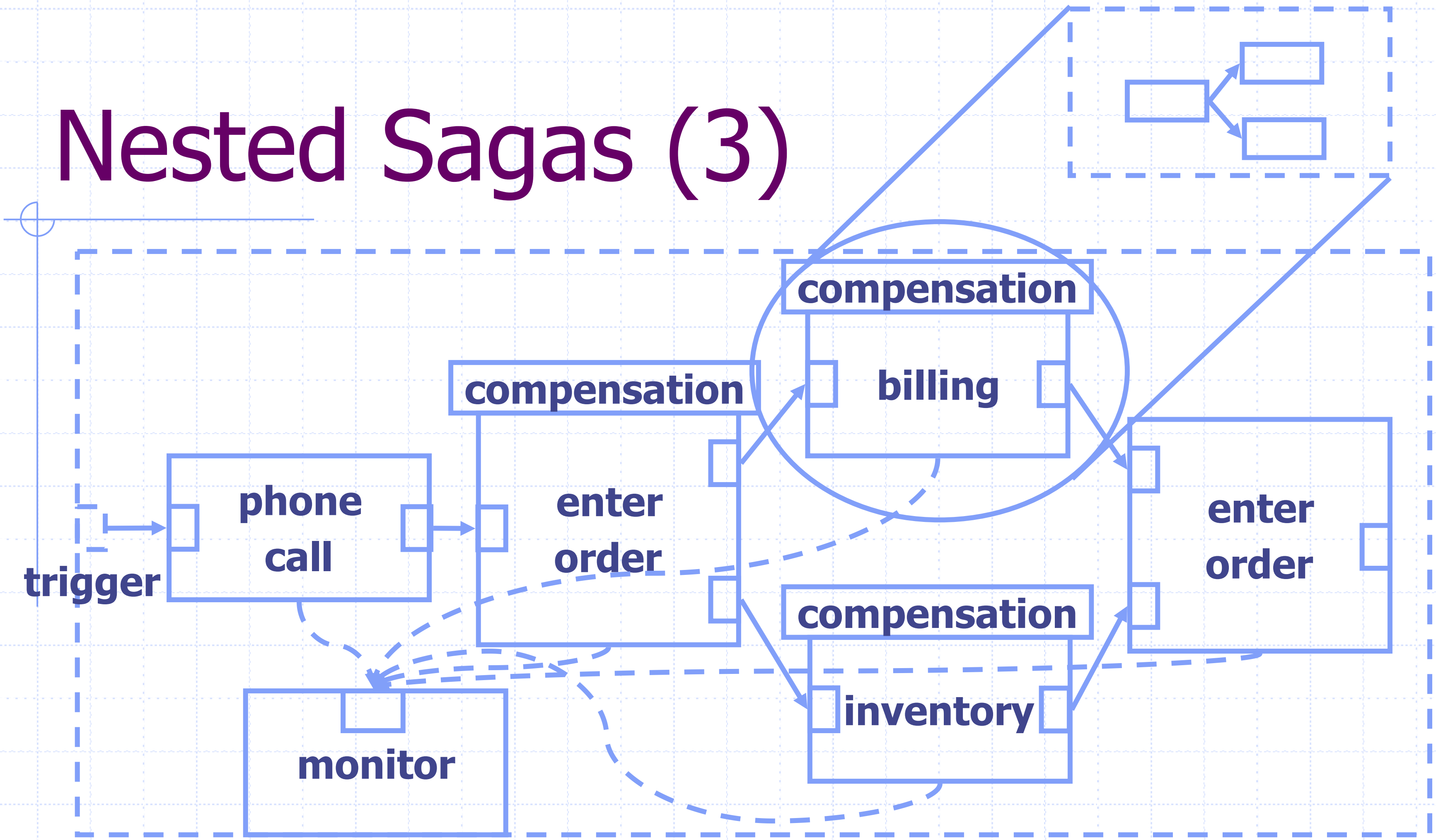
- Compensations are provided
  - No perfect roll-back
  - No isolation

Nested Sagas (2)



- Sagas can be nested

Nested Sagas (3)

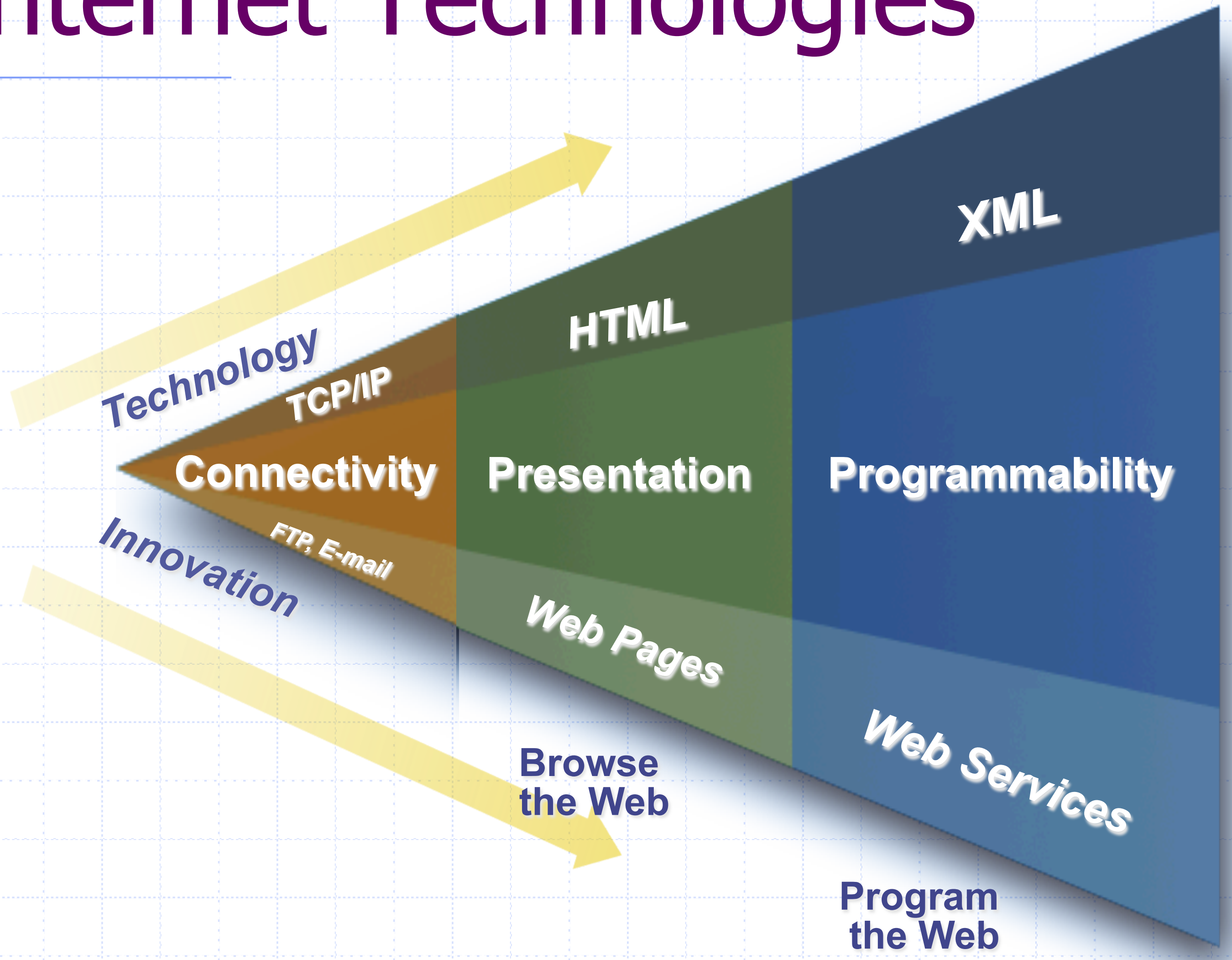


- An exception handler can be associated to each Saga

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

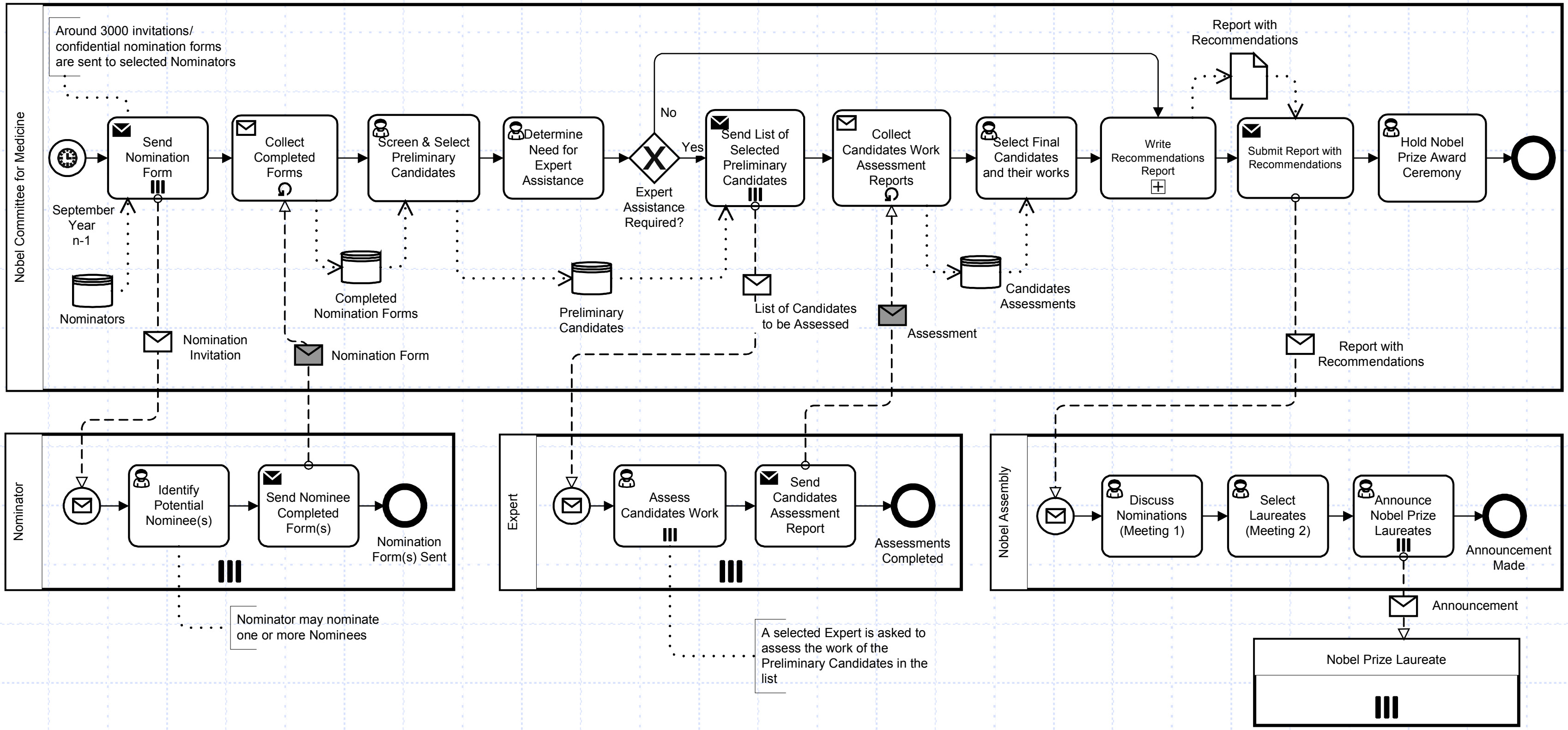


Web Service Orchestration

- **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

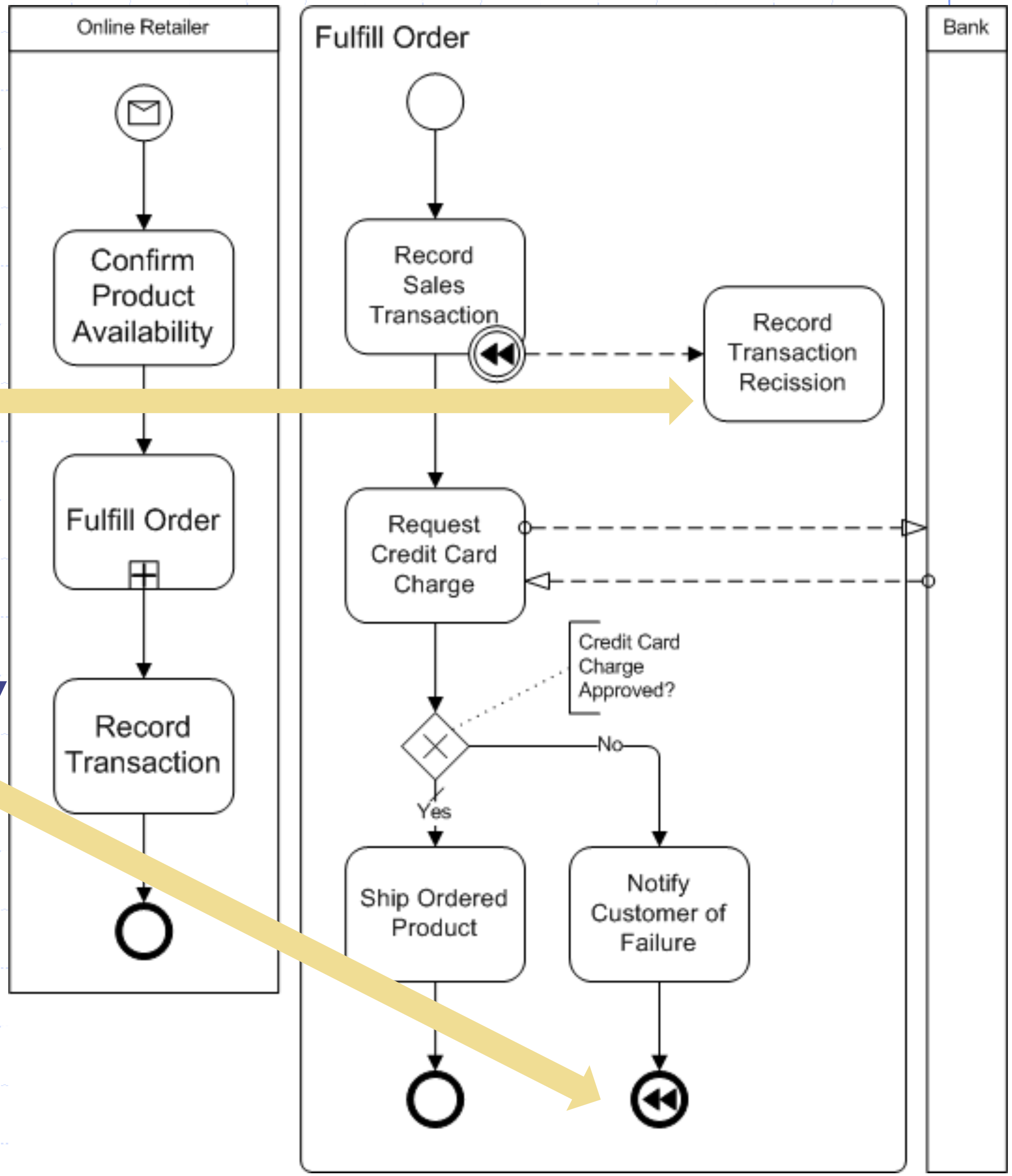
BPMN: Business Process Modeling Notation

- Selection of a Nobel Prize laureate



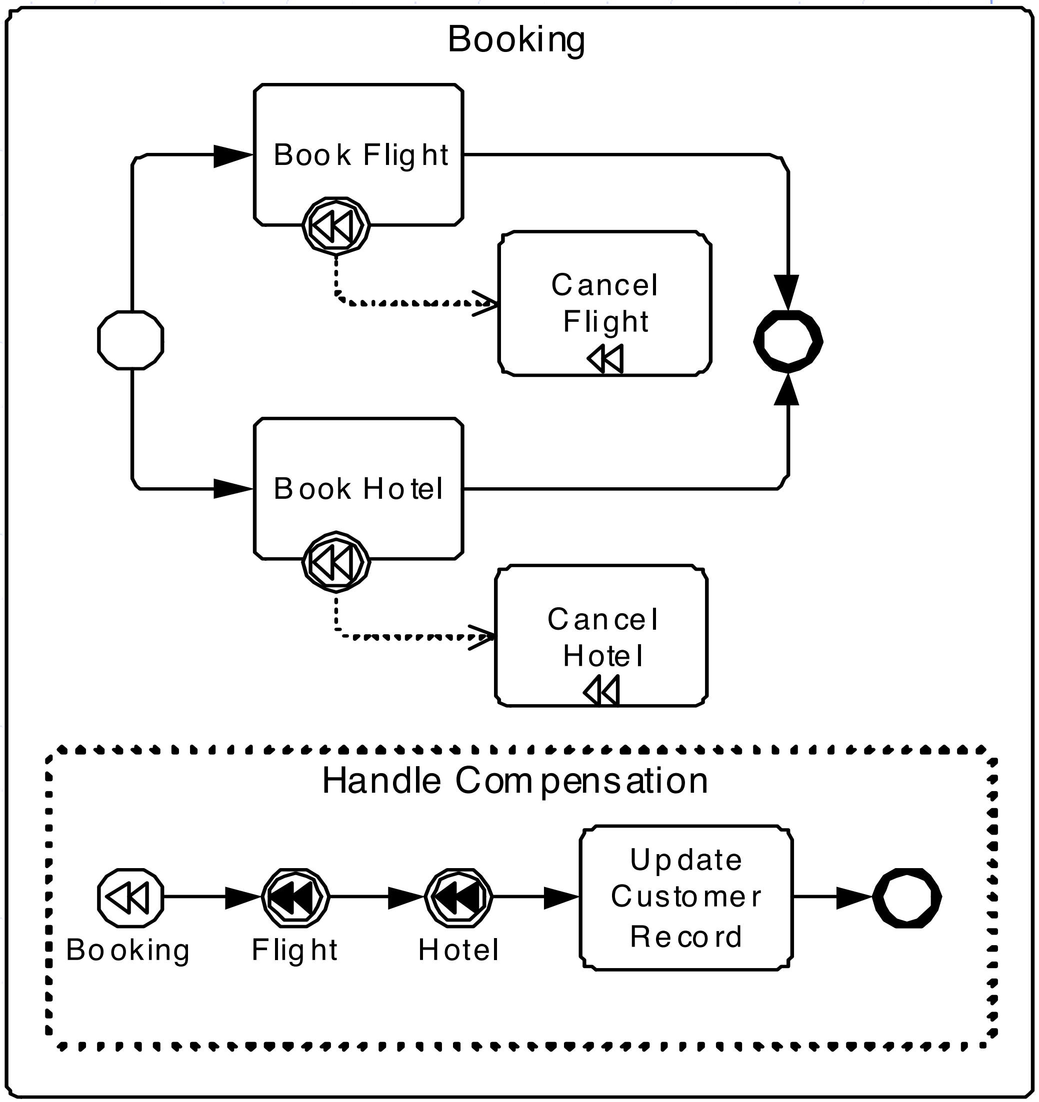
BPMN: Long Running Transactions

- 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

```
<scope name="mainScope">
  <faultHandlers>
    <catchAll>
      <compensateScope target="invoiceSubmissionScope" />
    </catchAll>
  </faultHandlers>
  <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 -->
  </sequence>
</scope>
```

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

JOLIE: programming orchestrators with a C / Java like syntax

```
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
   }
}
```

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
{
    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:

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

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

Location

Protocol

Interface

```
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***

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

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

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

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

Location

Protocol

Interface

JOLIE: work- and control-flow

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

sequence:
```
send@S( x ) ; receive( msg )
```

parallel:
```
send@S( x ) | receive( msg )
```

choice:
```
[ recv1( x ) ] { … }
[ recv2( x ) ] { … }
```

… as well as the usual control flow constructs

if then else:
```
if ( x > 1 ) { … } else { … }
```

for:
```
for( i = 0, i < n, i++ ) { … }
```

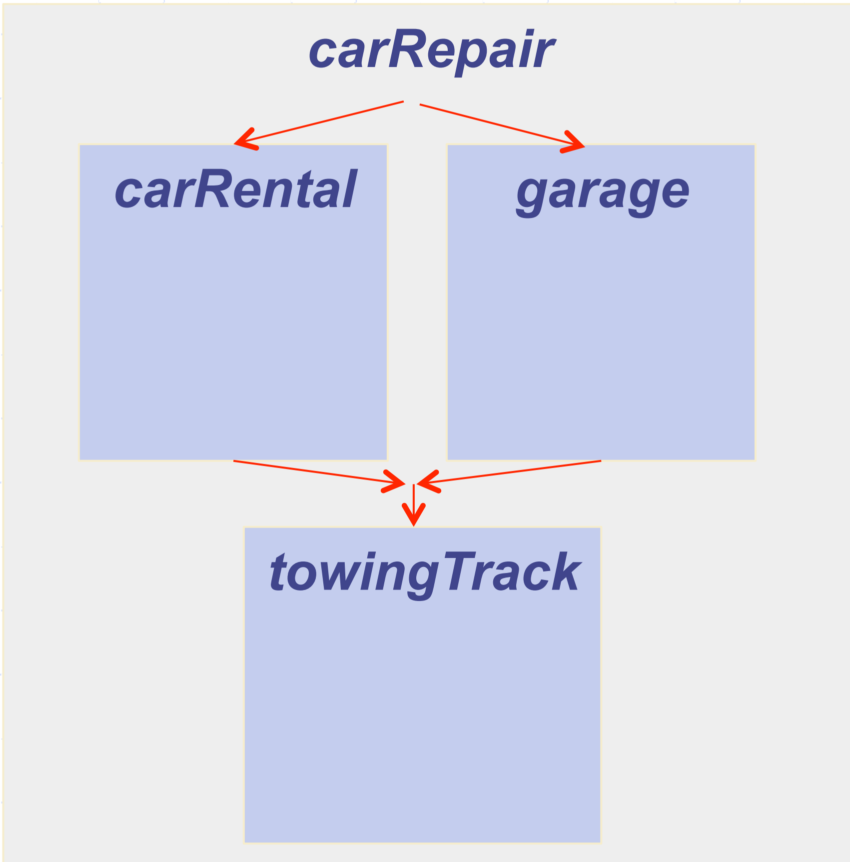
while:
```
while( i < 0 ) { … }
```

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**

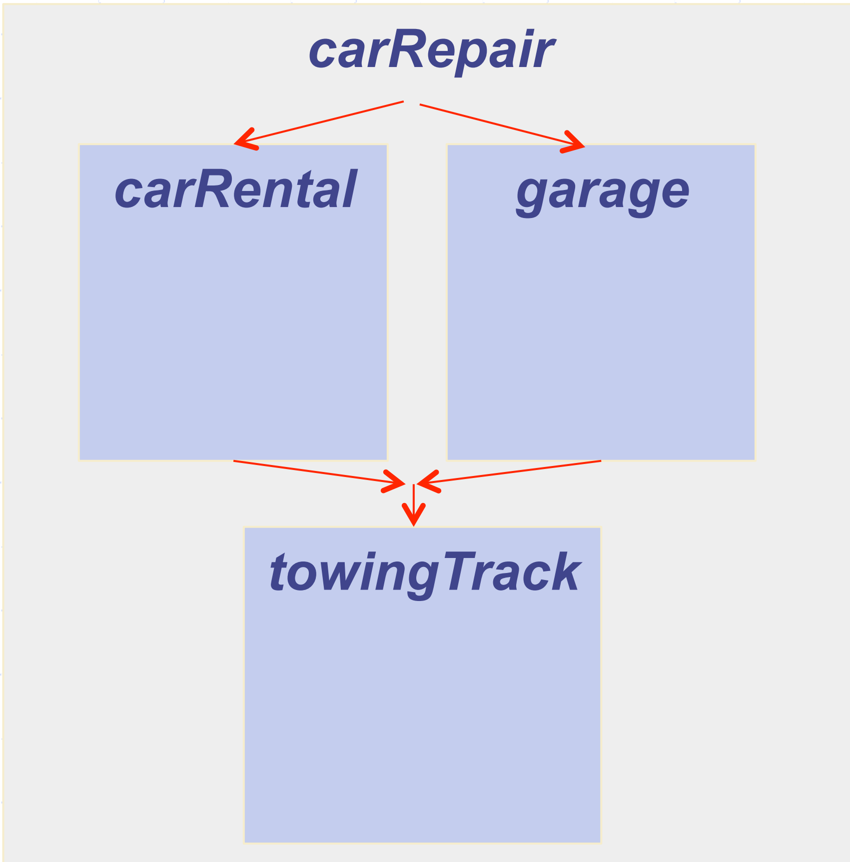
Statically Defined Hierarchy of Scopes

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



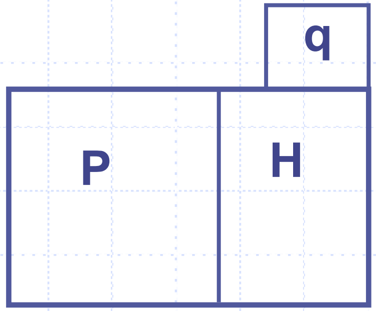
Fault handling

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



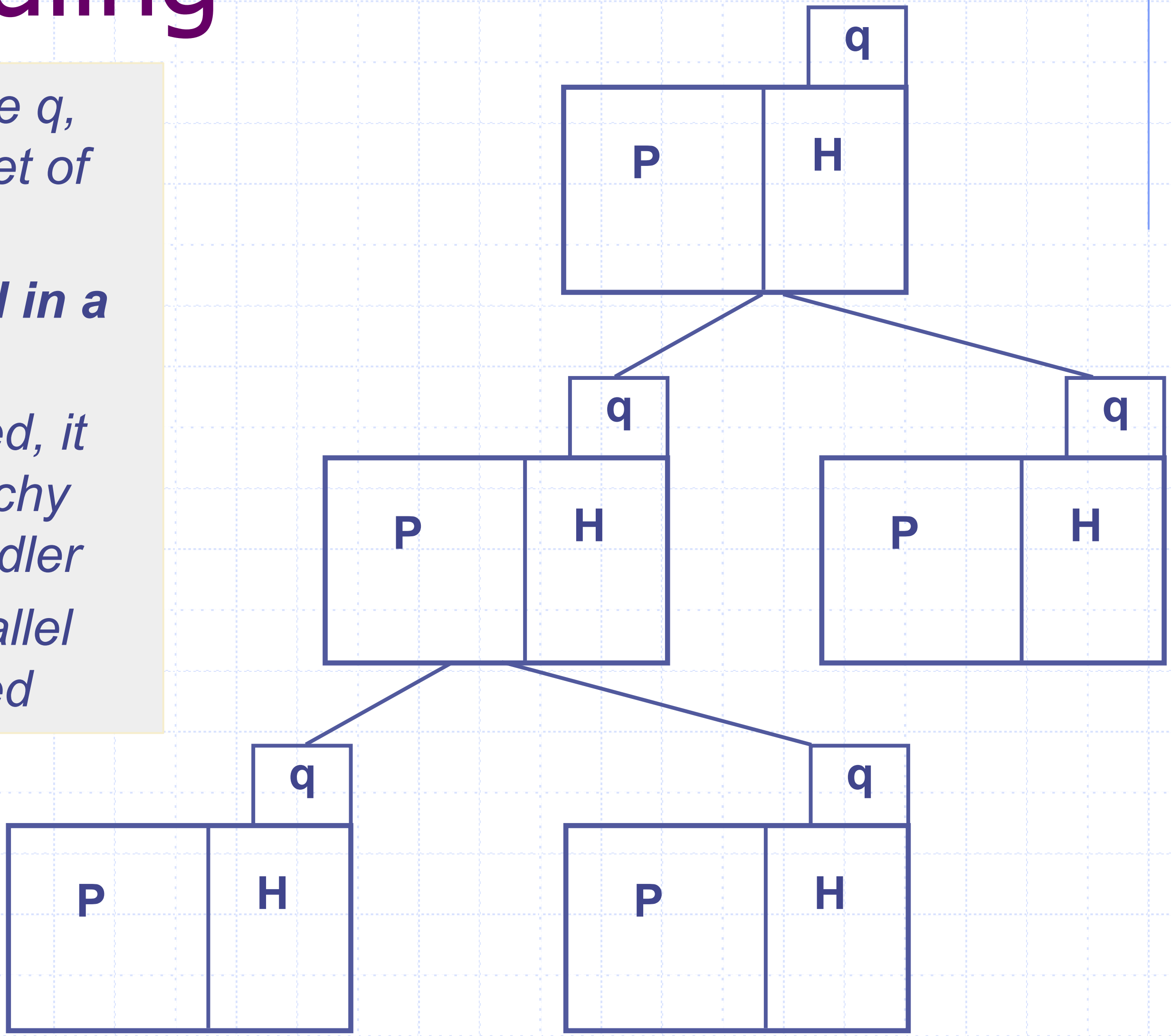
Fault handling

- ***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*



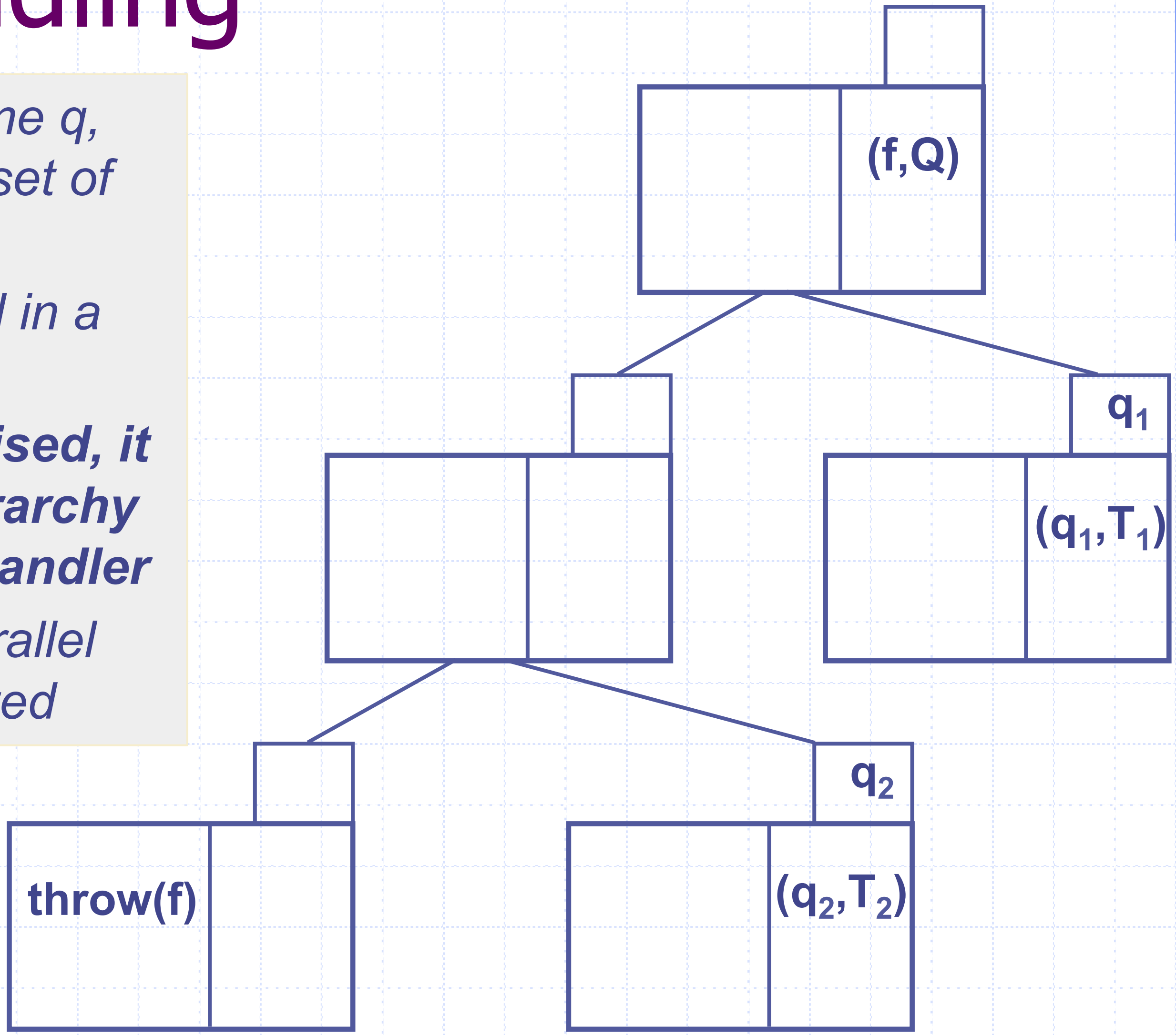
Fault handling

- *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*



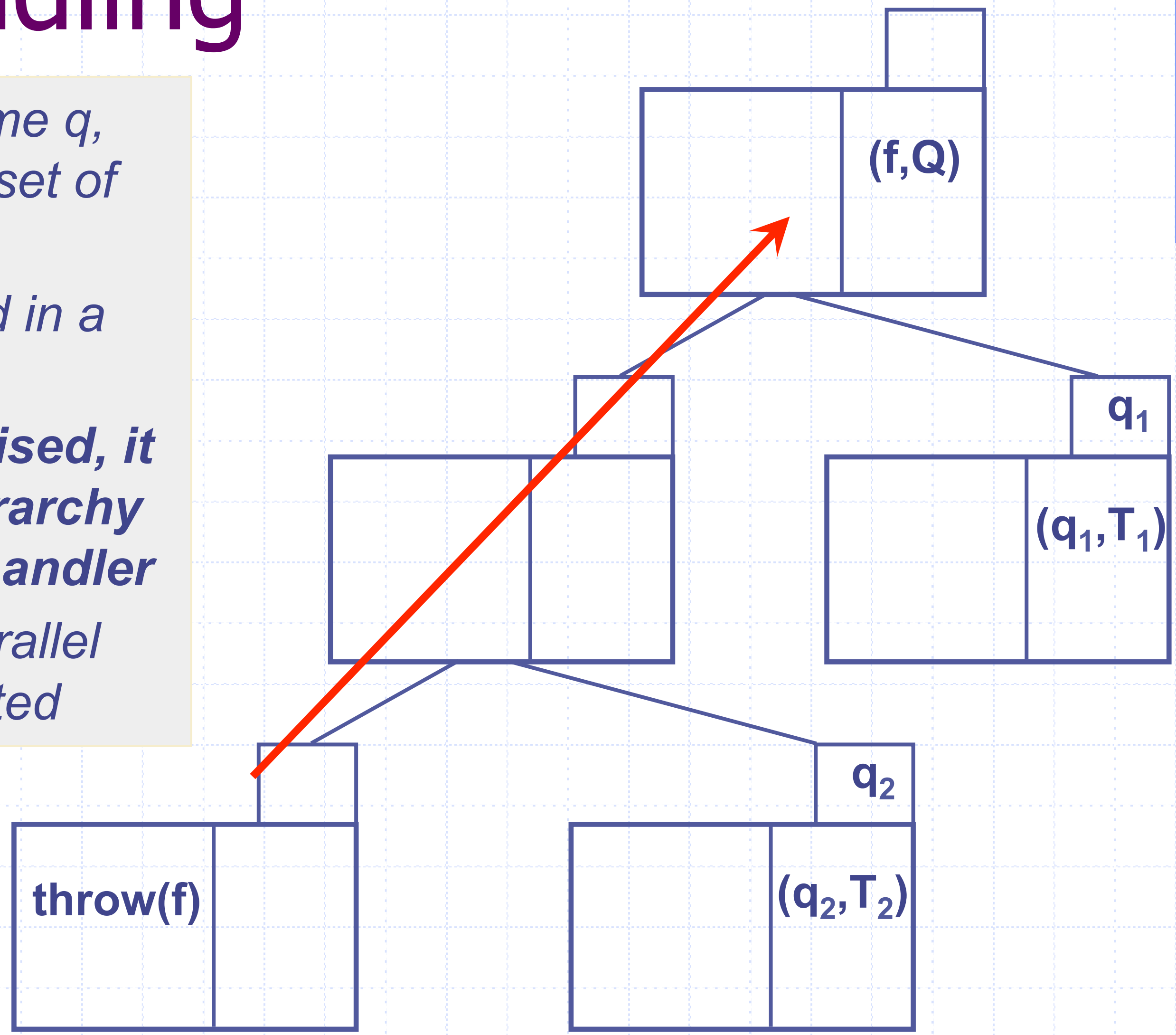
Fault handling

- *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*



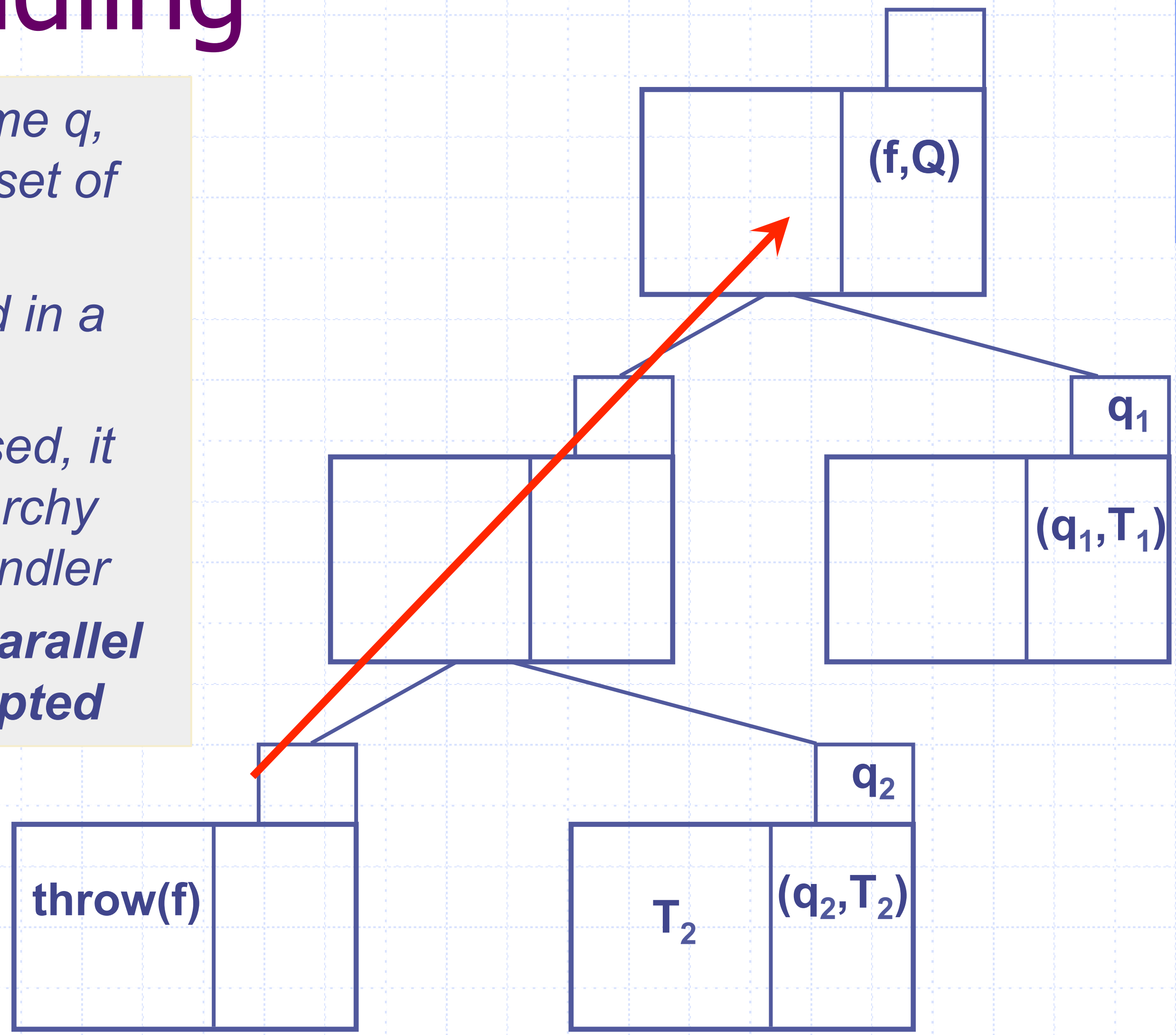
Fault handling

- *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*



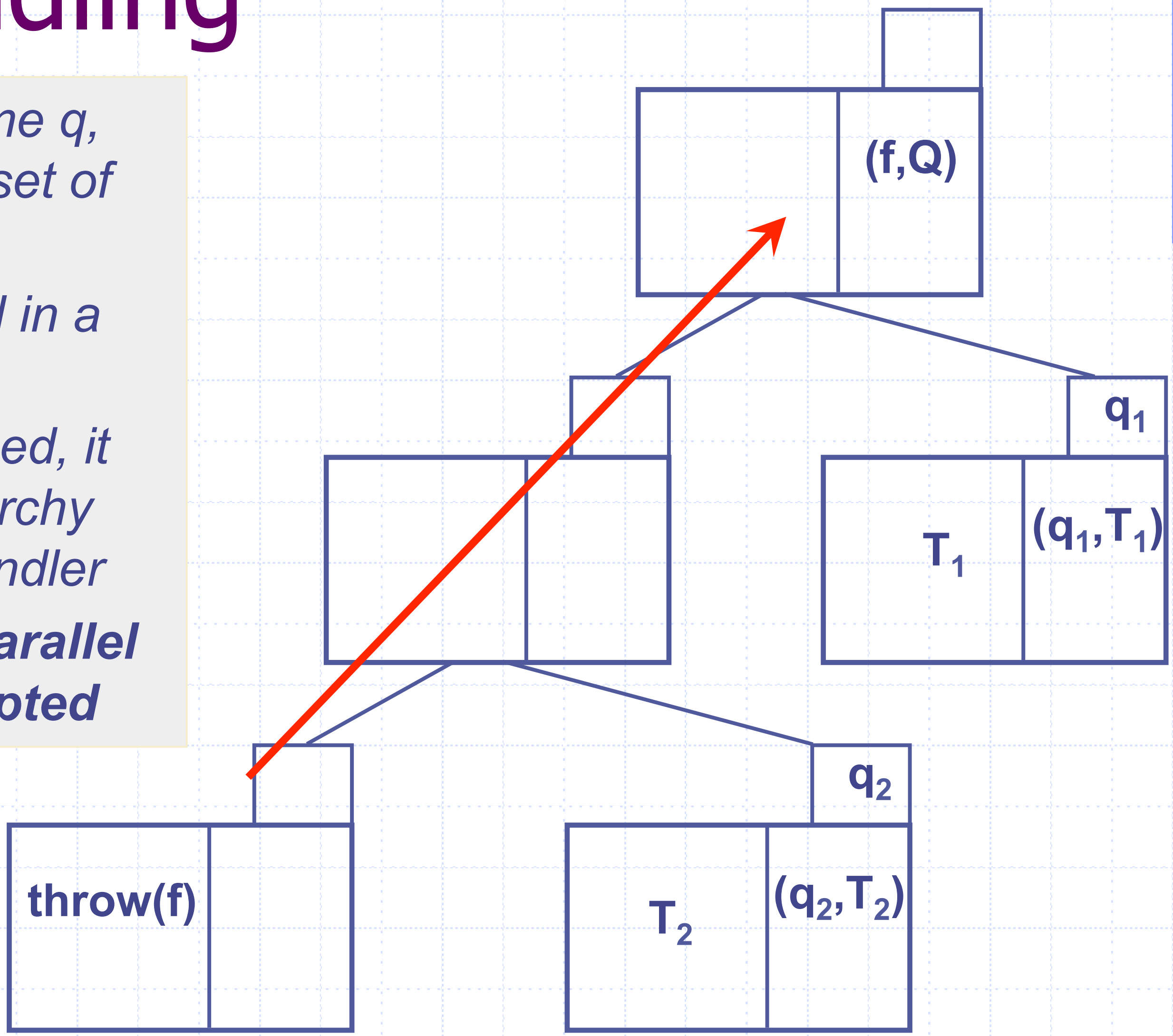
Fault handling

- *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***



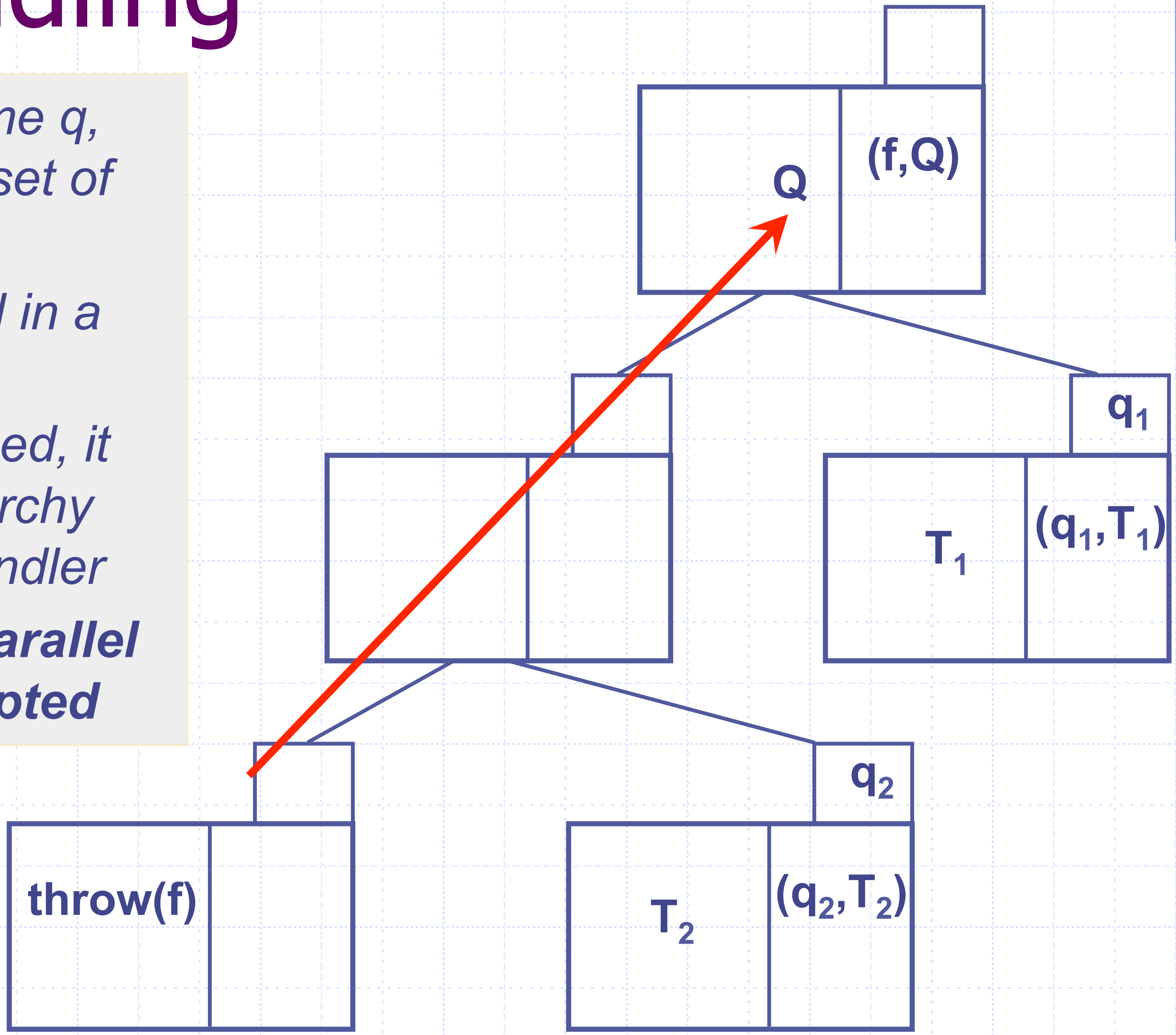
Fault handling

- *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

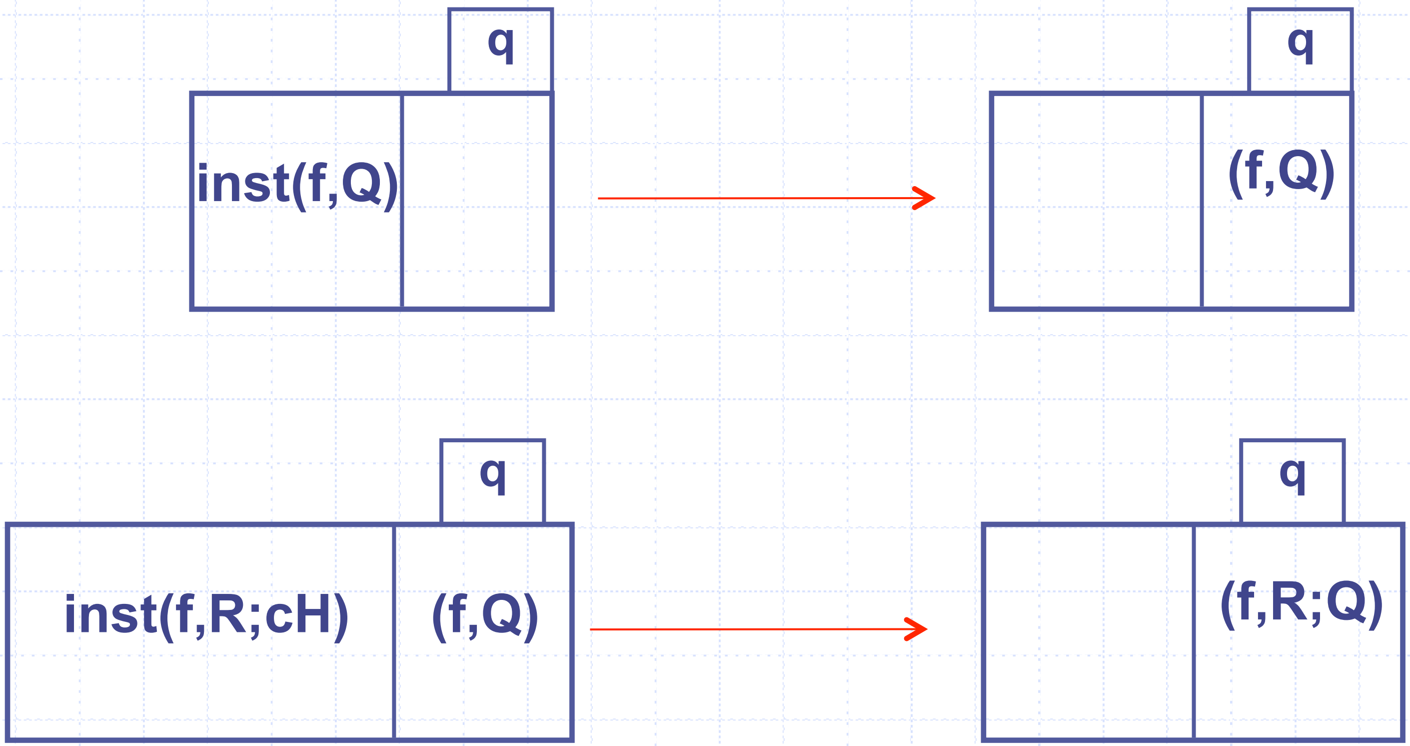
- *Scopes have a name q, an activity P, and a set of fault handlers H*
- *They are organized in a hierarchy*
- *When a fault is raised, it goes up in the hierarchy until it reaches a handler*
- ***While going up, parallel scopes are interrupted***



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



Example

- Consider:

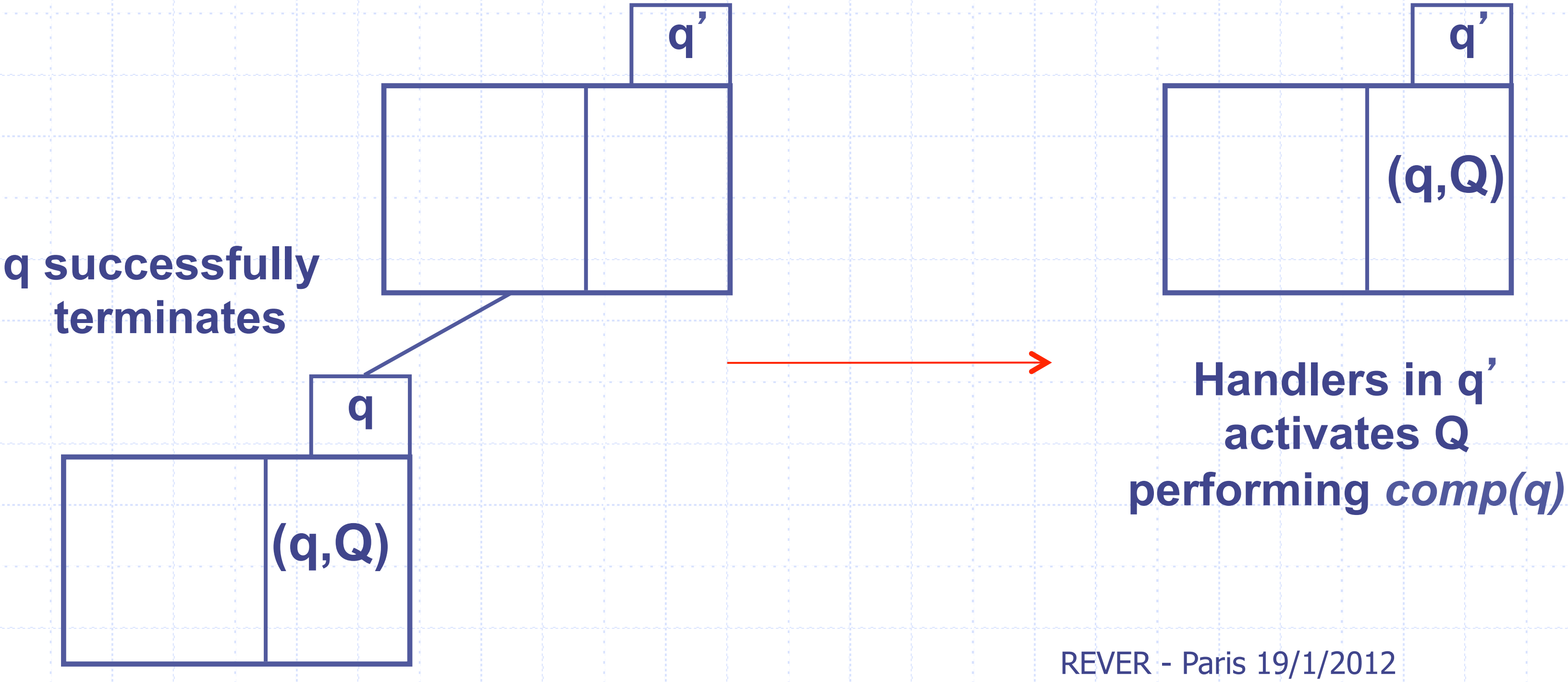
```
{throw(f) |
 while (i <100) if i%2=0 then P else Q, H}_q
```

- 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

```
{ throw(f) |
  while (i <100) if i%2=0
         then P;inst(cH;P′)
         else Q;inst(cH;Q′), H}_q
```

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

$$\{\, inst([fT \mapsto Bus; inst([q \mapsto cH; revBus])]);\\ Hotel; inst([q \mapsto revHotel]);\\ Train; inst([q \mapsto cH; revTrain])\}_q$$

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?...