Monitoring and Recovery of Semantic Web Services

Roman Vaculin

Institute of Computer Science
Academy of Sciences of the Czech Republic

Outline

- Introduction (Motivation + Web Services + OWL-S)
- Semantic Monitoring
  - Process model example and example problems
  - Primitive (semantic) events & Event types derived from OWL-S
  - Composite events & Event detection algebra
  - Semantic filtering
  - Event detection
- Exception handling and recovery
  - Types of failures and responses to them
  - Event types and event expressions
  - Fault handling:
    - *fault handlers*
    - *constraint violation handlers*
    - *event handlers*
    - *compensation constructs*
- Conclusions & References

Motivation

A software agent executing a Web Service must be able to

1. Understand the semantics of the Web Service
2. Interpret the course and the results of the execution (and respond to it)
3. Deal with erroneous states
4. Understand and interpret the sources of problems
   - so that it can for example recover or avoid the situation next time if possible

Semantic Monitoring & Recovery as an extension of Semantic Web Services offer an answer.

Introduction

- Importance of powerful monitoring, exception handling and recovery techniques increases:
  - operating environments of WS become more dynamic
  - WS systems are expected to work in an autonomous or semi-autonomous fashion
  - workflows are expected to be adaptive
- Little effort invested into monitoring & recovery of Sem. Web Services (SWS)
  - neither OWL-S nor WSMO provide support for exception handling and recovery
  - recovery of SWS in combination with monitoring and traditional SWS techniques for discovery and composition have great potential for increasing the autonomy of web services systems
- WS standards for workflow (WS-BPEL) and transactions (WS-Transaction, Business Transaction Protocol) solve the recovery problem partially
  - by providing support for some form of long running transactions (LRT)
  - no support for monitoring and user/designer defined exceptional states

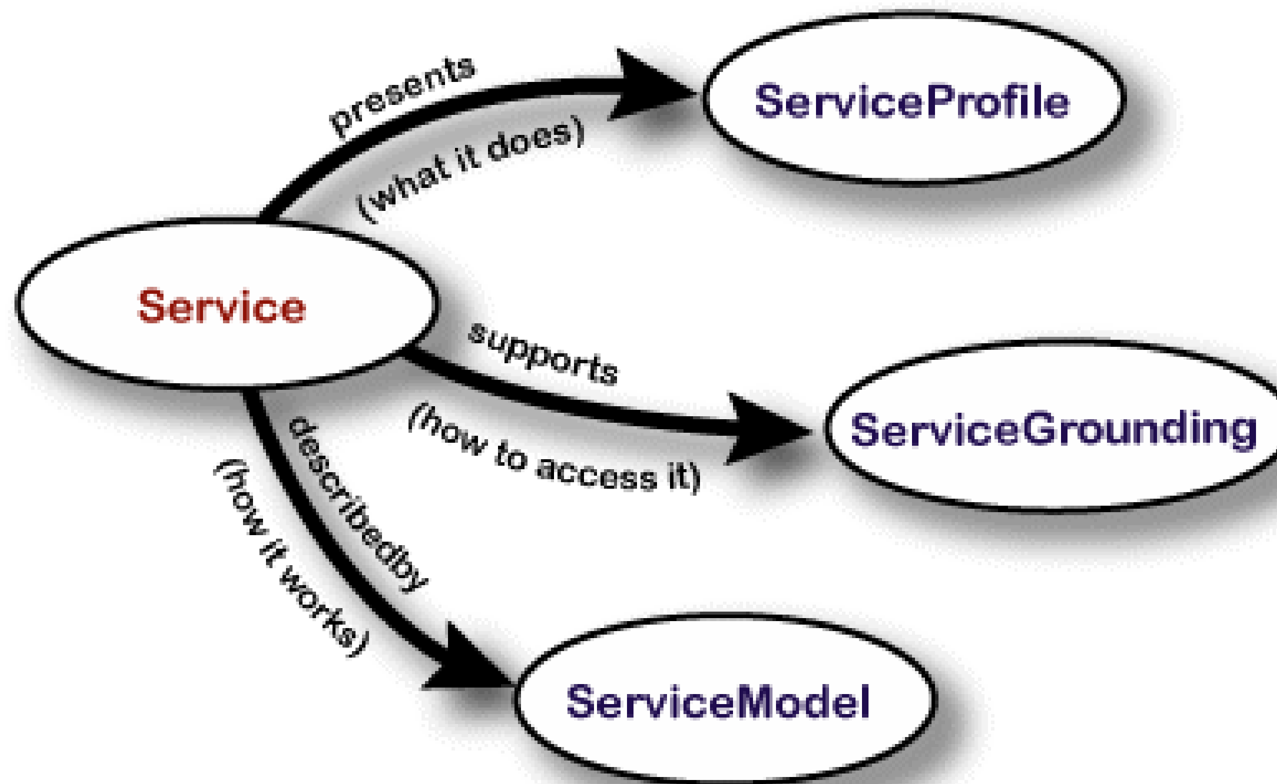
Web Services

- Defined by the W3C as :
  - a software system designed to support interoperable Machine to Machine interaction over a network
- Another definition:
  - A task-oriented,
  - XML-based business application
  - Network-accessible via an API (programmable) from anywhere (location transparent)
- Semantic Web services:
  - Basic idea:
    - *Use semantic languages for representation of WS capabilities and interaction with it*
  - Why?
    - *Automated understanding of data and service semantics (search)*
    - *Automated methods of reconciling syntactic and semantic data heterogeneity (interoperability)*
    - *Reasoning on, and processing of retrieved and understood data and services to accomplish individual goal(s) (automated composition)*

OWL-S introduction

- Semantic Web Services Description language
  - A language based on W3C recommendations
  - Expressed in OWL
- Addresses web services
  - Capability-based search and discovery
  - Interaction specifications
  - Execution
- Defines three components:
  - **Profile**: describes *what* the service does
  - **Process model**: *how* to interact with the service
  - **Grounding**: links the process model to the specific implementation infrastructure

OWL-S model



Service Profile

Service Profile
&process;#Parameter
&expr;#Condition
&process;#Result
&process;#Input
&process;#Output
Profile
&xsd;#anyURL
&xsd;#anyURL
Process
ServiceParameter
Thing
ServiceCategory
hasParameter
hasPrecondition
hasResult
hasInput
hasOutput
<rdfs:subPropertyOf "#hasParameter" />
serviceClassification
serviceProduct
serviceName
textDescription
has_process
serviceParameter
serviceCategory
serviceParameterName
sParameter
categoryName
taxonomy
code
value
ObjectProperty
DatatypeProperty
SubClass/Property

Service Process Model

&expr;#Condition
Result
Parameter
Perform
Participant
xsd:string
hasPrecondition
hasResult
hasParameter
process
hasParticipant
name
Process
Input
Output
Local
hasInput
hasOutput
hasLocal
disjointWith
Simple Process
Atomic Process
CompositeProcess
realizedBy
realizes
expandsTo
collapsesTo
disjointWith
invokable
composedOf
xsd:boolean
Control Construct
ObjectProperty
DatatypeProperty
SubClass/Property
components
Sequence
Split
Split-Join
Any-Order
Choice
unionOf

Introduction to monitoring

- Monitoring mechanisms can be used
  - during the execution to support a dynamic response to the given execution course
    - *fault / event handling*
    - *execution recovery*
    - *dynamic adaptation*
  - to support measuring and evaluation of Quality of Services (QoS)
  - after the execution is finished for analysis and auditing
    - *applications in areas such as (Semantic) Business Process Management and Process Mining*
- Typically, **primitive** and **composite events** are distinguished:
  - **Primitive events**: individual events emitted directly by various components of the systems or external events that can be detected directly
  - **Composite events**: complex events patterns consisting of several primitive events

Traditional vs. Semantic Monitoring

- In most works on event monitoring:
  - emitted events typically characterized by an **event type**
  - events derived from the system implementation and represented on the syntactic level
  - no declarative specification of event types and their parameters
- On the contrary, SWS frameworks
  - provide means for explicit specification of WS capabilities, interfaces and interaction protocols
    - *e.g., Inputs, Outputs, Preconditions, Effects, Classification, etc.*
    - *done by annotating WS by concepts with a clear semantics defined in ontologies*
  - semantic descriptions can be also used for describing event types and event instances
    - *also data associated with events can be annotated by ontology concepts*
- Advantages of Semantic Monitoring
  - easy processing and sharing by SW agents and applications (clear semantics)
  - more flexible event detection employing semantic reasoning
  - advanced analysis after the execution:
    - *complex filtering and querying techniques exploiting the rich semantic interaction trace*

Process model example

Shopping Service Process Model

repeat

repeat

Login
I: username
- xsd:string
password
- xsd:string
O: status
- xsd:boolean

LookupItem
I: category
- ItemCategory
name
- ItemName
O: item
- ItemDetails

AddItemToCart
I: id
- ItemID
O: id
- CartID

BuyAndShipItems
I: id - CartID
country - Country
city - City
street - Street
O: confirmation
- ConfirmationID

Logout
I: username
- xsd:string
O: status
- xsd:boolean

Domain Ontologies

ItemName
ItemCategory
GenericID
ItemDetails
ItemID
CartID
ConfirmatioID
Country
Street
City

Examples monitoring problems

1. **Event patterns using primitive events only:**
   a) Detect every call of a given operation (e.g., *Logout*).
   b) Detect when a particular result is produced, e.g., *Login* fails since the *username* cannot be verified.
   c) Filter service calls with a given parameter type, e.g., *LookupItem* calls with the *category* parameter that is an instance of *Book* class.
2. **Complex event patterns:**
   a) Detect repeated occurrence of some event within a certain time, e.g., 3 unsuccessful *Login* calls within 2 minutes.
   b) Detect situations when the customer logs out without buying anything.
   c) Detect service calls taking longer than a specified time (as a result a QoS metric might be updated)
3. **Off-line post-execution analysis:**
   a) Identify US customers shopping for *Books* that spent more than $1000 within 3 days.
   b) Analyze popularity of some workflow (specified by some pattern and its features).
   c) Analyze efficiency of a workflow (e.g., time to buy) or it effectiveness (i.e., if a given sequence of calls leads to purchasing a product).

Primitive (semantic) events

- **Primitive event occurrence** is an instantaneous, atomic occurrence of an interest at a point in time
- Primitive event occurrences are directly emitted by the system or its components
  - or is detected as an external event
- Each primitive event occurrence is an instance of some **event type** and possibly has additional information in the form of parameters
- We define
  - **primitive event types as concepts in an ontology** and
  - **occurrences of primitive events as instances** of ontology concepts
- OWL is used as an ontology language
  - details of an event can be specified by referring to relevant parts of the executed process model
  - convenient for OWL-S aware applications since they can easily interpret the content of events

Event types derived from OWL-S

Event	
timeStamp [1]	xsd:dateTime
process [1]	process:Process

ProcessCallEvent	
input [0..n]	ParameterValueBinding
effect [0..n]	expr:Expression
output [0..n]	ParameterValueBinding

ControlConstructEvent

ResultEvalEvent

GroundingEvent

PreconditionEvalEvent	
truthValue [1]	xsd:boolean
condition [1]	expr:Condition

AssignEvent	
dataValue [0..1]	any XSD primitive type
objectValue [0..1]	owl:Thing
toParameter [1]	process:Parameter

ParameterValueBinding	
dataValue [0..1]	any XSD primitive type
objectValue [0..1]	owl:Thing
toParameter [0..1]	process:Parameter

ExceptionEvent	
textMessage [0..1]	xsd:string

process:Process

expr:Expression

expr:Condition

process:Parameter

isa

process

effect*

input*

output*

condition

toParameter

process:hasPrecondition*

process:hasParameter*

Primitive event example

```
<AtomicProcessEndEvent>
      <timestamp>2007-03-12T12:35:12</timestamp>
      <process rdf:resource="&shopService;Login">
      <input>
       <ParameterValueBinding>
           <toParameter rdf:resource="&shopService;username"/>
           <dataValue>john</dataValue>
       </ParameterValueBinding>
      </input>
      <input>
      <ParameterValueBinding>
           <toParameter rdf:resource="&shopService;password"/>
           <dataValue>foo</dataValue>
       </ParameterValueBinding>
      </input>
      <output>
       <ParameterValueBinding>
           <toParameter rdf:resource="&shopService;status"/>
           <dataValue>true</dataValue>
       </ParameterValueBinding>
      </output>
</AtomicProcessCallEvent>
```

Event detection algebra

- Event detection algebras present a mechanism for composite events specification and detection
- In event algebras,
  - *primitive event types* and
  - a number of *operators*
  - are used to form **event expressions**
  - that represent an event pattern of interest
- In our case, primitive event types correspond to event types defined in the events ontology
- During the system execution primitive event occurrences are emitted
  - event occurrences form **event streams** which are used to define the semantics of the event algebra and for the purposes of the event detection
  - e.g., a **primitive event stream** is a set of primitive event occurrences of the same event type with different times

Algebra operators

- Composite events are defined by **event expressions** built from primitive event types and **algebra operators**:

Operator	Explanation
$A \wedge B$	Conjunction. Occurs when both A and B occur irrespective of their order.
$A \vee B$	Disjunction. Occurs when A or B occurs.
$A; B$	Sequence. Occurs when A occurs before B.
$A - B$	Negation. Occurs when there is an occurrence of A during which there is no occurrence of B.
A_t	Temporal restriction. Occurs when there is an occurrence of A shorter then t time units.
$A + t$	Temporal event. A temporal event is a special type of a primitive event that occurs t time units after an occurrence of A. A temporal event occurrence refers to the event occurrence of A that initiated it.

Semantic filtering

- To allow filtering of detected events based on their content we extended event expressions with **semantic filters**
- **Semantic filter** is an expressions in the form of conjunction of description logics atoms enriched with OWL datatypes and SWRL built-ins
  - Syntax is motivated by SWRL expressions that are used in SWRL rules antecedents
- Filter expressions allow us to match events represented as OWL instances
- We assume existence of a knowledge base KB that is used for evaluation of filter expressions
  - An execution engine (the OWL-S Virtual Machine) maintains the KB during execution of the process model and stores produced results in the KB

Filter expression

- A **filter expression** is a conjunction of **atoms**.
- An **atom** can be one of the following expressions:
  - ***C(s)*** (concept atom),
  - ***Po(s,t)*** (object property atom),
  - ***Pd(s,d)*** (datatype property atom),
  - ***sameAs(s,t)*** (same as atom),
  - ***differentFrom(s,t)*** (different from atom)
  - ***builtinID(d1,…,dn)*** (built-in atom),
  - where ***C*** is an OWL class name (primitive event type), ***Po*** is an OWL object property, ***Pd*** is an OWL datatype property, ***builtinID*** is an identifier of some SWRL built-in predicate with arity n, **d1,…,dn** are variables or OWL data values, **s** and **t** are variables or OWL individuals in the KB and **d** is a variable or an OWL data value.
- A **filter expression holds** with respect to the KB, if there exists an assignment of individuals (from the KB) and data values to all variables in the expression, such that all atoms hold.

Restricted event types

- In general, we allow every event expression to be associated with a filter expression
  - However, to maintain control over event detection we also introduce a restricted form of event expressions in which filter expressions can be associated with primitive event types only.
- **Restricted event type** is defined as follows:

  T= ?v : A[F]

  where ***T*** is a name of the defined event type, ***A*** is a primitive event type, ***?v*** is a variable and ***F*** is a filter expression.
- **Example (restricted type)**: The following expression defines a new restricted event type *Logout* derived from the *AtomicProcessEndEvent* event type:

  Logout = ?x : AtomicProcessEndEvent [
  process(?x,?process) &
  sameAs(?process,"&shopService;Logout")]

Restricted and extended event expressions

- **Restricted event expressions** are event expressions in which defined restricted types can be used in the same fashion as primitive types.
- **Example (restricted expression)**: *((Login; Logout) – Buy)*

- **Extended event expressions** are event expressions in which filter expression can be attached to any event subexpression.
  - If ***A*** is a valid event expression, also ***A[F]*** and ***?v: A[F]*** are valid expressions.
  - ***F*** stands for a filter expression and ***?v*** is a variable identifying an event occurrence which was detected as an instance of ***A***.
  - In extended event expressions, restricted event types can be used as well.
- **Example (extended expression):**

(?log1 : Login; ?log2 : Login)$_{120}$ [
input(?log1, ?par1) & toParameter(?par1, ?par1Name) &
sameAs(?par1Name, "&shopService;username") & dataValue(?par1, ?userName1) &
input(?log2, ?par2) & toParameter(?par2, ?par2Name) &
sameAs(?par2Name, "&shopService;username") & dataValue(?par2, ?userName1)]

Events detection

Detection process combines three techniques:

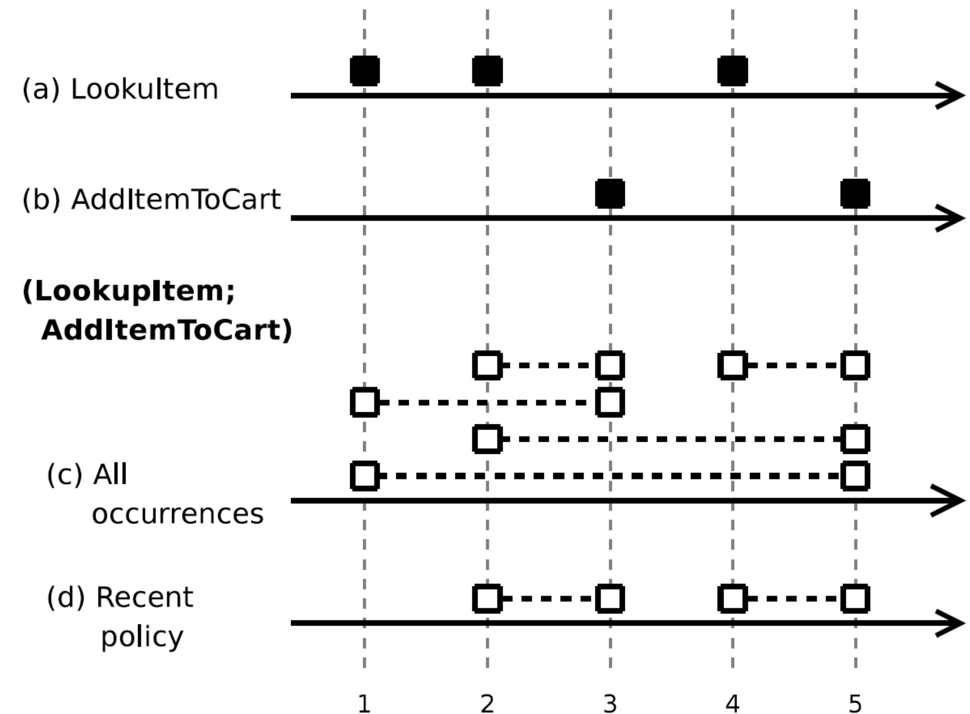
1. **Detection of primitive types** represented as OWL instances is based on instance checking (i.e., deciding if an event occurrence is of a given type)

2. **Detection of composite events** is based on event detection trees

3. **Semantic filters evaluation**:
   - after an event is detected for a given event expression, the attached filter must be evaluated
     - *filter expression is internally translated into a SPARQL query which is than evaluated against the current KB state*

Composite events detection

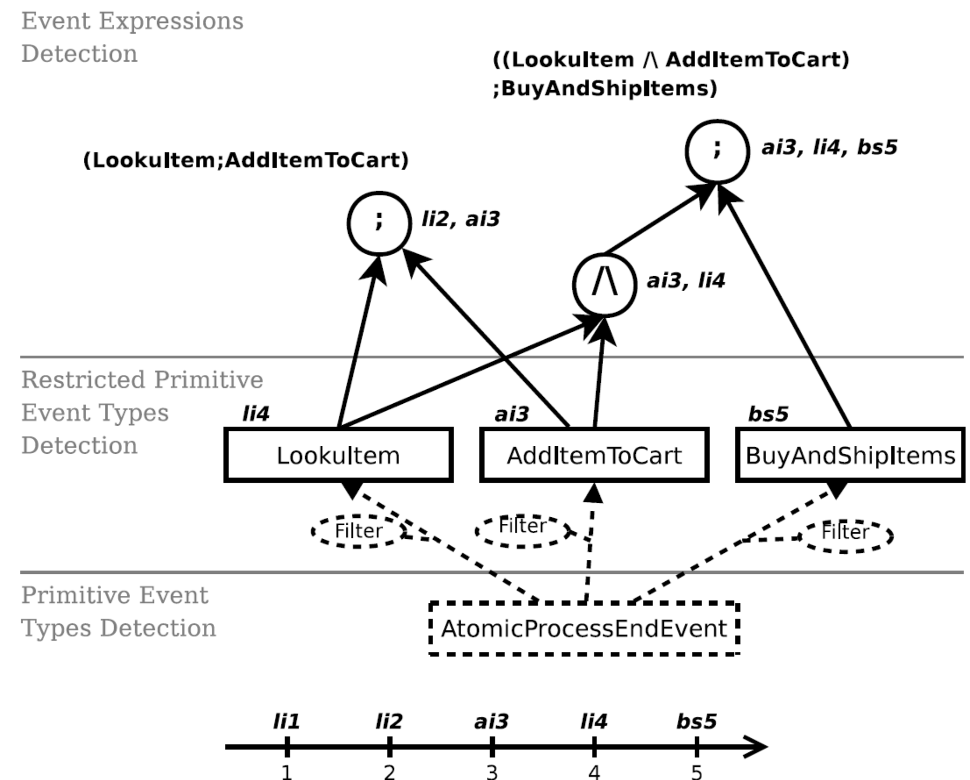
- Composite event is caused by occurrences of primitive events
- Since events can occur repeatedly, several combinations of primitive events can trigger a composite event at a time.
- **Restriction policies** can limit the number of detected events
- **Recent policy:** If there are more candidates, the one with the maximal start time is detected
  - guarantees that if there is one or more event occurrences of a composite event, one of them will be always detected



Event detection trees

- Present an efficient mechanism for detection of composite events defined by event expressions
- Each event expression represented as a tree
  - leaves represent event types occurring in the expression and
  - every other node represents one composition operator
- Detection starts at the bottom with primitive events and proceeds in the bottom up direction by progressively detecting occurrences of more complex subexpression
- Whenever a new event occurrence is detected by a node, the node notifies its parent(s)
- Every node maintains a history of event occurrences in its own buffer

Event detection trees example for 2 expressions; snapshot at time 5



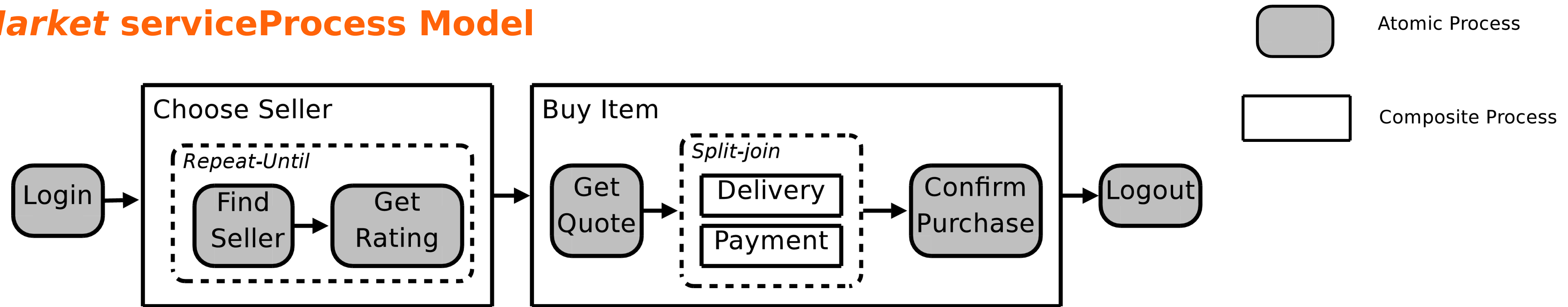
Exception Handling and Recovery

Approach

- To provide means for specification of reliable, adaptive process models, following is needed:
  - standard fault handling
  - mechanisms allowing a designer to define what situations are supposed to trigger an erroneous state
    - *e.g., dealing with SLAs and other constraints*
  - compensation mechanisms
  - powerful monitoring
- Solution: following extensions to the OWL-S specification
  - explicit fault and event handlers
  - *constraint-violation handlers* for associating constraint violation conditions with appropriate recovery actions
  - explicit recovery actions
  - *compensation actions* and *blocks* for undoing effects of the partial work after a fault has occurred
  - semantic monitoring techniques for detection of erroneous states

Another Process Model Example

eMarket serviceProcess Model



1. **Choose Seller** finds a seller with good rating
2. **Buy Item** gets the *price quote* and continues with **Delivery** and **Payment** in parallel and purchase confirmation

Additional Constraints:

- For **Get Rating** two candidate services available:
  - an unreliable but cheap public service (a preferred choice)
  - a reliable and expensive commercial service (backup in case of failure)
- Quote provided by the **Get Quote** service is valid only for a fixed time
  - If **Buy Item** does not finish within this time, it must be restarted and a new quote must be obtained.
  - If necessary,partial work needs to be compensated as part of terminating and restarting the process.

Execution processing states

During the execution of a process model, every process is in one of the following states:

1. **uninitialized** (process not started yet)
2. **started** (but not finished yet)
3. **finished** (successfully finished)
4. **failed** (a fault has occurred or the process was terminated).

When a failure occurs in a process

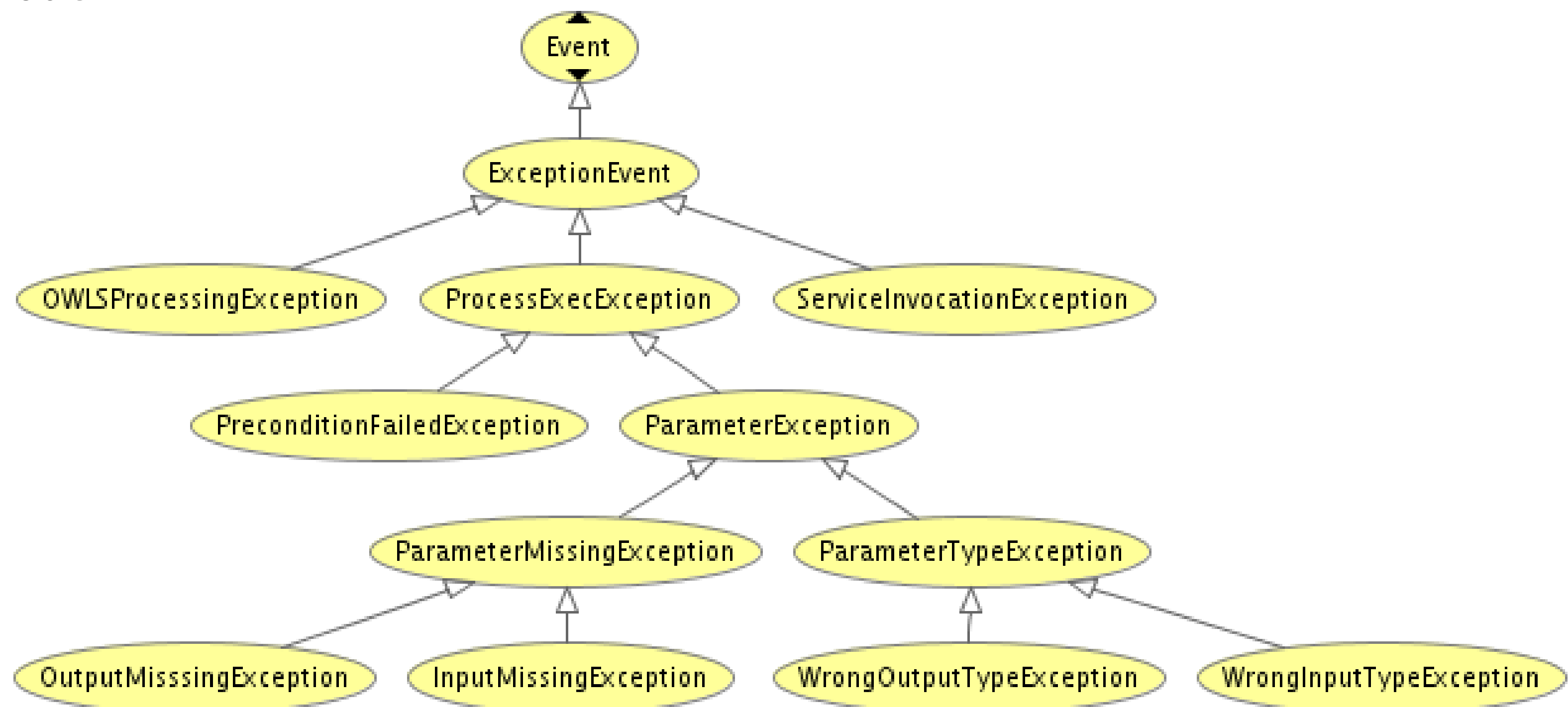
1. its state is switched from **started** to **failed**,
2. all its subprocesses are terminated and
3. the execution control is taken over by fault handling mechanisms.

Categories of erroneous situations

1. **Service invocation errors:**
   - Communication failure, serialization error, no / malformed response, time-out, etc.
2. **OWL-S processing errors:**
   - Parsing/syntax problems with malformed OWL-S files
3. **Process level execution errors:**
   - Erroneous situations caused by discrepancies on the process model level.
     - *e.g., a required input not provided by the client, a wrong input type provided, the precondition of a process fails so that it cannot be executed, etc.*
4. **Application level errors:**
   - Erroneous states specific to the application logic of a web service as, e.g., no *Seller* is found for a given *product.*
5. **Constraint violations:** resulting, e.g., from applicable SLAs
   - Soft and hard constraints are distinguished.
     - ***Hard constraint** violation considered as an **erroneous state** that leads to termination of the process and to triggering fault handling*
     - ***Soft constraint** violation does not trigger an abnormal execution state.*

Exception types

- **Exception types** for OWL-S are defined as concepts in the **exception types taxonomy**
  - extends the **event types taxonomy**
  - **event / exception types are concepts in an OWL ontology** and
  - **events / exceptions as instances** of ontology concepts
  - details of an event can be specified by referring to relevant parts of the executed process model



Exception Event Example

- Exception events instances contain context information specifying reasons for the failure

```
<WrongInputTypeException>
    <timestamp>2008-03-12T12:47:23</timestamp>
    <process rdf:resource="&eMarket;FindSeller"/>
    <parameter rdf:resource="&eMarket;product"/>
    <expectedType>&shopOntology;ProducCategory</expectedType>
    <invocationType>&xsd;string</invocationType>
    <dataValue>Book</dataValue>
</WrongInputTypeException>
```

Supported actions to respond to a failure

- **Neutral actions:** no effect on the state of the failed process
  - e.g., execution of a web service, a logging action, etc.
- **Recovery actions:** restore the state of a failed process to the finished state and return to a normal execution flow.
  - **retry(timeout)** and **retry(N)**
  - **replaceBy(otherProcess)**
- **Fault emitting actions:**
  - **throw(Fault)** and **re-throw**
- **Termination actions:** terminate the whole process model
  - **hardTerminate** terminates all processes without compensation,
  - **softTerminate** executes compensation of finished processes before terminating
- **Compensation actions:** start the compensation of a process.
  - **compensate** for invoking the compensation of the associated process
  - **compensateProcess(process)** to start compensation of a specified process.
- **Adaptation actions:** modify the overall execution flow of the service
  - e.g. **replaceProcessBy(originalProcess, newProcess)** or **skip(processName)**.

Fault handlers

- A process can define a list of **fault handlers** that are used after a fault has occurred to respond to a failure and possibly recover.
- A fault handler has the following form:

  FaultHandler(*FaultType* [faultVariable]) { actions }

- A fault handler is triggered if the occurred failure is instance of the ***FaultType*** (refers to type in the exception ontology)
- Processing rules: similar to programming languages
  - If more handlers are defined for a process, they are processed in the order in which they were declared. Only the first matching handler is triggered.
  - If no handler is matched, the fault is propagated to the parent process.
  - Allowed actions: all standard OWL-S control constructs, processes and all action categories described on previous slides
  - If no recovery action was successfully executed in the handler, the process remains in the ***failed*** state after the fault handler finishes.
  - An optional ***faultVariable*** can be used in the fault handler to access the fault occurrence and its value.

Example Fault Handler

- The following handler retries the `GetRating` service 2 times if it times out

```
AtomicProcess(GetRating) {
  FaultHandler(ServiceTimeoutException) {
     retry(2);
   }
}
```

- We use an abstract syntax instead of the XML serialization of OWL-S

Constraint violation handlers (CV-handlers)

- A process can define a list of **CV-handlers** to detect possible hard constraint violations which are considered as failures.
- A fault handler has the following form:

CV-Handler(*event-expression* [eventVariable]) { actions }

- Triggering a CV-handler results in an immediate termination of the process for which the CV-handler was defined and to changing its state from **started** to **failed**.
- Allowed actions: the same as in fault-handlers.
- Basic processing and detection:
  - A CV-Handler gets triggered when its ***event-expression*** matches an occurrence of a (possibly composite) event.
  - CV-handlers are active in their own process and in all embedded processes.
    - *Motivated by the fact that when a constraint is imposed on some process, it typically applies to all its embedded processes as well.*

Example CV-handler

- The following CV-handler associated with `BuyItem` composite process gets triggered 3 minutes after the process started
  - it first compensates all finished activities
  - and then retries the `BuyItem` service 3 times

```
CompositeProcess(BuyItem) {
   CV-Handler(BuyItemStarted + 3mins) {
      compensate;
      retry(3);
   }
}
```

- The **BuyItemStarted** is a primitive event type derived from the generic **CompositeProcessStartEvent** type defined in the events ontology:

```
 BuyItemStarted = ?x : CompositeProcessStartEvent [
                        sameAs(?x.process,"&e-market;BuyItem")
                      ]
```

Event Handlers

- A process can define a list of **event handlers** to detect possible soft constraint violations which are **NOT** considered as failures.
- A fault handler has the following form:

  EventHandler(*event-expression* [eventVariable]) { actions }

- Allowed actions: with exception of recovery and compensation actions the same as in fault-handlers.
- Basic processing and detection:
  - event handler gets triggered when its ***event-expression*** matches an occurrence of a (possibly composite) event.
  - event handlers are active in their own process and in all embedded processes.
  - triggering an event handler **does not** lead to the process termination.

Compensation Block

- A compensation construct allows a process designer to associate every process with actions that can be used for undoing effects of this process.
- A compensation construct has the following form:

  Compensation { actions }

- Compensation is activated by calling the **compensate** or **compensateProcess** actions,
- The compensation construct of a process is executed only when the process has **finished successfully**.
  - If the process is in a different state (e.g., failed), calling the compensation action for it will have no effect.
- When no compensation construct is defined for a composite process, the default compensation is used:
  - compensating all finished embedded processes in the reverse chronological order of their original invocation

CV-handlers processing

- CV-handlers are processed according to the following strategy:

  (1) Active CV-handlers are considered in the top-down order, starting with the root processes and progressing towards the process that emitted the primitive event;

  (2) if more CV-handlers are defined for one process they are considered in the same order as they were defined;

  (3) for one event occurrence only one CV-handler can get triggered.

Complex recovery example

This example demonstrates complex recovery with embedded fault handlers

```
AtomicProcess(GetRating) {
    FaultHandler(ServiceTimeoutException) {
        retry(2) {
            FaultHandler(ExceptionEvent) {
                replaceBy(CommercialGetRating);
            }
        }
    }
}

AtomicProcess(CommercialGetRating) {
    FaultHandler(ServiceTimeoutException){ retry(5); }
}
```

Conclusions

- Primitive and composite event specification and detection mechanisms suitable for monitoring of semantic web services
- Augmented the event algebra with semantic filters
  - a restricted variant (restricted expressions) suitable for runtime monitoring
- Mechanisms for fault handling and recovery of semantic web services based on OWL-S introduced
  - A novel combination of **event handlers** with **constraint violation handlers** that both rely on and take advantage of **powerful semantic monitoring techniques**
  - Explicit recovery actions play a critical role, since they enable a clear separation of ordinary actions from actions that allow recovery of failed processes and restoration of the normal flow
- Fully implemented in the OWL-S Virtual Machine
  - a generic execution engine for OWL-S process models
  - Preliminary tests show that detection of semantic events is feasible with current tools

Future work

- Efficiency and scalability
  - large process models
  - long running processes
- A strict SWRL based notation is awkward, we are considering a 'dot' based path notations
  - ?log1.input.toParameter instead of
    input(?log1, ?par1) & toParameter(?par1, ?par1Name)
- Introducing generic monitoring ontology layer(s) that would allow monitoring in an system independent fashion
- Semantic-enabled recovery actions, such as
  - ReplaceByEquivalent,
  - Advanced Back & Forward Recovery
  - Automatic Compensation

References

- *Roman Vaculín, Katia Sycara:* ***Specifying and Monitoring Composite Events for Semantic Web Services.*** *In The 5th IEEE European Conference on Web Services. IEEE Computer Society, November 26-28 2007.*
- *Roman Vaculín, Kevin Wiesner, Katia Sycara.* ***Exception handling and recovery of semantic web services****. In the IEEE International Conference on Networks and Services 2008. IEEE Computer Society, March 16-21, 2008.*
- *Kevin Wiesner, Roman Vaculín, Martin Kollingbaum, and Katia Sycara:* ***Recovery Mechanisms for Semantic Web Services****. Accepted for the 8th IFIP International Conference on Distributed Applications and Interoperable Systems, Springer Verlag, 2008.*