Combining ECA Rules with Process Algebras for the Semantic Web

Erik Behrends, Oliver Fritzen, Wolfgang May, Franz Schenk

Institut für Informatik, Universität Göttingen, Germany

{behrends,fritzen,may,schenk}@informatik.uni-goettingen.de

Supported by the EU Network of Excellence

REVERSE®
reasoning on the web

RuleML 2006, Athens, Georgia/USA, Nov. 10, 2006
Motivation and Goals

(Semantic) Web:
- XML: bridge the heterogeneity of data models and languages
- RDF, OWL provide a computer-understandable semantics

... same goals for describing behavior:
- description of behavior in the Semantic Web
- semantic description of behavior

Event-Condition-Action Rules are suitable for both goals:
- operational semantics
- ontology of rules, events, actions
ECA Rules

“On Event check Condition and then do Action”

- paradigm of *Event-Driven Behavior*,
- modular, declarative specification in terms of the domain ontology
- sublanguages for specifying *Events, Conditions, Actions*
- global ECA rules that act “in the Web”

Requirements

- ontology of behavior aspects
- modular markup definition
- implement an operational and executable semantics
Events and Actions in the Semantic Web

- applications do not only have an ontology that describes static notions
  - cities, airlines, flights, etc., relations between them ...
- but also an ontology of events and actions
  - cancelling a flight, cancelling a (hotel, flight) booking,
- Domain languages also describe behavior:

![Diagram]

- Domain Ontology
- Events
- Concepts
- Actions
- Classes
- Relationships
- Individuals

relationships

raise

influence
Embedding of Languages

... there are not only atomic events and actions.

ECA Language:
<event/> <query/> <test/> <action/>

ECA Language embeds Event Language, Query Language, Test Language, and Action Language.

Event Language:
Composite Events

Query Language:
Queries

Test Language:
Conditions

Action Language:
Complex Reactions

Application-Domain Language:
Atomic Events, Literals, Atomic Actions

Active Concepts Ontologies

Domain Ontologies

ECA-CCS 5
Rule Markup: ECA-ML

<!ELEMENT rule (event,query*,test?,action+) >
<eca:rule rule-specific attributes>
  <eca:event identification of the language >
  event specification, probably binding variables
</eca:event>
  <eca:query identification of the language >  <!-- there may be several queries -->
  query specification; using variables, binding others
</eca:query>
  <eca:test identification of the language >
  condition specification, using variables
</eca:test>
  <eca:action identification of the language >  <!-- there may be several actions -->
  action specification, using variables, probably binding local ones
</eca:action>
</eca:rule>
\[ action(X_1, \ldots, X_n) \leftarrow \]
\[ event(X_1, \ldots, X_k), \ query(X_1, \ldots, X_k, \ldots X_n), \ test(X_1, \ldots, X_n) \]
<!ELEMENT rule (event,query*,test?,action+) >
<eca:rule xmlns:travel="http://www.travel.de">
  <eca:event xmlns:snoop="http://www.snoop.org">
    <snoop:seq>
      <travel:delayed-flight flight="{$flight}"/>
      <travel:canceled-flight flight="{$flight}"/></snoop:seq>
  </eca:event>
  <eca:query>
    <eca:variable name="email">
      <eca:opaque lang="http://www.w3.org/xpath">
        doc("http://xml.lufthansa.de")/flights[code="{$flight}"]/passenger/@e-mail
      </eca:opaque>
    </eca:variable>
  </eca:query>
  <eca:action xmlns:smtp="...">
    <smtp:send-mail to="$email" text="...">
  </eca:action>
</eca:rule>
Active Concepts Ontologies

- Domains specify atomic events, actions and static concepts

Composite [Algebraic] Active Concepts

- Event algebras: composite events
- Process algebras (e.g. CCS)
- Consist of *composers/operators* to define composite events/processes,
- Leaves of the terms are atomic domain-level events/actions,
- As operator trees: “standard” XML markup of terms
- RDF markup as languages,
- Every expression can be associated with its language.
Composite Actions: Process Algebras

e.g., CCS - Calculus of Communicating Systems [Milner‘80]

- operational semantics defined by transition rules, e.g.
  - a sequence of actions to be executed,
  - a process that includes “receiving” actions,
  - guarded (i.e., conditional) execution alternatives,
  - the start of a fixpoint (i.e., iteration or even infinite processes), and
  - a family of communicating, concurrent processes.

- Originally only over atomic processes/actions
- reading and writing simulated by communication
  - $a$ (send), $\bar{a}$ (receive) “match” as communication

... extend this to the (Semantic) Web environment with autonomous nodes.
Adaptation of Process Algebras

Goal: specification of reactions in ECA rules

- liberal asynchronous variant of CCS: go on when possible, waiting and delaying possible
- extend with variable bindings semantics
- input variables come bound to values/URIs
- additional variables can be bound by “communication”
- queries as atomic actions: to be executed, contribute to the variable bindings
- event subexpressions as atomic actions: like waiting for $\bar{a}$ communication

$\Rightarrow$ subexpressions in other kinds of component languages
Languages in the Action Component

Process Algebra Responsibility:
- Process Engine
- Event Detector
- Query Engine
- Domain Broker
- Domain Nodes
- Atomic Events
- Literals
- Atomic Actions
- Composer

Other Responsibilities:
- Event Language
- Query/Condition Language

Diagram Connections:
- Process Engine implements Composer
- Action Component Language, e.g. CCS uses Event Language
- Event Language embeds Atomic Events
- Event Language embeds Literals
- Event Language embeds Atomic Actions
- Event Language embeds Query/Condition Language
CCS Markup

- <ccs:sequence>CCS subexpressions</ccs:sequence>
- <ccs:alternative>CCS subexpressions</ccs:alternative>
- <ccs:concurrent>CCS subexpressions</ccs:concurrent>

- <ccs:fixpoint variables="X_1 X_2 \ldots X_n" index="i" // "my" index localvars="...">n subexpressions</ccs:fixpoint>

- <ccs:atomic-action>domain-level action</ccs:atomic-action>
- <ccs:event xmlns:ev-ns="uri">event expression</ccs:event>
- <ccs:query xmlns:q-ns="uri">query expression</ccs:query>
- <ccs:test xmlns:t-ns="uri">test expression</ccs:test>

Embedding Mechanisms: Same as in ECA-ML

- communication by logical variables
- namespaces for identifying languages of subexpressions
Example

Consider the following scenario:

- if a student fails twice in a written exam (composite event), it is required that another oral assessment takes place for deciding upon final passing or failure.

- Action component of the rule: Ask the responsible lecturer for a date and time. If a room is available, the student and the lecturer are notified. If not, ask for another date/time.

```plaintext
fix X . (ask_appointment($Lecturer,$Subj,$StudNo) : 
  ∂ proposed_appointment($Lecturer,$Subj,$DateTime) :
  (available(room,$DateTime) + 
   (¬ available(room,$DateTime) : X))) : 
inform($StudNo,$Subj,$DateTime) : 
inform($Lecturer,$Subj,$DateTime)
```
<eca:rule xmlns:uni="http://www.education.de">
  <eca:event> failed twice – binds $student ID and $course </eca:event>
  <eca:query> binds e-mail addresses of the student and the lecturer </eca:query>
  <eca:action xmlns:ccs="...">
    <ccs:seq
      <ccs:fixpoint variables="X" index="1" localvars="$date $time $room">
        <ccs:seq>
          <ccs:atomic> send asking mail to lecturer </ccs:atomic>
          <ccs:event> answer binds $date and $time </ccs:event>
          <ccs:query> any room $room at $date $time available? </ccs:query>
          <ccs:alt>
            <ccs:test> yes </ccs:test>
            <ccs:seq>
              <ccs:test> no </ccs:test> <ccs:variable name="X"/>
            </ccs:seq>
          </ccs:alt>
        </ccs:seq>
      </ccs:fixpoint>
      <ccs:atomic> send message ($date, $time, $room) to student </ccs:atomic>
      <ccs:atomic> send message ($date, $time, $room) to lecturer </ccs:atomic>
    </ccs:seq>
  </eca:action>
</eca:rule>
Service-Based Architecture

1.1: register rule
eca: travel: match: snoop: ccs: smtp:

1.2: register event
travel: match: snoop:

1.3: atomic event patterns
match: travel:

1.4: register me
travel:

2.1a: atomic events
travel:

2.1b: atomic events
travel:

2.2: atomic events
travel:

2.3: booking
travel:

2.4: detected parameters

3: detected parameters

4: detected parameters

5.1: action
ccs: travel: smtp:

5.2a: atomic actions
travel:

5.2b: atomic actions
smtp:

5.3a: booking
travel:

5.3b: message (here: confirm) by url

5.4: detected parameters

ECA Engine
eca:

Event Detection
snoop:

Action Engine
ccs:

Domain Broker
travel:

Atomic Event Matcher
match:

Lufthansa
travel:

SNCF
travel:

SMTP Mail Service
smtp:

Client C:
Travel Agency
travel:

Language Services Application Domain
Comparison

- CCS (extended with events and queries) strictly more expressive than ECA rules alone:
  ECA pattern in CCS: \textit{event}:\textit{condition}:\textit{action},
- many ECA rules have much simpler actions and do not need CCS,
- useful to have CCS as an \textit{option} for the action part.
**Summary**

- RDF/OWL as integrating semantic model in the Semantic Web
- describe events and actions of an application within its RDF/OWL model
- languages of different expressiveness/complexity available
- ECA rules
  - components
  - application-level atomic events and atomic actions
  - specific languages (event algebras, process algebras)
- Architecture: functionality provided by specialized nodes
Thank You
Questions ??

Further information and publications:
http://dbis.informatik.uni-goettingen.de/eca/
Complementing Slides
example: CCS (Calculus of Communicating Systems, Milner 1980)

describes the execution of processes as a transition system:
(only the asynchronous transitions are listed)

\[
\begin{align*}
\text{fix}_i \vec{X} \vec{P} & \xrightarrow{a} P' \\
\text{fix} \vec{X} \vec{P} & \xrightarrow{a} P'
\end{align*}
\]
Atomic Event Specifications

Sample Event:

```xml
<travel:canceled-flight flight="LH123">
  <travel:reason>bad weather</travel:reason>
</travel:canceled-flight>
```

Event expressions require an auxiliary formalism for specifying relevant events:

- type of event ("travel:canceled-flight"),
- constraints ("must have a travel:reason subelement"),
- extract data from events ("bind @flight to variable flight")

Sample: XML-QL-style matching

```xml
<atomic-event language="match">
  <travel:canceled-flight flight="{$flight}">
    <travel:reason/>
  </travel:canceled-flight>
</atomic-event>
```
Event Expressions: Languages

- EventExpression
  - Atomic Event Description
    - Domain Event
      - Domain Ontology
    - Atomic Event Description Formalism
  - Composite Event Specification
    - EventComposer
      - cardinality
        - EventAlgebra
          - identifier

Rule Model

Ontologies/Languages
<eca:rule xmlns:travel="...">
  <eca:variable name="theSeq">
    <eca:event xmlns:snoop="...">
      <snoop:sequence>
        <snoop:atomic-event language="match">
          <travel:delayed-flight flight="\{\$Flight\}" minutes="\{\$Minutes\}"/>
        </snoop:atomic-event>
        <snoop:atomic-event language="match">
          <travel:canceled-flight flight="\{\$Flight\}"/>
        </snoop:atomic-event>
      </snoop:sequence>
    </eca:event>
  </eca:variable>
</eca:rule>

binds variables:
- Flight, Minutes: by matching
- theSeq is bound to the sequence of events that matched the pattern
Tasks

- ECA Engine: Rule Semantics
  - Control flow: registering event component, receiving “firing” answer, continuing with queries etc.
  - Variable Bindings, Join Semantics
- Generic Request Handler: Mediator with Component Engines
  - depending on Service Descriptions
- Component Engines: dedicated to certain Event Algebras, Query Languages, Action Languages
- Domain Services (Portals): atomic events, queries, atomic actions
ECA Architecture

ECA Engine:

```xml
<rule>
  <event xmlns:ev="..."/>
  <query xmlns:ql="..."/>
  <test xmlns:tst="..."/>
  <action xmlns:act="..."/>
</rule>
```

Component Language Services

- Travel:
- Banking:
- LH
- SNCF

Domain Services

- Uni:

Individual Services
Communication of Variable Bindings

XML markup for communication of variable bindings:

```xml
<log:variable-bindings>
  <log:tuple>
    <log:variable name="name" ref="URI"/>
    <log:variable name="name">any value</log:variable>
  </log:tuple>
  <log:tuple>...
  <log:tuple>...
</log:variable-bindings>
```
Communication ECA → GRH

- the component to be processed
- bindings of all relevant variables

[Sample: a query component]

```xml
<eca:query xmlns:ql="url"
   rule="rule-id" component="component-id">
   <!-- query component -->
   <eca:query>
   <log:variable-bindings>
   <log:tuple> ... </log:tuple>
   : 
   <log:tuple> ... </log:tuple>
   </log:variable-bindings>
</eca:query>
```

- `url` is the namespace used by the event language
- identifies appropriate service
Communication

ECA engine sends component to be processed together with bindings of all relevant variables to GRH.

Generic Request Handler (GRH)

- Submits component (with relevant input/used variable bindings) to appropriate service (determined by namespace/language used in the component)
- if necessary: does some wrapping tasks (for non-framework-aware services)
- receives results and transforms them into flat variable bindings and sends them back to the ECA engine ...
- ... where they are joined with the existing tuples ...
- ... and the next component is processed.
Communication Component Engine → GRH

result-bindings-pairs (semantics of expression)

```xml
<log:answers rule="rule-id" component="component-id">
  <log:answer>
    <log:result>
      <!-- functional result -->
    </log:result>
    <log:variable-bindings>
      <log:tuple> ... </log:tuple>
      :
      <log:tuple> ... </log:tuple>
    </log:variable-bindings>
  </log:answer>
  <log:answer> ... </log:answer>
  :
  <log:answer> ... </log:answer>
</log:answers>
```
Communication GRH → ECA

- set of tuples of variable bindings (i.e., input/used variables and output/result variables)
- is then joined with tuples in ECA engine
- ... and next component is processed