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 REWERSE® 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 \textit{in the Semantic Web}
- semantic description \textit{of} behavior

\textbf{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](image)

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

raise

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

ECA Language:

\[
\text{ECA Language :} \quad <\text{event/} > <\text{query/} > <\text{test/} > <\text{action/} >
\]
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>
Binding and Use of Variables in ECA Rules

\[
\text{action}(X_1, \ldots, X_n) \leftarrow
\]

\[
event(X_1, \ldots, X_k), \query(X_1, \ldots, X_k, \ldots X_n), \test(X_1, \ldots, X_n)
\]

(Composite) Event Detection Engine

Event Detection Engine

 register event comp.
 upon detection: result variables

Query Engine

 send query, receive result

Semantic Web: Domain Brokers and Domain Nodes

Action/Process Engine

 send action, + vars
<!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="{${flight}}"]/passenger/@e-mail
      </eca:opaque>
    </eca:variable>
    <eca:action xmlns:smtp="...">
      <smtp:send-mail to="${email}" text="..."/>
    </eca:action>
  </eca:query>
</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), ă (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 Engine**
- **Action Component Language, e.g. CCS**
- **Composer**
  - name

**Process Algebra Responsibility**

**Other Responsibilities**

- **Domain Broker**
- **Domain Nodes**
- **Domain Language**
  - uri
- **Event Detector**
- **Query Engine**
- **Event Language**
- **Query/Condition Language**
- **Atomic Events**
- **Literals**
- **Atomic Actions**

**Embeds**

- 1..*
- *
- *

**Uses**
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
1.2: register event
   travel: match: snoop:

2.2: atomic events
   travel:

4: detected parameters

3: detected parameters

1.3: atomic event patterns
   match: travel:

5.2a: atomic actions
   travel:

2.1b: atomic events
   travel:

5.2b: atomic actions
   smtp:

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

ECA Engine
eca:

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

1.4: register me
travel:

5.1: action
ccs: travel: smtp:

5.3a: booking
travel:

5.3b: message
(here: confirm)
by url

5.1: booking
travel:

Domain Broker
travel:

Event Detection
snoop:

Action Engine
ccs:

SMTP Mail Service
smtp:

Atomic Event Matcher
match:

Atomic Events
travel:

Lufthansa
travel:

SNCF
travel:

Client
travel:

Language Services Application Domain

Application Domain

ECA-CCS
Comparison

- CCS (extended with events and queries) strictly more expressive than ECA rules alone:
  ECA pattern in CCS: `event:condition:action`,
- many ECA rules have much simpler actions and do not need CCS,
- useful to have CCS as an *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)

\[
\begin{align*}
\text{Action Component: Process Algebras} \\
\text{Example: } & CCS \text{ (Calculus of Communicating Systems, Milner 1980)} \\
\text{Describes the execution of processes as a transition system:} \\
\text{(only the asynchronous transitions are listed)} \\
\end{align*}
\]

\[
\begin{align*}
a : P \xrightarrow{a} P \\
\sum_{i \in I} P_i \xrightarrow{a} P \\
P \xrightarrow{a} P' & \quad \Rightarrow \\
Q \xrightarrow{a} Q' & \quad \Rightarrow \\
P \parallel Q \xrightarrow{a} P' \parallel Q & \\
P \parallel Q \xrightarrow{a} P \parallel Q' & \\
P_i \{\text{fix } \vec{X} \overrightarrow{P} / \vec{X} \} \xrightarrow{a} P' & \\
\text{fix}_i \vec{X} \overrightarrow{P} \xrightarrow{a} P' & \\
\end{align*}
\]
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

Composite Event Specification

Domain Event

describes

Domain Ontology

uses

from

Atomic Event Description Formalism

EventComposer

cardinality

EventAlgebra

identifier
<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:
<rule>
  <event xmlns:ev="\ldots"/>\ldots</event>
  <query xmlns:ql="\ldots"/>\ldots</query>
  <test xmlns:tst="\ldots"/>\ldots</test>
  <action xmlns:act="\ldots"/>\ldots</action>
</rule>

Component Language Services

ECA-CCS

Generic Request Handler

Component Language Services

travel:
banking:

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:variable-bindings>
```
Communication ECA $\rightarrow$ 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:variable-bindings>
  </eca:query>
</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.
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