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

Abstract

We describe how Event-Condition-Action (ECA) rules can be combined with Process Algebras like CCS as specification of the action component to obtain a powerful, declarative formalism that also covers intuitively procedural tasks in an appropriate way. Since both formalisms have a concise formal semantics, verification and other kinds of reasoning about such specifications are possible. Using a rule markup with cleanly distinguished rule components allows for such a compositional approach. The approach is currently under implementation in a General ECA Framework for the Web and the Semantic Web.

1 Introduction

Event-Condition-Action (ECA) rules are a popular paradigm in Event Processing: "ON event IF condition DO action" has a clear declarative semantics and induces an immediate operational realization, as e.g. provided on the low level by SQL triggers. Usually, the *event* part is either an atomic event or given by some formalism for specifying composite events (often derived from *event algebras*. The *condition* is in most cases expressed by a database query language. The *action* part is often given as a program (e.g., in PL/SQL) in a procedural, non-declarative way.

For the latter, the use of *Process Algebras* provides a mechanism to provide also a *declarative* specification. Furthermore, since process algebras allow not only for executing a piece of program code, but also the definition of more complex *processes*, including the definition of independent, communicating processes, the resulting model is also more expressive than current formalisms and languages for active rules.

Moreover, the combination on the semantical level of a *rule ontology* with a *process ontology* makes such rules full citizens of the Semantic Web. In addition to just executing such rules, they are embedded in a semantic-level model of behavior that allows for exchanging rules and components. Reasoning *about* them is also supported since verification of process algebra specifications is well-understood.

In [12] and [11] we described the global architecture and the general markup principles of a *General Framework for Behavior in the Semantic Web* that is based on the idea of ECA rules over heterogeneous event, condition, and action formalisms. The current paper provides the details of the action component according to this framework.

Structure of the paper. The paper is structured as follows: Section 2 gives an overview of the ECA framework for the (Semantic) Web where our proposal is embedded, and motivates the proposed extensions. We give an overview of CCS in Section 3 which provides the algebraic structure of the action component. While in the basic formalism of CCS, all atomic items are considered to be actions, we show in Section 4 that in the Semantic Web setting, some of these "actions" actually correspond to events or conditions/queries wrt. the underlying domains. We give the markup of the action component that then consists straightforwardly of the CCS term markup and the atomic items in Section 5. Section 6 then describes how the action component is processed by the services in the Framework. Section 7 concludes the paper.

2 Overview: The ECA Framework

The ECA Framework aims at adding behavior to the Web and to the Semantic Web in form of ECA rules acting in an environment of autonomous nodes of different types:

- Application nodes: they carry out the real "businesses", e.g., airlines, train companies, or universities. In the Semantic Web, applications in the same domain should use an agreed domain ontology, e.g. a traveling ontologies could be shared by airlines and train companies.
- Language nodes: they support generic specification languages such as nodes that execute ECA rules, or that implement certain composite event specification formalisms, query languages, or the CCS-style language presented in this paper.
- Infrastructure nodes: they provide infrastructure as mediators between language nodes and domain nodes. Infratruture nodes include the domain brokers that implement a portal functionality for a given domain based on its ontology, and *Language and Service Registries* that serve for finding language or domain nodes.

2.1 Domain Ontologies

Application nodes provide functionality in certain domains. Every domain ontology – e.g. for banking or traveling – defines *static notions* (classes, relationships) and *dynamic notions* such as events and actions as shown in Figure 1.

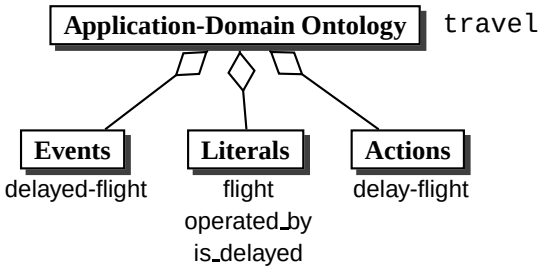

Figure 1: Components of Ontologies

Usually, a domain is supported by many domain nodes. Additionally, *domain brokers* act as mediators for a domain.

2.2 ECA Rules

The core of the approach is a model and architecture for ECA rules that use heterogeneous actual event, query, and action languages. The condition component is divided into queries (that can be expressed in different languages) and a test component:

ON *event* AND *additional knowledge*, IF *condition* THEN DO *something*.

The relevant atomic events and atomic actions are usually events and actions from the domain ontologies. Events are raised by application nodes (e.g. Lufthansa), and communicated by infrastructure nodes like domain brokers. Rules can react on atomic events, or on composite ones (defined e.g. by an event algebra). Detection of composite events is done by dedicated services. The ECA engine registers relevant event patterns there, and upon detection of such an event, the service reports it to the ECA engine. The ECA engine then evaluates additional queries against the Semantic Web for obtaining further information and then evaluates the condition. Finally, it executes the action component. Analogous to the event component, the action component of a rule can be an atomic action (e.g., an action of the domain ontology, sending a message, or also raising an event), or a composite action. In our approach, we propose to specify composite actions by a *process algebra*.

For dealing with heterogeneous languages, the approach is parametric in the used component languages. Users register rules where they use component languages of their choice at an ECA service in the Web that implements the high-level control.

The markup of the rules in the proposed ECA-ML language [11] indicates the "language borders" between the ECA level and the nested components by their namespaces; as indicated below in an example using a SNOOP [5]-style language as event algebra (implemented as a prototype). Figure 2 (from [11]) illustrates the structure of the rules and the corresponding types of languages.

```
<eca:rule xmlns:eca="http://.../eca/2006/eca-ml">
  <eca:event xmlns:snoopy=
    "http://www.semwebtech.org/eca/2006/snoopy">
    <snoopy:sequence>
      nested expression in the event language markup
    </snoopy:sequence>
  </eca:event>
  <eca:query xmlns:ql1="uri of query language ql_1">
    nested expression in ql_1 markup
  </eca:query>
   :
  <eca:query xmlns:qln="uri of query language ql_n">
    nested expression in ql_n markup
  </eca:query>
  <eca:test xmlns:cl="uri of condition language">
    test expression over obtained information
  </eca:test>
  <eca:action xmlns:ccs="http://.../eca/2006/ccs">
    nested expression in CCS as described below
  </eca:action>
</eca:rule>
```

While the semantics of the ECA *rules* provides the infrastructure and global semantics, the components are handled by specific services that implement the respective languages. The services are identified via the language namespaces.

2.3 Services for Component Languages

The components are specified as nested subexpressions of the form

```
<eca:component xmlns:lang="embedded-lang-ns-uri">
  embedded fragment in embedded language's
  markup and namespace
</eca:component>
```

in arbitrary formalisms or languages. For processing the components, a *language processor node* for the indicated specification language is determined (by querying a *Language and Service Registry* infrastructure node) and the task is submitted to this node.

2.4 Communication Markup

For dealing with heterogeneous languages, the approach does only minimally constrain the component languages: Information flow between the ECA engine and the Event, Query, Test, and Action components is provided by *logical variables* in the style of deductive rules, production rules etc. Thus, e.g., languages following a functional style

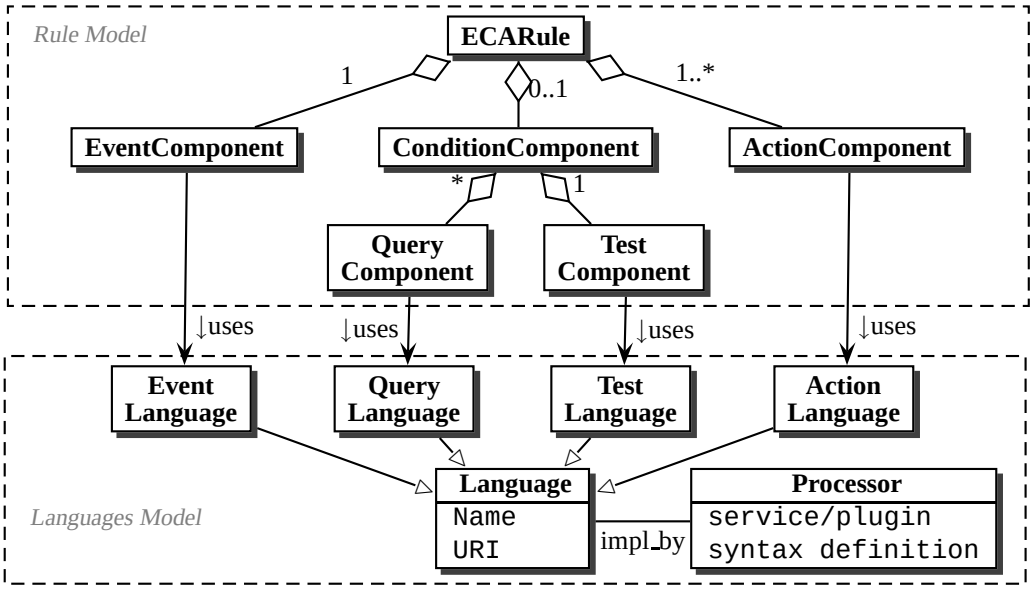

Figure 2: ECA Rule Components and Corresponding Languages (from [11])

(such as XPath/XQuery), a logic style (such as Datalog or SPARQL [17]), or both (F-Logic [10]) can be used as query languages. The semantics of the event part (that is actually a "query" against an event stream that is evaluated incrementally) is –from that point of view– very similar, and the action part takes variable bindings as input.

Given this semantics, the ECA rule combines the evaluation of the components as follows in the style of production rules (cf. Figure 3):

$$action(X_1, \ldots, X_k) \leftarrow event(X_1, \ldots, X_n),$$
$$query(X_1, \ldots, X_n, \ldots X_k),\ test(X_1, \ldots, X_k)\ .$$

The evaluation of the event component (i.e., the successful detection of a (composite) event) binds variables to values that are then extended in the query component, possibly constrained in the test component, and propagated to the action component.

The ECA engine processes the rule components by submitting the component together with the actual variable bindings in the below format to an appropriate service:

```
<log:variable-bindings xmlns:log="http://.../2006/logic">
  <log:tuple>
    <log:variable name="X">value</log:variable>
    <log:variable name="Y">value</log:variable>
    <log:variable name="Z">value</log:variable>
  </log:tuple>
  <log:tuple>
    <log:variable name="X">value</log:variable>
    <log:variable name="Y">value</log:variable>
    <log:variable name="Z">value</log:variable>
  </log:tuple>
    :
</log:variable-bindings>
```

Apart from [12] and [11] that describe the global architecture and the general markup principles, a detailed description of the ECA level can be found in [2]. In this paper, we focus on the action component.

3 The Action Component

3.1 Choice of Formalism

The basic and pure form of ECA rules just supports a simple kind of action specifications, given as a set or sequence of actions that is to be executed.

For more sophisticated tasks and a more intuitive specification, we propose to use a process algebra (e.g., CCS [13]) in the action component. With this, the action component can be used to specify e.g., the following concepts:

1. a sequence of actions to be executed (as in simple ECA rules),
2. a process that includes "receiving" actions (which are actually events in the standard terminology of ECA rules),
3. guarded (i.e., conditional) execution alternatives,
4. the start of a fixpoint (i.e., iteration or event infinitely running process), and
5. a family of *communicating, concurrent processes*.

The above patterns can be employed as follows for specifying behavior: (2) above can e.g. be used to define a negotiation strategy that communicates with a counterpart. (3) can include different reactions to the answers of the counterpart, (4) extends the behavior even to try again. Note

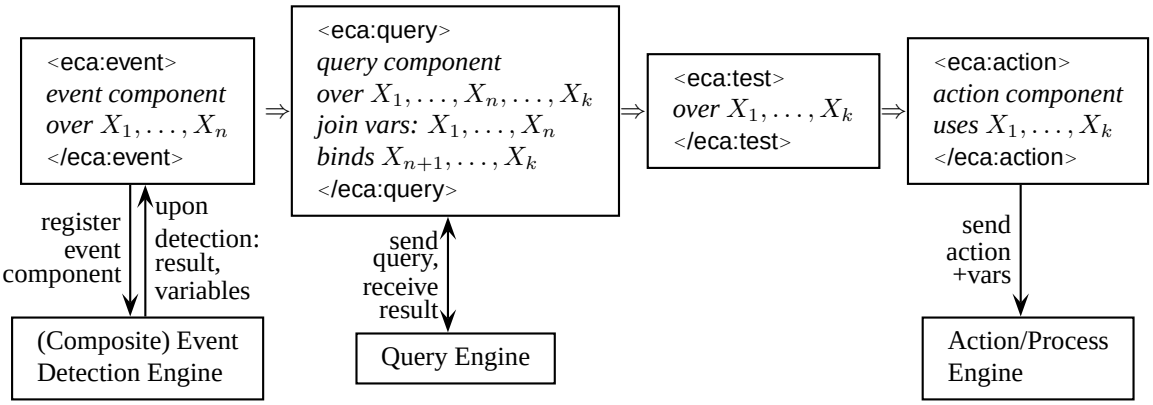

Figure 3: Use of Variables in an ECA Rule

that in these cases, only one side of the communication is specified, whereas the behavior of the counterpart is defined by other rules (with an other owner). (5) can be used to specify even more complex behavior as a reaction as known from the agent community.

These tasks can also be expressed by (sets of) simple ECA rules, but this leads to a much less intuitive, and hard-to-understand specification. The composition of the ECA and process algebra concepts (and ontologies) provides a comprehensive framework for describing behavior in the Semantic Web.

3.2 The CCS Process Algebra

Process Algebras describe the semantics of processes in an algebraic way, i.e., by a set of elementary processes (carrier set) and a set of constructors. The semantics can either be given as *denotational semantics*, i.e., by specifying the denotation of every element of the algebra (e.g., CSP – Communicating Sequential Processes, [9]), or as an *operational semantics* by specifying the behavior of every element of the algebra (e.g., CCS – Calculus of Communicating Systems, [13, 14]). Processes defined by Process Algebras can e.g. be used for the specification of *communication*, i.e., for basic protocols, or for defining the behavior of interacting (Semantic) Web Services (note that process algebras provide concepts for defining infinite processes), or in the action part of ECA rules.

Basic Process Algebra (BPA). A simple algebra that uses only the combinators $x+y$ and $x \cdot y$ for processes (alternative and sequential communication) is known as BPA, defining essentially the processes that can be characterized in Dynamic Logic [8] and Hennessy-Milner-Logic [15].

In many cases of the ECA Framework, BPA is sufficient for specifying the action component.

Calculus of Communicating Systems (CCS). CCS extends BPA by more expressive operators. The carrier set of a CCS [13] algebra is given by a set $\mathcal{A}$ of action names from which processes are built by using several connectives. Every element of the algebra is called a *process*. By carrying out an action, a process changes into another process. Considering the modeling as an Labelled Transition System, a process can be regarded as a state or a configuration, which allows to use Model Checking for verifying properties of CCS specifications. Action names become labels and the transition relation is given by the rules specifying the execution of actions.

A CCS algebra with a carrier set $\mathcal{A}$ is defined as follows, using a set of process variables:

1. With X a process variable, X is a process expression.
2. Every $a \in \mathcal{A}$ is a process expression.
3. With $a \in \mathcal{A}$ and P a process expression, $a : P$ is a process expression (prefixing; sequential composition).
4. With P and Q process expressions, $P \times Q$ is a process expression (parallel composition).
5. With I a set of indices, $P_i : i \in I$ process expressions, $\sum_{i \in I} P_i$ (binary notation: $P_1 + P_2$) is a process expression (alternative composition).
6. With I a set of indices, $X_1, \ldots, X_k$ process variables, and $P_1, \ldots, P_k$ process expressions, $\mathsf{fix}_j \vec{X} \vec{P}$ is a process expression (definition of a communicating system of processes). The fix operator binds the process variables X_i, and fix_j is the jth one of the k processes which are defined by this expression.

Process expressions not containing any free process variables are *processes*.

The (operational) semantics of a CCS algebra is given by

transition rules [13]:

$$a : P \xrightarrow{a} P \quad , \quad \frac{P_i \xrightarrow{a} P}{\sum_{i\in I} P_i \xrightarrow{a} P} \text{(for } i \in I\text{)}$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q'}{P \times Q \xrightarrow{ab} P' \times Q'} \quad , \quad \frac{P_i\{\mathsf{fix}\ \vec{X}\vec{P}/\vec{X}\} \xrightarrow{a} P'}{\mathsf{fix}_i \vec{X}\vec{P} \xrightarrow{a} P'}$$

Additionally, asynchronous CCS allows for delays:

$$\begin{array}{lcl} \partial P & := & \mathsf{fix}\ X(1 : X + P)\ ,\ X \text{ not free in } P, \text{ and} \\ P_1|P_2 & := & P \times \partial Q + \partial P \times Q \\ a.P & := & a : \partial P\ . \end{array}$$

with the corresponding transition rules

$$\partial P \xrightarrow{1} \partial P \quad , \quad \frac{P \xrightarrow{a} P'}{\partial P \xrightarrow{a} P'}$$

$$\frac{P \xrightarrow{a} P'}{P|Q \xrightarrow{a} P'|Q} \quad , \quad \frac{Q \xrightarrow{a} Q'}{P|Q \xrightarrow{a} P|Q'}$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q'}{P|Q \xrightarrow{ab} P'|Q'}$$

The possibility of Delay is especially important when "waiting" for something to occur, e.g., for synchronization.

4 The Atomic Actions

Usually, CCS is presented with symbolic atomic action names a_1, a_2 etc. For the application in the ECA Framework, the atomic actions are taken from those of the application domains, e.g. "travel:delay-flight('LH123','30 min')" when flight LH123 has to be delayed by 30 min. Such atomic actions are then executed by the appropriate node of the Web (depending on the context, here: the Lufthansa Web node). Furthermore, the carrier set of the process algebra expression is extended with several kinds of pseudo-actions as follows:

"Inverse" Communication Actions. For modeling communication, actions a_1, a_2 in CCS etc. come with an "inverse" $\bar{a}_1$, $\bar{a}_2$, that correspond as input/output or sending/receiving, that can be executed together as an *internal* transition (i.e., which is not visible to the outside, often denoted by τ):

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P \times Q \xrightarrow{\tau} P' \times Q'}$$

In terms of the ontology of behavior defined by the ECA Framework, these "inverse" actions correspond more to "events" than actions. Note that in CCS, such specifications often use the delay (∂)-operator for waiting for snychronization. We will integrate them in the following as events.

Actions with Parameters. Actions are usually parameterized, e.g. "book flight no N on $date$". Communication between the rule components is provided by variable bindings. Accordingly, the specification of the action component uses variables as parameters to the actions.

Example 1

- *A money transfer (from the point of the view of the bank) is already a simple process:*
  transfer(Am, Acc_1, Acc_2) :=
  debit(Acc_1, Am) : deposit(Acc_2, Am) .
- *a standing order (i.e., a banking order that has to be executed regularly) is defined as a fixpoint process, involving an event The following process transfers a given amount from one account to another every first of a month (where "first_of_month is a temporal event):*
  fix X.(first_of_month : debit(Acc_1, Am) :
  deposit(Acc_2, Am) : ∂ X)
- *A more detailed view could e.g. check if the balance will stay positive, and if not, notify the account holder:*
  fix X.(first_of_month : send_query($Acc_1 \geq Am$?) :
  ((∂ : rec_msg(yes) :
  debit(Acc_1, Am) : deposit(Acc_2, Am)) +
  (∂ : rec_msg(no) : send_msg($owner,...))) : ∂ X)
  (using messaging for queries and message receipt events for answers).

In this example, the fact that it is the first of a month is communicated explicitly by sending (issued e.g. by a timer process) and receiving actions.

Another way would be to express the same as a complete ECA rule "if the event first_of_month occurs, then do ..." instead of a fixpoint process.

Example 2 *Consider the following scenario: if a student fails twice in an exam, he is not allowed to continue his studies. If the second failure is in a written exam, it is required that another oral assessment takes place for deciding upon final passing or failure.*

This can be formalized as an ECA rule that reacts upon an event failed($Subject,$StudNo) and then in a further query checks whether this is the second failure of $StudNo in $Subject, and whether the exam was a written one. The action component of the rule should then specify the process of (organizing) the additional assessment: as an action, the responsible lecturer will be asked for a date and time (send a mail), that will be entered by him into the system (in CCS: a "receiving" communication action; in our approach: an event),
The action component is thus as follows:

ask_appointment($Lecturer, $Subject, $StudNo) :
∂ propose_appointment($Lecturer, $Subject, $DateTime) :
find_room($DateTime, $Room) :
inform($StudNo, $Subject, $DateTime, $Room) :
inform($Lecturer, $Subject, $DateTime, $Room)

In this example,
propose_appointment($Lecturer, $Subject, $DateTime) is an event – for this, it is allowed to be delayed (∂). In contrast, all other items are actions that are actually executed by the process as soon as possible.

Note that entering the grade and further consequences are not covered by this action. Instead, it is appropriate to have a separate rule that reacts (again) on entering grades and, if the grade was established by such an additional assessment, take appropriate actions.

Conditions. In CCS and related concepts, such as CSP [9] and ACP [3], there is no explicit notion of states, the properties of a state are given by the (sequences of) actions which can be executed. When representing a stateful process, queries and values are represented e.g., as "read that $A > 0$", or by explicit messages (as the account balance in Example 1). We omit the "read", and allow queries and conditions as regular components of a process:

- "executing" a query means to evaluate the query, extend the variable bindings, and continue.
- "executing" a condition means to evaluate it, and to continue for all variable bindings that evaluate to "true". Note that for a conditional alternative $((c : a_1)+(\neg c : a_2))$, all variable bindings that satisfy c will be continued in the first branch, and the others are continued with the second branch.

Example 3 (Processes with Conditions)

- *Consider again the scenario from Example 2, but now only one room is suitable for such assessments. Here, the process in the action part must iterate asking the lecturer for an alternative date/time until the room is available. This is done by combining CCS's fixpoint operator with a conditional alternative:*
  fix$X$.(ask_appointment($Lecturer,$Subj,$StudNo) :
  ∂ propose_appointment($Lecturer,$Subj,$DateTime) :
  (available(room,$DateTime) +
  (not_available(room,$DateTime) : X))) :
  inform($StudNo,$Subj,$DateTime) :
  inform($Lecturer,$Subj,$DateTime)
- *The account check in Example 1 can also be expressed by a conditional alternative:*
  fix$X$.(first_of_month :
  (($Acc_1 \geq Am$? : debit(Acc_1, Am) : deposit(Acc_2, Am)) +
  ($Acc_1 < Am$? : send_msg($owner,...))) : ∂ X)

Figure 4 shows the relationship between the generic process algebra language and the contributions of the domain languages and the event and test component languages.

5 The Language Markup

Thus, in our approach, processes are built over

- actions,
- events, and
- conditions

by using the CCS connectives. The language markup has the usual form of a tree structure over the CCS composers in the ccs namespace. The leaves are contributed by (i) atomic actions of the underlying domains, (ii) events and conditions/tests. The latter are not necessarily atomic, but are seen as black-boxes from the CCS point of view, containing markup from appropriate languages as used in the ECA event and test components (and handled by the respective services).

5.1 Atomic and Leaf Items

As discussed above, the atomic items can be atomic actions, or embedded events or test subexpressions. In accordance with ECA-ML [11], the latter are embedded into ccs:event and ccs:test elements. The language identification is done again via the namespaces.

Atomic Actions. Atomic actions belong to some domain namespace, thus the element is in general the action in XML markup "itself" (including variables as {$varname}):

```
<domain-ns:action-name attributes>
  contents
</domain-ns:action-name>
```

As an example consider an atomic action that books a given flight (flight code bound to variable $flight) at a given date (bound to variable $date):

```
<travel:book-flight code="{$flight}" date="{$date}"/>
```

Embedded Events. Embedded events are also leaves, contained in ccs:event elements (with the same semantics as eca:event elements on the ECA level):

```
<ccs:event xmlns:el="ev-uri">
  event expression in appropriate markup
</ccs:event>
```

Arbitrary event languages and formalisms that are supported by some service are allowed. Note that composite events integrate smoothly since they are considered to occur with the final detection of the composite event.

Embedded Conditions. Embedded tests are handled exactly in the same way:

```
<ccs:test xmlns:cl="cl-uri">
  test expression in appropriate markup
</ccs:test>
```

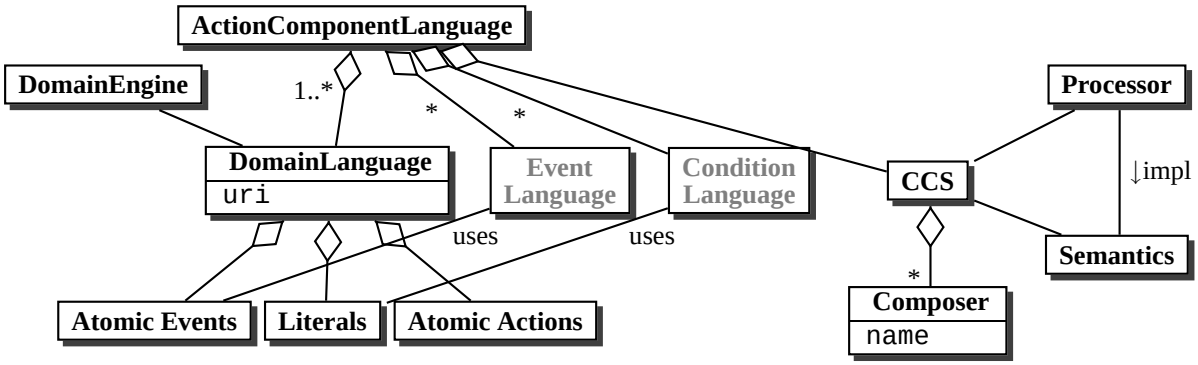

Figure 4: Structure of the Action Component as an Algebraic Language using CCS

Embedded Opaque Items. Opaque actions (i.e., program code, mainly for queries, tests and also for actions) can be embedded as leaf elements:

<ccs:opaque {url="node-url"|language="name"}>
program code fragment
</ccs:opaque>

For such fragments, either an URL where the action has been sent to (as HTTP GET) is given, or a the language is indicated (then the fragment must contain the addressing of the target node itself).

5.2 CCS Algebra

Following a straightforward principle for term markup, the CCS operators are represented by XML elements (with parameters as attributes) according to the following nearly-DTD specification:

```
<!ENTITY % operand "(delay | sequence |
   alternative | concurrent | fixpoint |
   atomic-action | event | query | test |
   opaque)">
<!ELEMENT delay EMPTY>
<!ELEMENT sequence
  (%operand;, %operand;+)>
  <!ATTLIST sequence mode "async">
<!ELEMENT alternative
  (%operand;, %operand;+)>
<!ELEMENT concurrent
  (%operand;, %operand;+)>
<!ELEMENT fixpoint (%operand;+)>
  <!ATTLIST fixpoint
     variables #REQUIRED NMTOKENS
     index #REQUIRED NMTOKEN
     localvars #IMPLIED NMTOKENS>
<!ELEMENT atomic-action ANY>
<!ELEMENT event ANY>
<!ELEMENT query ANY>
<!ELEMENT test ANY>
<!ELEMENT action ANY>
  <!ATTLIST event,test,action
          xmlns:%name; #REQUIRED %URI;>
<!ELEMENT opaque ANY>
  <!ATTLIST opaque
     language #IMPLIED CDATA
     url #IMPLED CDATA>
  <!ATTLIST all-elements group-by "">
```

The semantics of the elements is described below.

- The content of all the "simple" operators consists of at least two subelements of the ccs namespace.
- <ccs:delay/> indicates a delay (for waiting, in case that synchronous context is used),
- <ccs:seq mode="*mode*">*contents*</ccs:seq> indicates a sequence. *mode* can be sync or async corresponding to synchronous CCS (with ":" as standard combinator) or asynchronous CCS (with "." as standard combinator); default is mode="async".
- <ccs:alt>*contents*</ccs:alt> stands for "$\sum$" and "+" (alternatives),
- <ccs:concurrent mode="*mode*">*contents*</ccs:concurrent> represents "$\times$" and "|" (parallel),
- The <ccs:event xmlns:*lang*="*uri*">, <ccs:query xmlns:*lang*="*uri*">, <ccs:test xmlns:*lang*="*uri*">, and <ccs:action xmlns:*lang*="*uri*"> elements allow for embedded events, queries, tests, or actions (the latter even allow for embedding an action/process specification in another language).

Handling of "new" Variables in Fixpoint Processes. For integration with the ECA Framework that uses logical variables (that can be bound only once), variables that are bound during the evaluation of the fixpoint part must be considered to be local to the current iteration, and only the final result is then bound to the actual logical variable:

- <ccs:fixpoint variables="$var_1 \ldots var_n$" index="j"
  localvars="*list of variables*">
  contents</ccs:fixpoint>

provides the markup for fixpoint constructs. The var_i are the process variables, j is the index of the one of the processes that is chosen, and the variables distinguished to be local can be bound in each iteration; after reaching the fixpoint they keep the value of the last iteration.

Example 4 (Variables in Fixpoint Processes) *Consider again Example 3(2). There, each iteration of the fixpoint process searching for a date where the room is available binds $DateTime. The actual semantics is easy to understand and implement: just keep the last value.*

Grouping. For each subexpression, it can be specified if it is executed for the whole set, or separately for each tuple, or some grouping (in the same way as grouping in SQL) is applied. Clearly, subactions can only have finer granularity than the outer expressions). For specifying grouping, each action element has an optional attribute

group-by="*variable list*"

that indicates grouping. E.g., given variables X, Y, Z, group-by="X Y" means to execute the subexpression separately for all sets that have X and Y in common. Default is default is group-by="" which means to have one group with all tuples. For convenience, group-by="-separately" mans to process every tuple separately, and group-by="-bulk" also means to have one group with all tuples.

5.3 Example

Consider a rule that does the following: if a flight is first delayed and then cancelled (note: use of a join variable), make a reservation for each passenger at the airport hotel, and send each business class passenger an SMS.

```
<eca:rule xmlns:eca="http://.../eca/2006/eca-ml">
 <eca:event xmlns:snoopy="http://.../eca/2006/snoopy">
  <snoopy:sequence>
   <travel:delayed-flight flight="{$flight}"/>
   <travel:canceled-flight flight="{$flight}"/>
  </snoopy:sequence>
 </eca:event>
 <eca:query>
  <eca:opaque language="xpath" variable="$passenger">
   //flights/flight[code=$flight]/passenger
  </eca:opaque>
 </eca:query>
 <eca:query>
  <eca:opaque language="xpath" variable="$name">
   string($passenger/name)
  </eca:opaque>
 </eca:query>
 <eca:action xmlns:ccs="http://.../eca/2006/ccs">
  <ccs:par>
   <ccs:atomic-action>
    <travel:reserve-room hotel="hotel uri" name="$name"/>
   </ccs:atomic-action>
   <ccs:sequence>
    <ccs:test >
     <eca:opaque language="xpath">
      $passenger/@class="business"
     </eca:opaque>
    <ccs:test>
    <ccs:query>
     <eca:opaque language="xpath" variable="$phone">
      string($passenger/phone)
     </eca:opaque>
    </ccs:query>
    <ccs:atomic-action>
     <comm:send-sms to= "$phone">
      "we are very sorry ... and booked a room in ... for you"
     </comm:send-sms>
    </ccs:atomic-action>
   </ccs:sequence>
  </ccs:par>
 </eca:action>
</eca:rule>
```

- after the event part, $flight is bound to the flight number,
- after the queries, for each passenger there is a tuple of bindings $flight, $passenger, $name where $passenger holds the XML record of that passenger and $name holds the name.
- The action component does two things in parallel: reserve rooms, and for each tuple (i.e., for each passenger) check if it is a business class passenger (test), if yes, get his phone number (query) and send an SMS.

6 Processing

The actual processing of the action component with its embedded expression also makes use of the ECA-ML infrastructure as shown in Figure 5:

- Actions are executed "immediately" by submitting them to the domain nodes (if specified by an URL) or to a domain broker (responsible for the domain, forwarding them to appropriate domain nodes),
- Events, or, more exactly, event *patterns* to be detected are submitted to an appropriate event detection service in the same way as <eca:event> components of ECA rules.
- queries and tests are evaluated by appropriate language processors. A standard language for tests consists of boolean connectives and XPath/XQuery-built-in predicates can be evaluated locally.

The tasks for processing embedded <ccs:event>, <ccs:query>, <ccs:test>, and <ccs:action> elements

with embedded fragments of other languages are closely related to each other (and actually the same as for <eca:event>, <eca:query>, <eca:test>, and <eca:action> elements on the ECA level). Such elements are of the form

```
<ccsns:event xmlns:lang="embedded-lang-ns">
  embedded fragment in embedded language's markup
</ccsns:event>
```

For processing the components, a *language processor node* for the indicated specification language is determined (by querying a *Language and Service Registry* infrastructure node) and the task is submitted to this node. As described in Section 2.4, the communication in the framework is standardized to use the format of variable bindings. The actual process of determining an appropriate service and organizing the communication is performed by a *Generic Request Handler* that has been implemented in [2]. Thus, the implementation of the CCS engine is only concerned with the actual CCS operators.

7 Conclusion

In this work, we presented the combination of a general ECA architecture with CCS as a specification formalism for processes. We have shown that both paradigms can be intertwined to profit from each other by using well-defined mechanisms for language identification and communication. The resulting framework provides a concise logical semantics (with which also the specification of the event component by event algebras fits).

7.1 ECA Rules vs. CCS

In general, it is possible to decompose a CCS description completely into ECA rules with atomic actions (or with restricting the action component to BPA process specifications), or to express ECA rules as a special form of CCS processes over the above-mentioned atomic items. The advantage of supporting both formalisms in the framework lies in the appropriateness of modeling: there is behavior that is preferably and inherently formalized as ECA rules, and there is behavior that is preferably formalized in CCS. Providing both formalisms thus eases the specification, and with this also the understandability and maintenance of behavior:

> "If something happens (event) in a certain situation (condition), then proceed according to a given policy (action, as CCS process)."

Having ECA rules and processes allows to model both *reactive* and *continuous* behavior in an appropriate way.

7.2 Ontology and Reasoning

The framework with its clean distinction between the notions of events, the static parts of a the domains, and actions also provides an *ontology of behavior*. Using the ontology, behavior cannot only be described and implemented on the level of syntax and operational semantics (as ECA, event algebras and process algebras do naturally), but also on a formal semantic level that allows to *reason* about it across the used formalisms. The formalisms implemented now are to be understood as samples, and multiple other formalisms with different abstraction level, expressiveness and complexity can be integrated with the ontology. At the end, our aim is to develop a rule editor that allows to specify rules and processes using the ontology notions, and automatically maps them onto appropriate formalisms for events and actions.

7.3 Related Work

So far, reasoning and verification, e.g., by model checking, have only be applied to individual formalisms. Mainly, the combination of static and dynamic aspects falls short in such approaches. Logic-based approaches that use formalizations far from "events" and "processes" have been discussed e.g. for the use of modal temporal logic for executable process specifications in [7], [6] (using full first-order past temporal logic, with $\exists$ and $\forall$ quantifiers), [16] (replacing the quantifiers by a functional assignment $[X \leftarrow t]\varphi(X)$ that binds a variable X) or, with a Transaction Logic [4] that proposes a specialized logic with temporal connectives. Transaction Logic is a comprehensive rule-based formalism that does not have a strict ECA distinction, but follows the Logic Programming style. In Transaction Logic, in contrast to modal logic where states are given as first-order structures, states are given as abstract *theories* over a signature $\mathcal{L}$ – i.e., it allows for embedding other formalisms here, that can e.g. be first-order theories, or, "now", OWL-based worlds. An overview of related formalisms can be found in [1].

7.4 Current State and Further Work

The approach is currently under implementation. The ECA module with the GRH are completed, together with a reference implementation of a SNOOP [5] -based event detection module. The current step completes the functionality of the component languages, including the above CCS service, and provides sample application nodes and basic infrastructure nodes like domain brokers.

The next steps will be concerned with completing the implementation and carrying out a larger case study in an application, investigating optimizations based on a cross-component algebraic theory, formalizing the behavior ontology in OWL and developing reasoning procedures.

References

[1] J. J. Alferes and W. May. Course: Evolution and reactivity for the web. In *REWERSE Summer School*, Springer LNCS 3564, pp. 134–172, 2005.

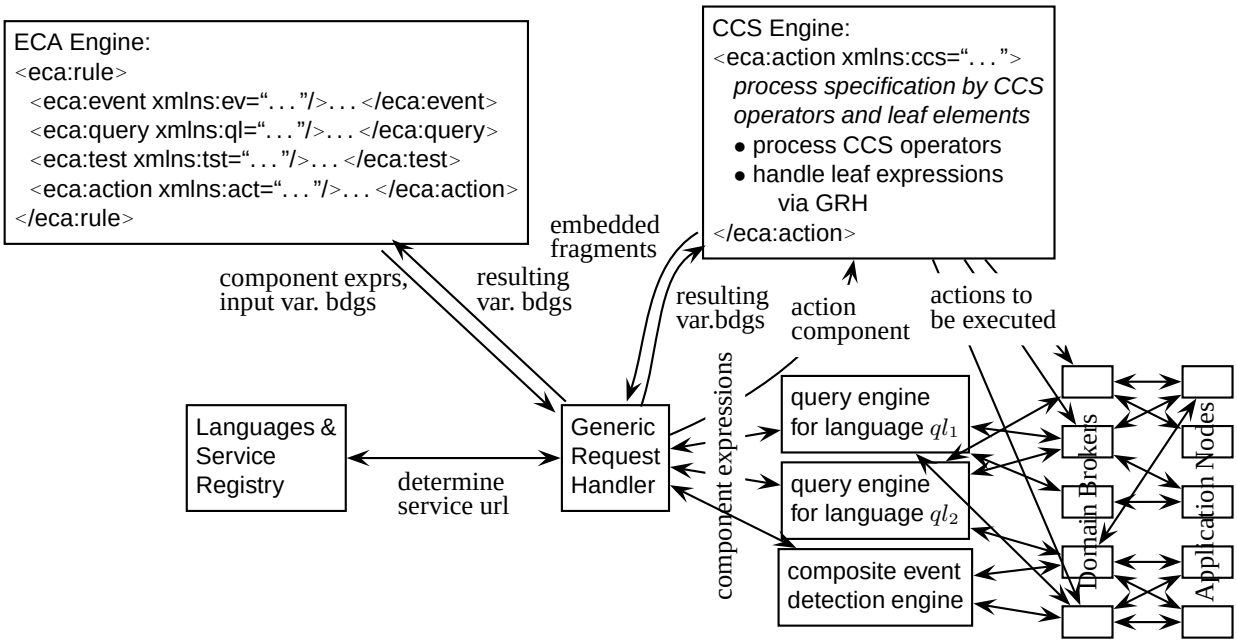

Figure 5: Architecture

[2] E. Behrends, O. Fritzen, W. May, and D. Schubert. An ECA Engine for Deploying Heterogeneous Component Languages in the Semantic Web. In *Web Reactivity (EDBT Workshop)*, to appear in Springer LNCS, 2006.

[3] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 1(37):77–121, 1985.

[4] A. J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133(2):205–265, 1994.

[5] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *VLDB*, pages 606–617, 1994.

[6] J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Transactions on Database Systems (TODS)*, 20(2):149–186, 1995.

[7] D. Gabbay. The declarative past, and imperative future: Executable temporal logic for interactive systems. In *Temporal Logic in Specification*, Springer LNCS 398, pp. 409–448, 1989.

[8] D. Harel. *First-Order Dynamic Logic*. Springer LNCS 68, 1979.

[9] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[10] M. Kifer and G. Lausen. F-Logic: A higher-order language for reasoning about objects, inheritance and scheme. In *SIGMOD*, pp. 134–146, 1989.

[11] W. May, José Júlio Alferes, and Ricardo Amador. Active rules in the semantic web: Dealing with language heterogeneity. *Rule Markup Languages (RuleML)*, Springer LNCS 3791, pp. 30–44, 2005.

[12] W. May, J. J. Alferes, and R. Amador. An ontology- and resources-based approach to evolution and reactivity in the semantic web. In *Ontologies, Databases and Semantics (ODBASE)*, Springer LNCS 3761, pp. 1553–1570, 2005.

[13] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, pages 267–310, 1983.

[14] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[15] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 1(100):1–77, 1992.

[16] A. P. Sistla and O. Wolfson. Temporal conditions and integrity constraints in active database systems. In *SIGMOD*, pp. 269–280, 1995.

[17] SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`, 2006.