# A General Framework for Evolution and Reactivity in the Semantic Web

**José Júlio Alferes**    **Wolfgang May**

CENTRIA, Universidade Nova de Lisboa, Portugal

Institut für Informatik, Universität Göttingen, Germany

Supported by the EU Network of Excellence

REWERSE©

reasoning on the web

# Excerpts of this talk ...

... have been given on different aspects at the following events:

- PPSWR 2005, Dagstuhl, Germany, Sept. 12-16, 2005:
  A General Language for Evolution and Reactivity in the Semantic Web
- ODBASE 2005, Agia Napa, Cyprus, Okt. 31 - Nov. 4, 2005:
  An Ontology- and Resources-Based Approach to Evolution and Reactivity in the Semantic Web
  (Ontology of rules, rule components and languages, and the service-oriented architecture)
- RuleML 2005, Galway, Ireland, Nov. 10-12, 2005:
  Active Rules in the Semantic Web: Dealing with Language Heterogeneity
  (Languages and their markup, communication and rule execution model)
- REWERSE A3-I4 Meeting, Hannover, Germany, Nov. 21/22, 2005:
  A General Framework for Evolution and Reactivity in the Semantic Web

# Further Contributors

- At DBIS, Universität Göttingen, Germany:
  Erik Behrends, Oliver Fritzen, Franz Schenk
  Students: Carsten Gottschlich, Elke von Lienen, Daniel Schubert, Sebastian Spautz
- At CENTRIA, Universidade Nova de Lisboa, Portugal:
  Ricardo Amador
  Students:

# Part I: Overview and Situation

# 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

# Behavior

- evolution of *individual* nodes (updates + reasoning)
- *cooperative* evolution of the Web (local behavior + communication)
- different abstraction levels and languages

# Behavior

- decentral P2P structure, autonomous nodes
- communication
- behavior located in nodes
  - local level:
    - based on local information (facts + received messages)
    - executing local actions (updates + sending messages + raising events)
  - Semantic Web level (in a given application area): execution located at a certain node, but “acting globally”:
    - global information base
    - global actions (including intensional RDF/OWL updates)

# Update Propagation and Semantic Updates

Overlapping ontologies and information between different sources:

- updates: in the same way as there are semantic query languages, there must be a semantic update language.
- updating OWL data: just tell (a portal) that a property of a resource changes
  intensional, global updates
  $\Rightarrow$ must be correctly realized in the Web!
- *reactivity* – see such updates as *events* where sources must react upon.

# Cooperative Evolution of the Semantic Web

There are not only *queries*, but there are *activities* going on in the Semantic Web:

- Semantic Web as a base for processes
  - Business processes, designed and implemented in participating nodes: banking, . . .
  - Predefined cooperation between nodes: travel agencies, . . .
  - Ad-hoc rules designed by users
- The less standardized the processes (e.g. human travel organization), the higher the requirements on the Web assistance and flexibility
  $\Rightarrow$ *local behavior of nodes* and *cooperative behavior in "the Web"*

# Communication

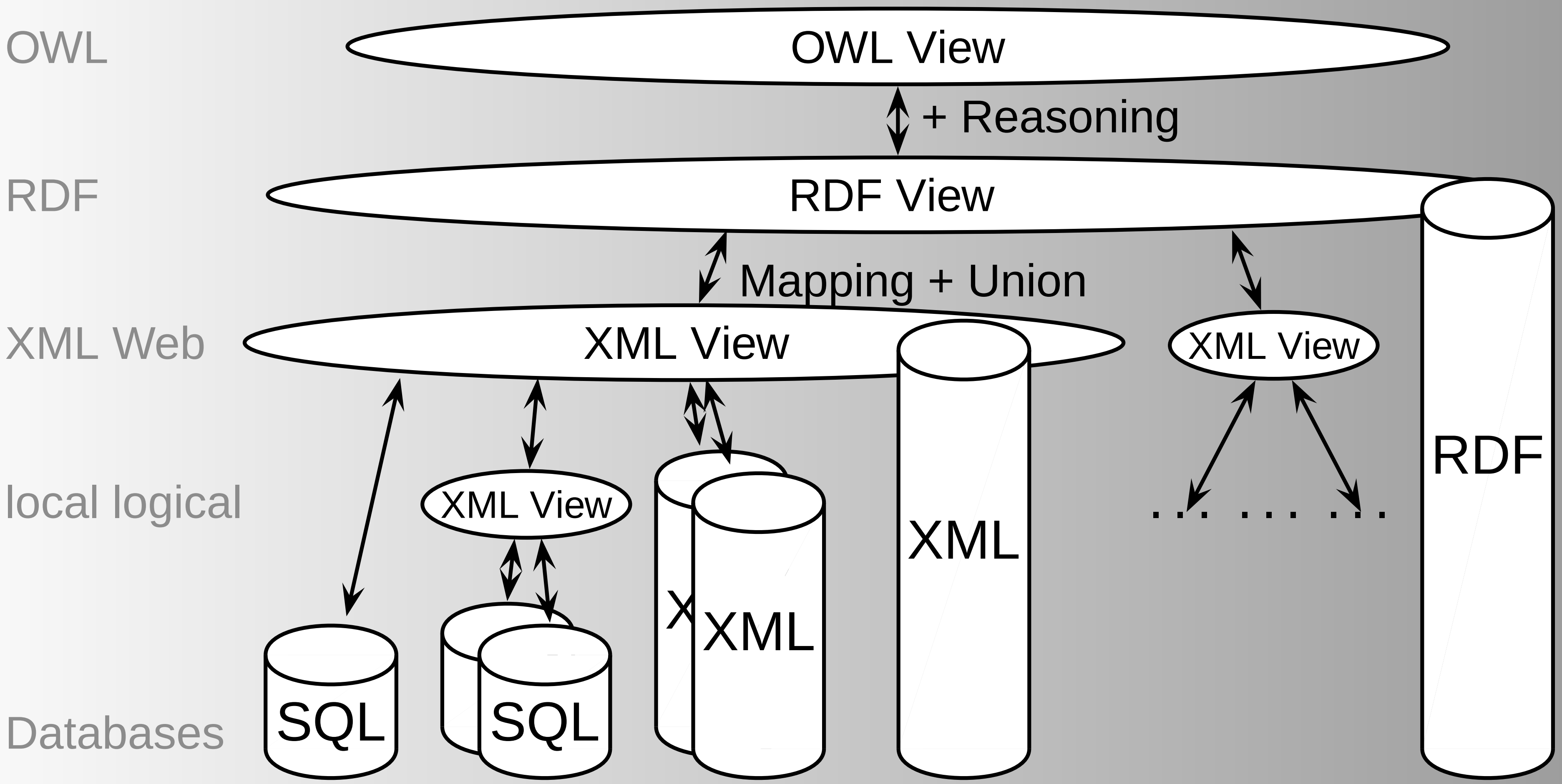
⇒ specify and implement propagation by communication/propagation strategies

# Propagation of Changes

Information dependencies induce communication paths:

- direct communication: subscribe – *push*
  based on registration; requires activity by provider
- direct communication: polling – *pull*
  regularly evaluate remote query
  – yields high load on “important” sources
  – outdated information between intervals
- + mapping into local data, view maintenance

# Abstraction Levels



# Individual Semantic Web Node

- local state, fully controlled by the node
- [optional: local behavior; see later]
- stored somehow: relational, XML, RDF databases
- local knowledge: KR model, notion of integrity, logic
  Description Logics, F-Logic, RDF/RDFS+OWL
- query/data manipulation languages:
  - database level, logical level
- mapping? – logics, languages, query rewriting, query containment, implementation
- For this *local* state, a node should *guarantee consistency*

# A Node in the Semantic Web

A Web node has not only its own data, but also “sees” other nodes:

- agreements on ontologies (application-dependent)
- agreement on languages (e.g., RDF/S, OWL)
- how to deal with inconsistencies?
  - accept them and use appropriate model/logics, reification/annotated statements (RDF), fuzzy logics, disjunctive logics
  - or try to fix them $\Rightarrow$ evolution of the Semantic Web
- tightly coupled peers: sources are known
  - predefined communication
- “open” world: e.g. travel planning

# A Node in the Semantic Web (Cont'd)

- Non-closed world
- incomplete view of a part of the Web
  - how to deal with incompleteness?
    different kinds of negation
    queries, information about events
- how to extend this view?
  - find appropriate nodes
    - information brokers, recommender systems
    - negotiation, trust
  - ontology querying and mapping
- static (model theory) vs. dynamic (query answering in restricted time; detection of changes/events)
- different kinds of logics, belief revision etc.

# Global Evolution

Semantic Web as a network of *communicating* nodes.

- Dependencies between different Web nodes,
- global Semantic Web model is an integrating view, overlapping sources $\longrightarrow$ consistency
- (the knowledge of) every node presents an excerpt of it
  - view-like with explicit reference to other sources
    + always uses the current state
    - requires permanent availability/connectivity
    - temporal overhead
  - materialize the used information
    + fast, robust, independent
    - potentially uses outdated information
  - view maintenance strategies (web-wide, distributed)

# Evolution and Behavior

Behavior is ...
... doing something

- when it is required
  - upon user interaction, a message, or a service call
  - as a reaction to an internal event (temporal, update)
  - upon some events/changes in the “world”

**Working Hypothesis**

$\Rightarrow$ use Event-Condition-Action Rules as a well-known paradigm.

# Part II: The Approach

# ECA Rules

"On Event check Condition and then do Action"

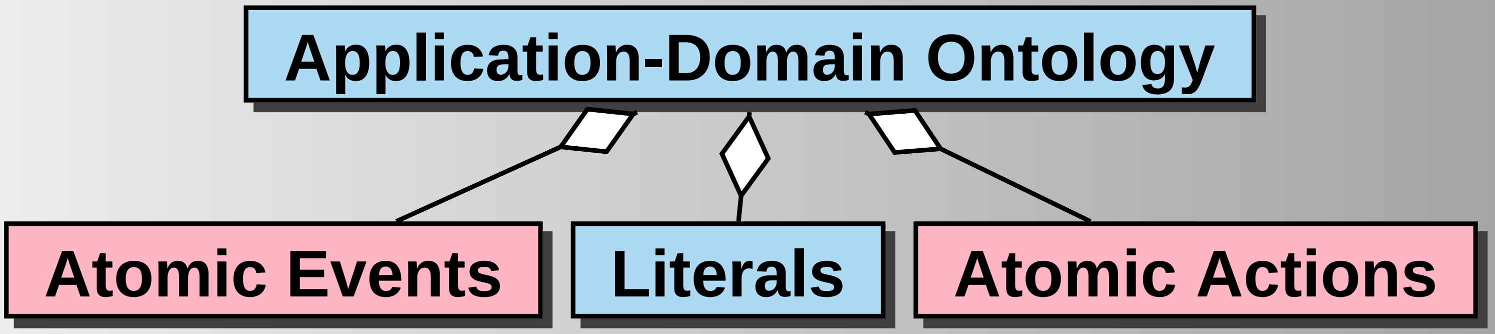
- Active Databases
- modular, declarative specification
- sublanguages for specifying *Events*, *Conditions*, *Actions*
- simple kind (database level): triggers
- high level: Business Processes, described in terms of the domain ontology

# Events and Actions in the Semantic Web

- applications do not only have an ontology that describes static notions
  - cities, airlines, flights, hotels, etc., relations between them ...
- but also an ontology of events and actions
  - cancelling a flight, cancelling a (hotel, flight) booking,
- allows for correlating actions, events, and derivation of facts
  - intensional/derived events are described in terms of actual events
    e.g., “economy class of flight X is now 50% booked”
    (derived by “if *simple event* and *condition* then (raise) *derived event*”)

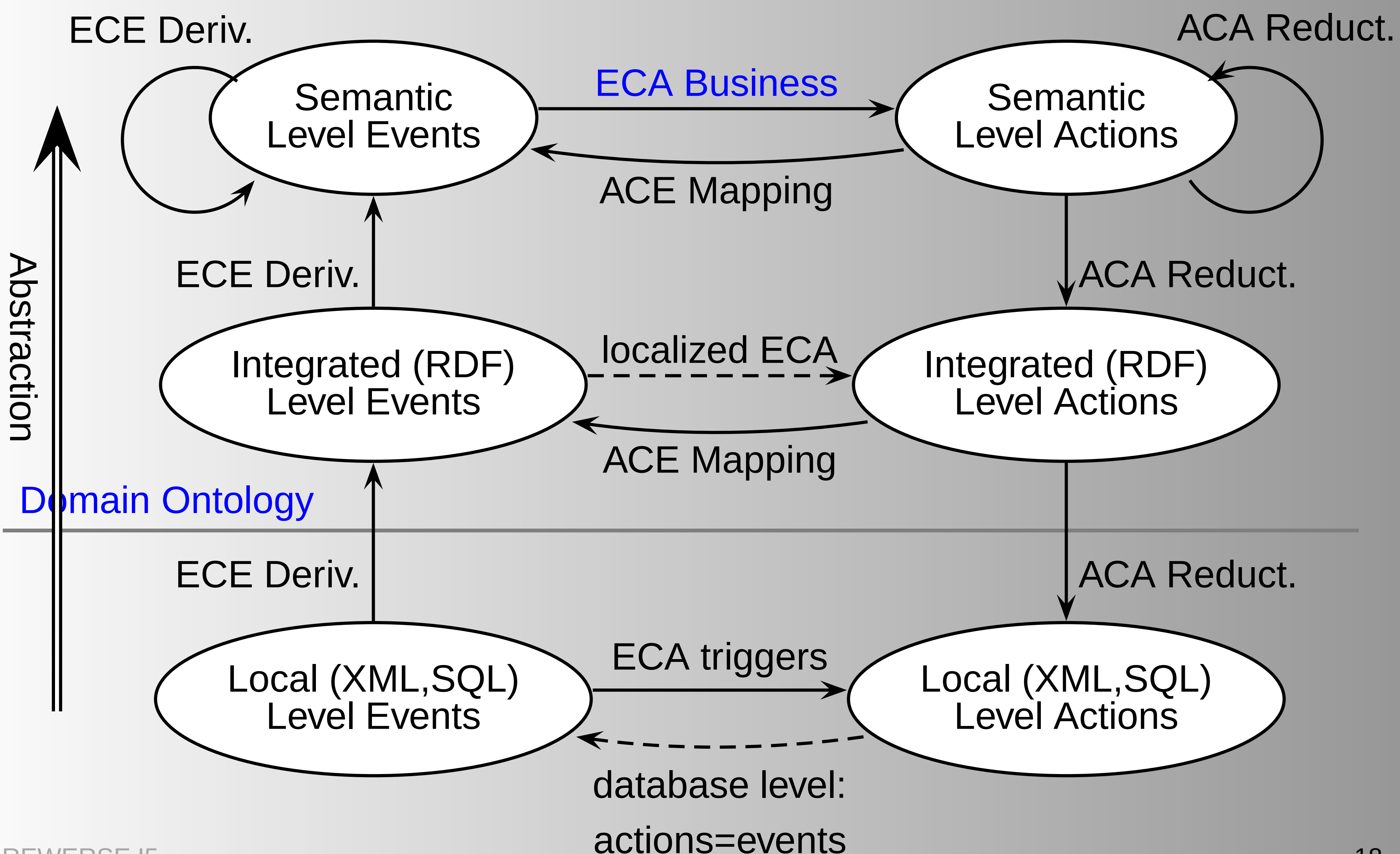
# Goals and Requirements

- Domain languages also describe behavior:




- correlate actions, events and state
- combine application-dependent semantics with generic concepts of behavior
- Ontology of behavior aspects
- [Markup]
- Operational Semantics

# Abstraction Levels and Types of Rules



# Behavior on the Web: Abstraction Levels

- OWL ontology level: *Business Processes*
- XML/RDF level:
  - cooperation and communication between closely coupled nodes on the XML Web level
  - local behavior of an application on the logical level
- database level: internal behavior (cf. SQL triggers) in terms of database items

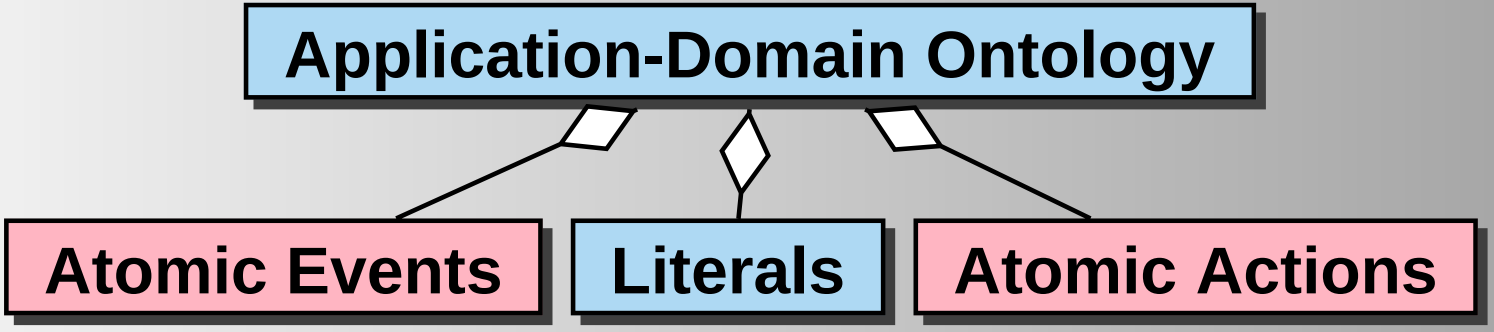
## Additional Derivation and Implementation Rules

- high-level actions are translated to lower levels
- events are derived from
  - lower-level events, same-level events
  - same-level actions

# Sources of Events

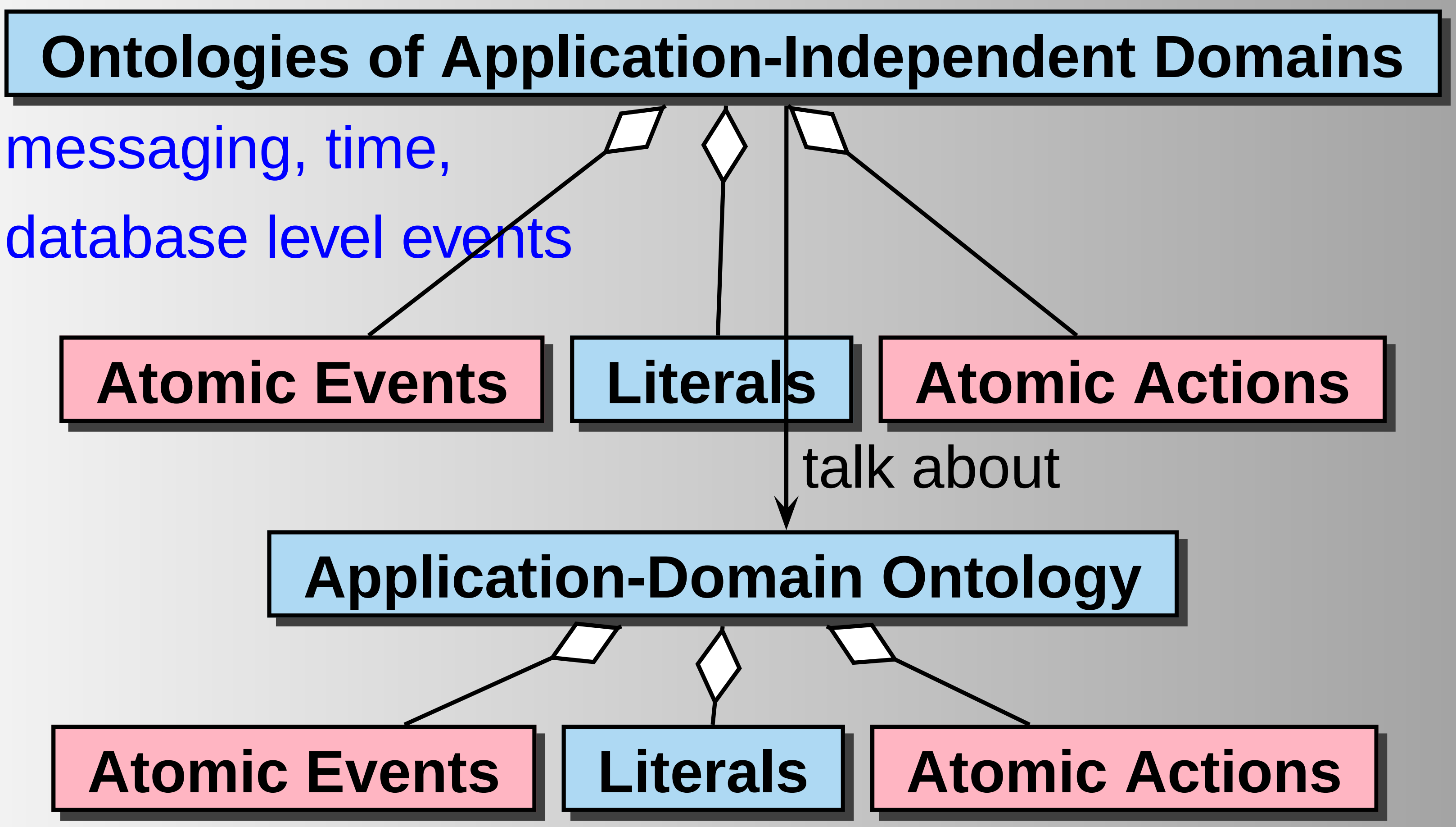
- local events: updates on the local knowledge
  - database level: updates of tuples, insertion into XML data
  - actions on the ontology level (e.g., banking:transfer(Alice, Bob, 200) or cancel-flight(LH0815))
- application-independent events: communication events, system events, temporal events

# Ontologies including Dynamic Aspects




- correlate actions, state, and events

# Ontologies including Dynamic Aspects




- correlate actions, state, and events

# Triggers on the XML Level

- similar to SQL triggers:
  ON *update* WHEN *condition* BEGIN *action* END
- *update* is an event on the XML level
  - immediately caused and identical with an update action
  - native storage: DOM Level 2/3 events
  - relational storage: must be raised/detected internally
- usually local action or ... raising a higher-level event.

# Events on the XML Level

- `ON {DELETE|INSERT|UPDATE} OF` *xsl-pattern*: operation on a node matching the *xsl-pattern*,
- `ON MODIFICATION OF` *xsl-pattern*: update anywhere in the subtree,
- `ON INSERT INTO` *xsl-pattern*: inserted (directly) into a node,
- `ON {DELETE|INSERT|UPDATE] [SIBLING [IMMEDIATELY]] {BEFORE|AFTER}` *xsl-pattern*: insertion of a sibling

⇒ extension to the local database (e.g., eXist), easy to combine with XUpdate “events”

# Sample Rule on the XML Level

- reacts on an event in the XML database
- here: maps it to an event on the RDF level
- actually an *ECE derivation rule*

```
ON INSERT OF department/professor
let $prof:= :NEW/@rdf-uri,
      $dept:= :NEW/parent::department/@rdf-uri
RAISE RDF_EVENT(INSERT OF has_professor OF department)
   with $subject:= $dept, $property:=has_professor, $object:=$prof;
```

# Triggers on the RDF Level

## Events on the RDF Level

- `ON {INSERT|DELETE|UPDATE} OF` *property* `[OF` *class*`]`.
- `ON {CREATE|UPDATE|DELETE} OF` *class*:
  if a new resource of a given class is created.

On the RDF/RDFS level, also metadata changes are events:

- `ON NEW CLASS`,
- `ON NEW PROPERTY [OF CLASS` *class*`]`

Note: these triggers on the RDF level are intended to be used for *local* behavior of an RDF database.
Higher level events must be raised (=derived) from such basic ones.

# Sample Rule on the RDF Level

- reacts on an event on the RDF view level
- here: maps it to an event on the OWL level
- again an *ECE derivation rule*

```
ON INSERT OF has_professor OF department
  % (comes with parameters $subject=dept,
  %    $property:=has_professor and $object=prof)
  % $university is a constant defined in the (local) database
RAISE EVENT
  (professor_hired($object, $subject, $university))
```

... which is then an event of the domain ontology.

# Actions, Events, Derived Events

Logical events differ from actions: an event is an observable (and volatile) consequence of an action.

- action:
  “debit 200E from Alice’s bank account”
- direct events:
  “a change of Alice’s bank account”
  “a debit of 200E from Alice’s bank account”
  “the balance of Alice’s bank account becomes below zero”
- derived events:
  “the balance of the account of a premier customer becomes belov
  “50% of all accounts at branch X are now below zero”

# Actions, Events, Derived Events

- action: “book a flight for Alice with LH0815 FRA-LIS, 20.3.2006”
- update: some changes in the Lufthansa database
- events:
  “a booking of seat 18A on flight LH0815, 20.3.2006”
  “LH0815, 20.3.2006 is fully booked”
  “there are no more tickets on 20.3. from Germany to LIS”
  - can be raised from the database updates (SQL triggers)
  - can be *derived* from the semantics of the action

# Global and Remote Events

Events are caused by updates to a certain Web source
Applications are not actually interested where this happens

global application-level events “somewhere in the Web”

- “on change of VAT do ...”
- “if a flight is offered from FRA to LIS under 100E”

⇒ requires detection/communication strategies

... so far to the analysis of events and actions.
Let’s continue with the rules.

# Analysis of Rule Components

... have a look at the clean concepts:

"On Event check Condition and then do Action"

- Event: specifies a rough restriction on what *dynamic* situation probably something has to be done.
  Collects some parameters of the events.
- Condition: specifies a more detailed condition, including *static* data if actually something has to be done.
  $\Rightarrow$ evaluate a ((Semantic) Web) query.
- Action: actually *does* something.

## Example

"if a flight is offered from FRA to LIS under 100E and I have no lectures these days then do ..."

# SQL Triggers

```
ON {DELETE|UPDATE|INSERT} ...
WHEN where-style condition
BEGIN
  // imperative code that contains
  // - SQL-queries into PL/SQL variables
  // - if ... then ...
END;
```

- only very simple events (atomic updates)
- WHEN part can only access information from the event
- large parts of evaluating the condition actually happen in the non-declarative PL/SQL program part
  $\Rightarrow$ no reasoning possible!

# A More Detailed View of ECA

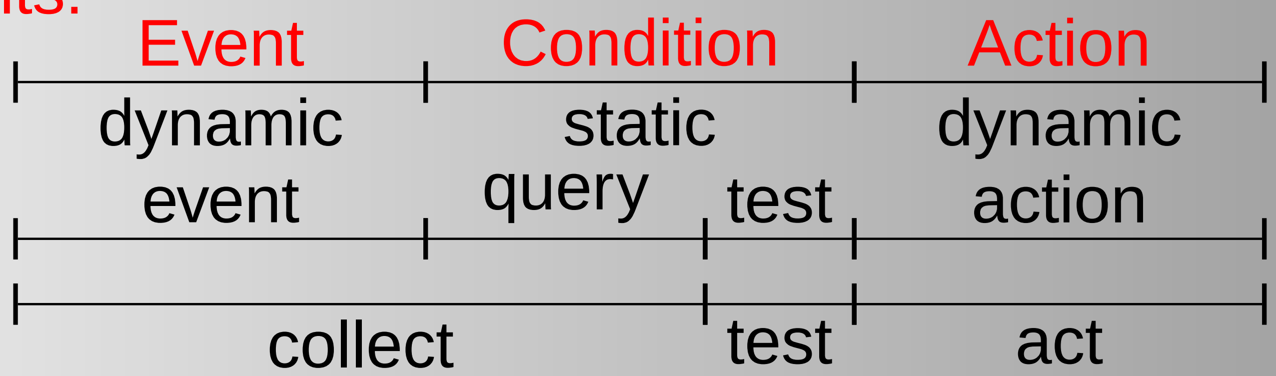
- the event should just be the dynamic component
- “if a flight is offered from FRA to LIS under 100E and I have no lectures these days then do ...”
  - “100E” is probably contained in the event data (insertion of a flight)
  - my lectures are surely not contained there

$\Rightarrow$ includes another query before evaluating a condition
SQL: would be in an `select ... into ...` and `if` in the action part.

# Clean, Declarative "Normal Form"

"On Event check Condition and then do Action"

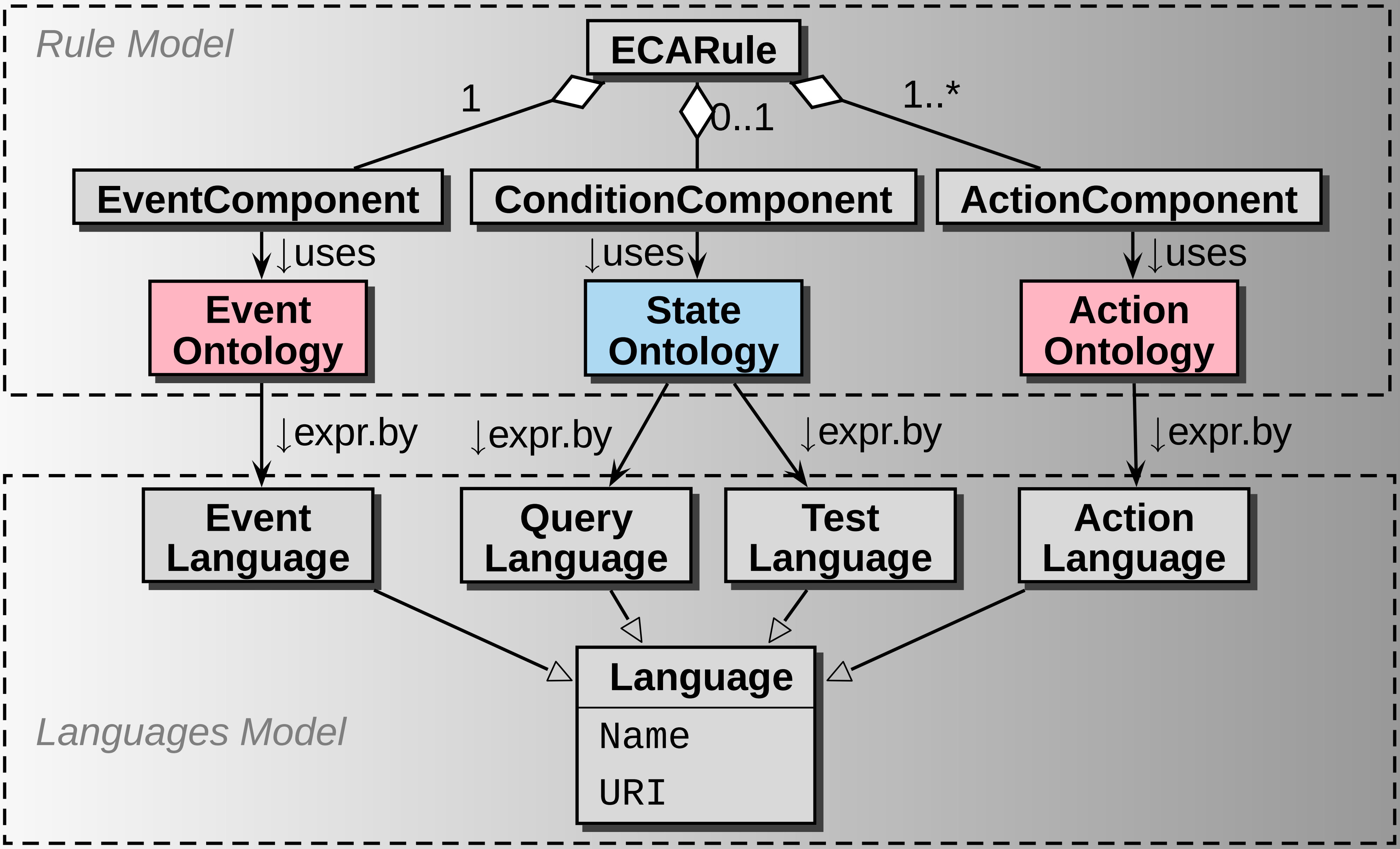
Rule Components:




- Event: detect just the dynamic part of a situation,
- Query: then obtain additional information by queries,
- Test: then evaluate a *boolean* condition,
- Action: then actually do something.

- Component sublanguages: heterogeneous

# Modular ECA Concept: Rule Ontology



# 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**>

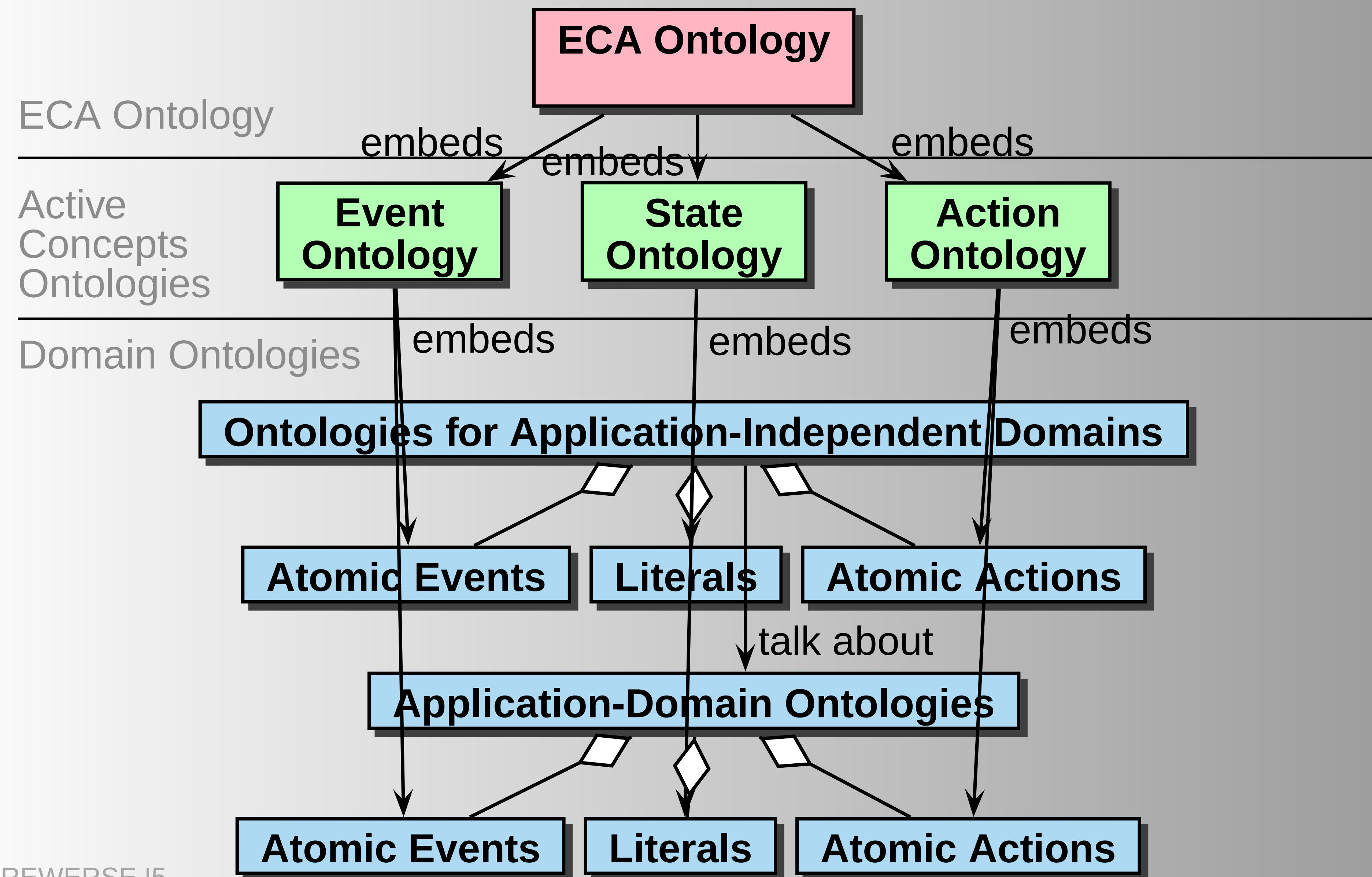
# Example

Sample Event:

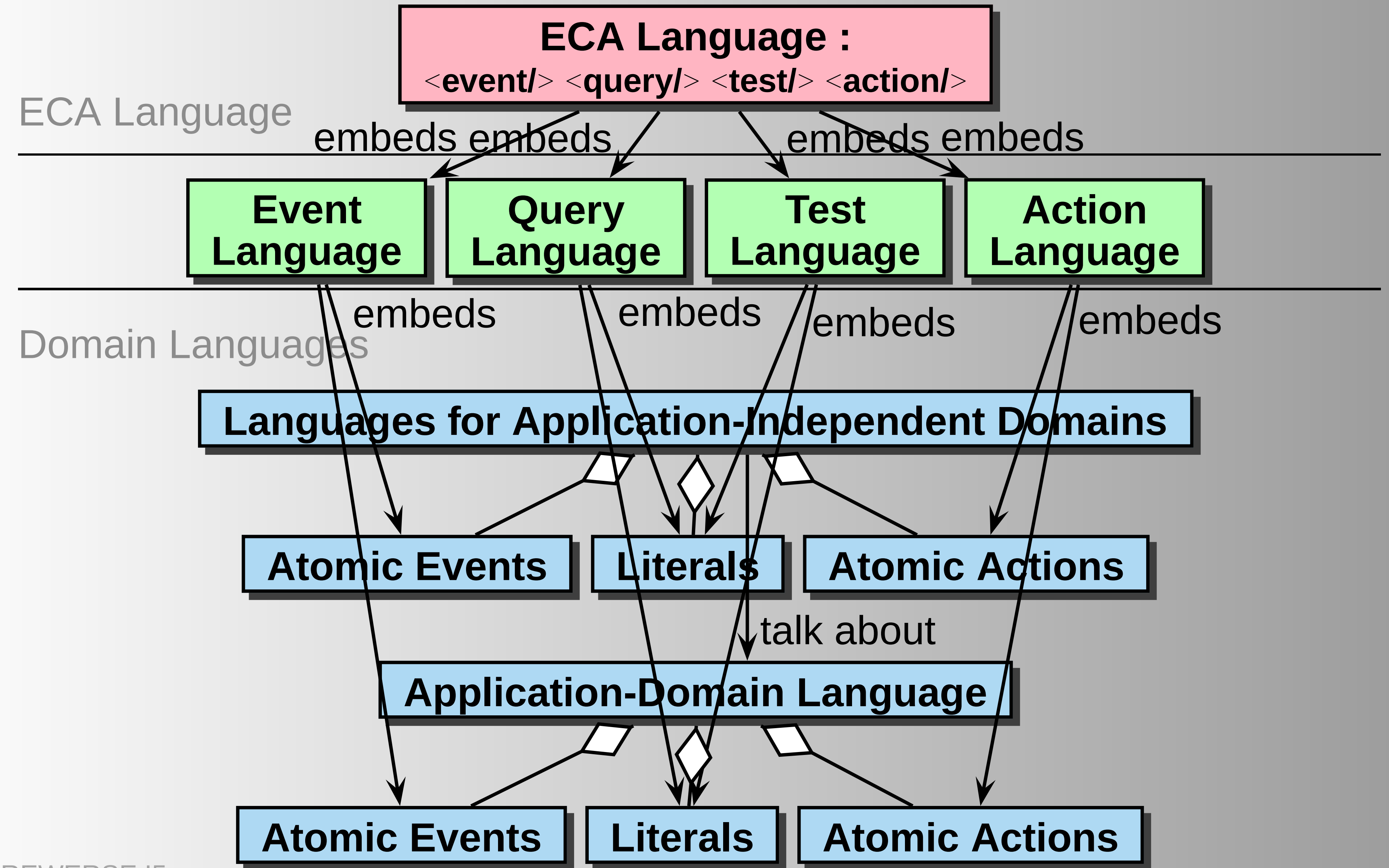
```
<travel:cancel-flight flight="LH123" >
    <travel:reason>bad weather</travel:reason>
</travel:cancel-flight >
```

```
<eca:rule>
  <eca:event xmlns:travel="www.travel.com" >
    <eca:atomic-event>
      <travel:cancel-flight flight=" $flight "/>
    <eca:atomic-event>
  </eca:event>
  <eca:query> . . .  </eca:query>
  <eca:test> . . .  </eca:test>
  <eca:action > do something with $flight </eca:action>
</eca:rule>
```

# Combination of Ontologies



# Embedding of Languages



# Active Concepts Ontologies

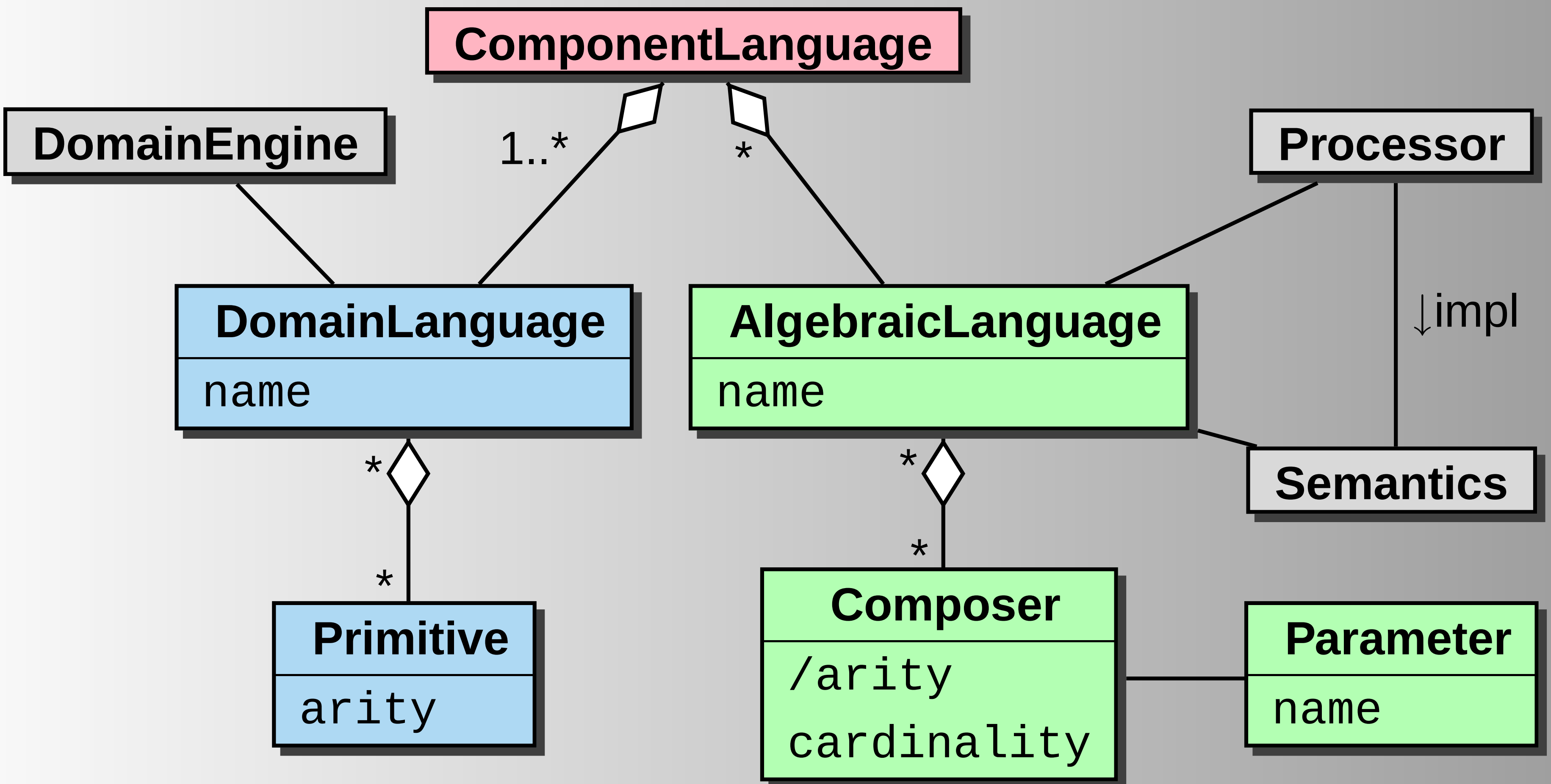
- Domains specify atomic events, actions and static concepts

## Composite [Algebraic] Active Concepts

- Event algebras: composite events
  - (when) $E_1$ and some time afterwards $E_2$ (then do $A$)
  - (when) $E_1$ happened and then $E_2$, but not $E_3$ after at least 10 minutes (then do $A$)
  - well-investigated in Active Databases (e.g. SNOOP).
- Process algebras (e.g. CCS)

$\Rightarrow$ See concepts defined by these *formal methods* as defining *ontologies*.

# Algebraic Sublanguages

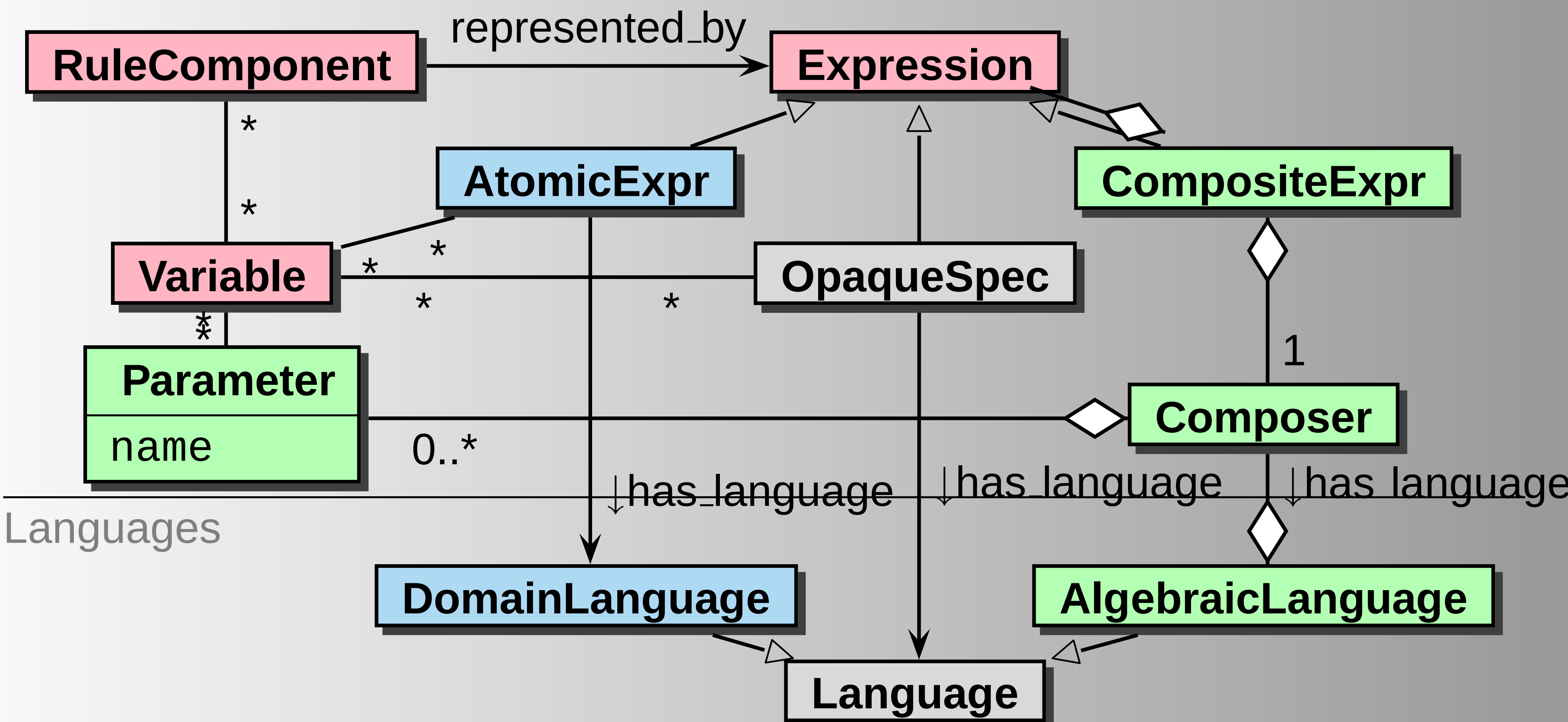


# Opaque Components

Compatibility with current Web standards:

- current (query) languages do in general not use markup, but program code
- allow *opaque* components:
  - query component: XQuery, XPath, SQL
  - action component: updates in XQuery, XUpdate, SQL

# Syntactical Structure of Expressions

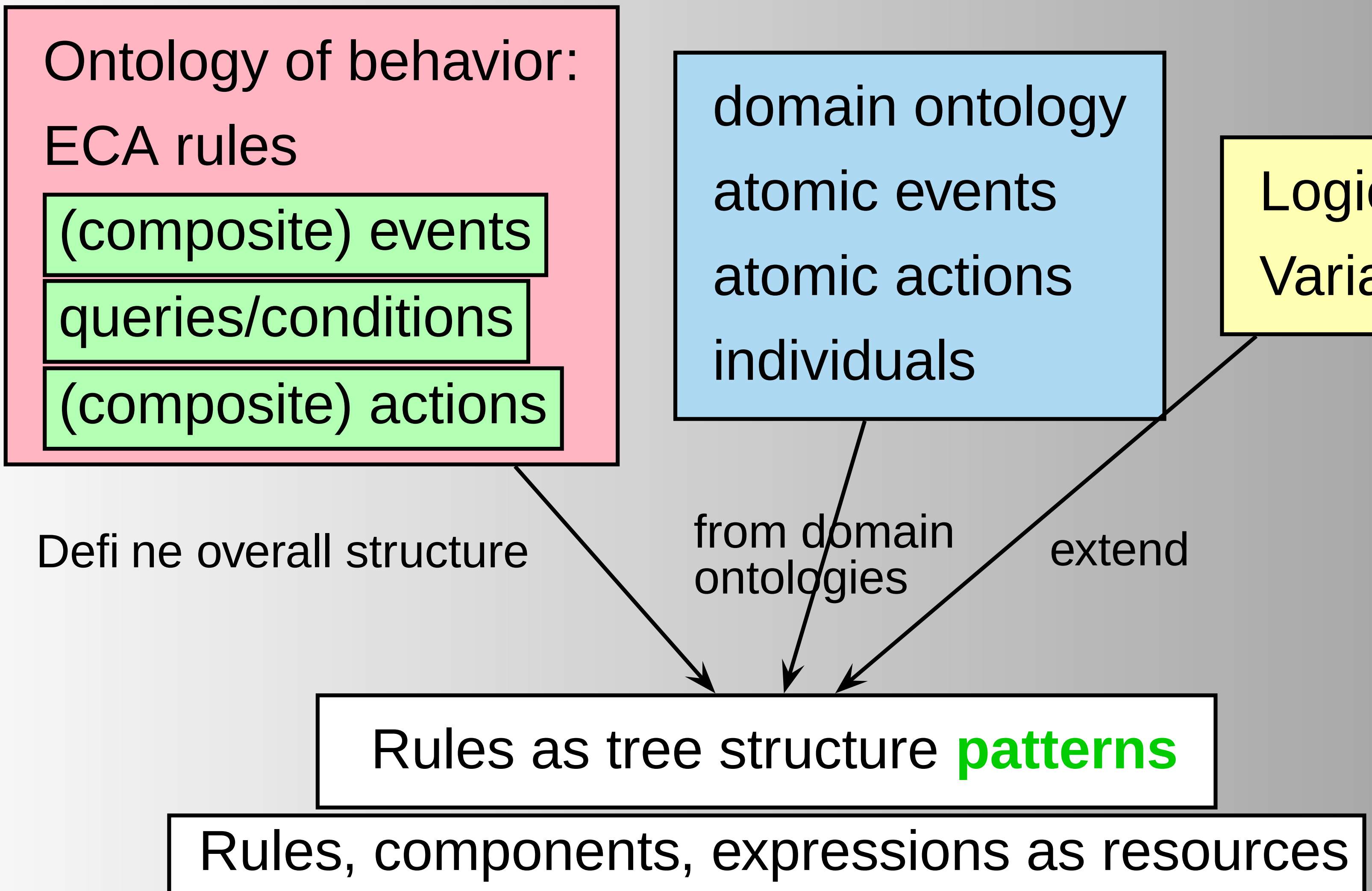



- as operator trees: “standard” XML markup of terms
- RDF markup as languages
- every expression can be associated with its language

# Subconcepts and Sublanguages

- different languages, different expressiveness/complexity
- common structure: algebraic languages
- e/q/t/a subelements contain a language identification, and appropriate contents
- embedding of languages according to language hierarchy:
  - algebraic languages have a natural term markup.
  - every such language “lives” in an own namespace,
  - domain languages also have an own namespace,
- information flow between components by variables,
- (sub)terms must have a well-defined result.

# ECA Rule Markup



# Rule Semantics/Logical Variables

Deductive Rules: $head(X_1, \ldots, X_n) :- body(X_1, \ldots, X_n)$

- bind variables in the body
- obtain a set of tuples of variable bindings
- “communicate” them to the head
- instantiate/execute head for each tuple

# Rule Semantics/Logical Variables

Deductive Rules: $head(X_1,\ldots,X_n) :- body(X_1,\ldots,X_n)$

- bind variables in the body
- instantiate/execute head for each tuple

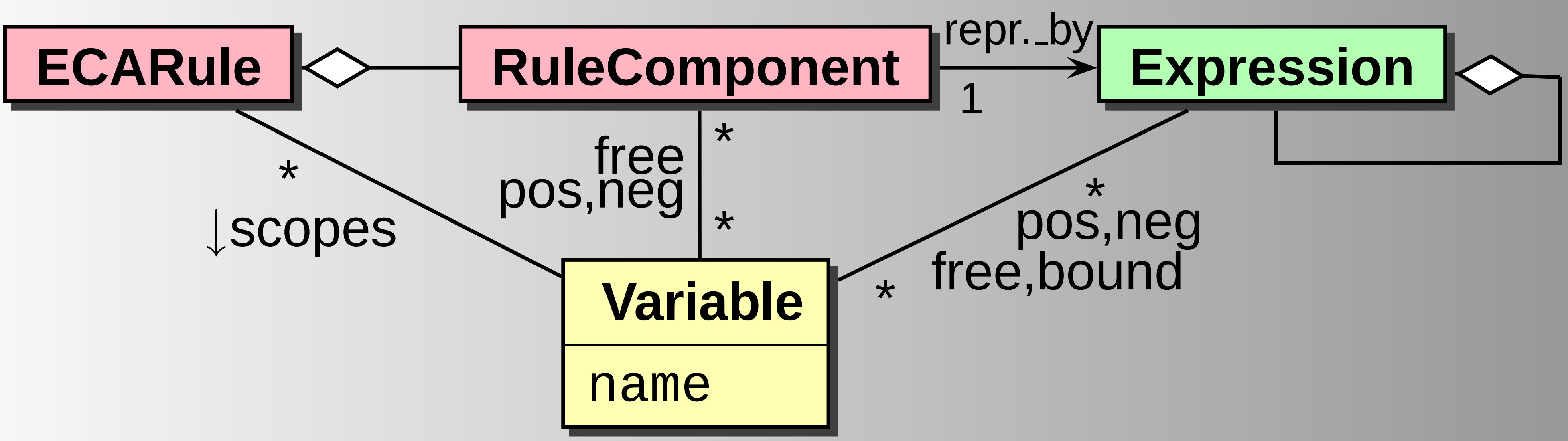
ECA Rules

- initial bindings from the event
- additional bindings from queries
- restrict by the test
- execute action for each tuple

$$action(X_1,\ldots,X_n) :-$$
$$event(X_1,\ldots,X_k),\ query(X_1,\ldots,X_k,\ldots X_n),\ test(X_1,\ldots,X_n)$$

# Rule Semantics

- Deductive rules: variable bindings Body$\rightarrow$Head
- communication/propagation of information by *logical variables*:
  $E \xrightarrow{+} Q \rightarrow T$ & A
- safety as usual (extended with technical details ...)



# Binding and Use of Variables

- Variables can be bound to values, XML fragments, RDF fragments, and (composite) events
- Logic Programming (Datalog, F-Logic): variables occur free in patterns.
  Markup uses XSLT-style
  <variable name="*var-name*">*language-expr* </variable>
  and $*var-name*
  inside component expressions.
- functional style (event algebras, SQL, OQL, XQuery): expressions return a value/fragment.
  ⇒ must be bound to a variable to be kept and reused.
  <variable name="*var-name*">*language-expr* </variable>
  on the rule level around a component expression.

# Operational Semantics of Rules

- Event: fires the rule
  - returns the sequence that matched the event
  - optional: variable bindings
- Query: obtain additional static information
  - returns the answer/set of answers
  - optional: for each answer, restrict/extend variable bindings (join semantics)
- Condition:
  - check a boolean condition, constrain variable bindings
- Action:
  - do something by using the variable bindings.

# Rule Markup: Example (Stripped)

```
<!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:delay-flight flight=" $flight "/> <travel:cancel-flight flight=" $flight "/>
    </snoop:seq>
  </eca:event>
  <eca:query>
    <eca:variable name="email">
      <eca:opaque lang="http://www.w3.org/xpath">
        doc("www.lufthansa.de")/flights[code="$flight "]/passenger/@e-mail
      </eca:opaque> </eca:variable> </eca:query>
  <eca:action xmlns:smtp="...">
    <smtp:send-mail to="$email" text="..."/>
  </eca:action>
</eca:rule>
```

# Event Algebras

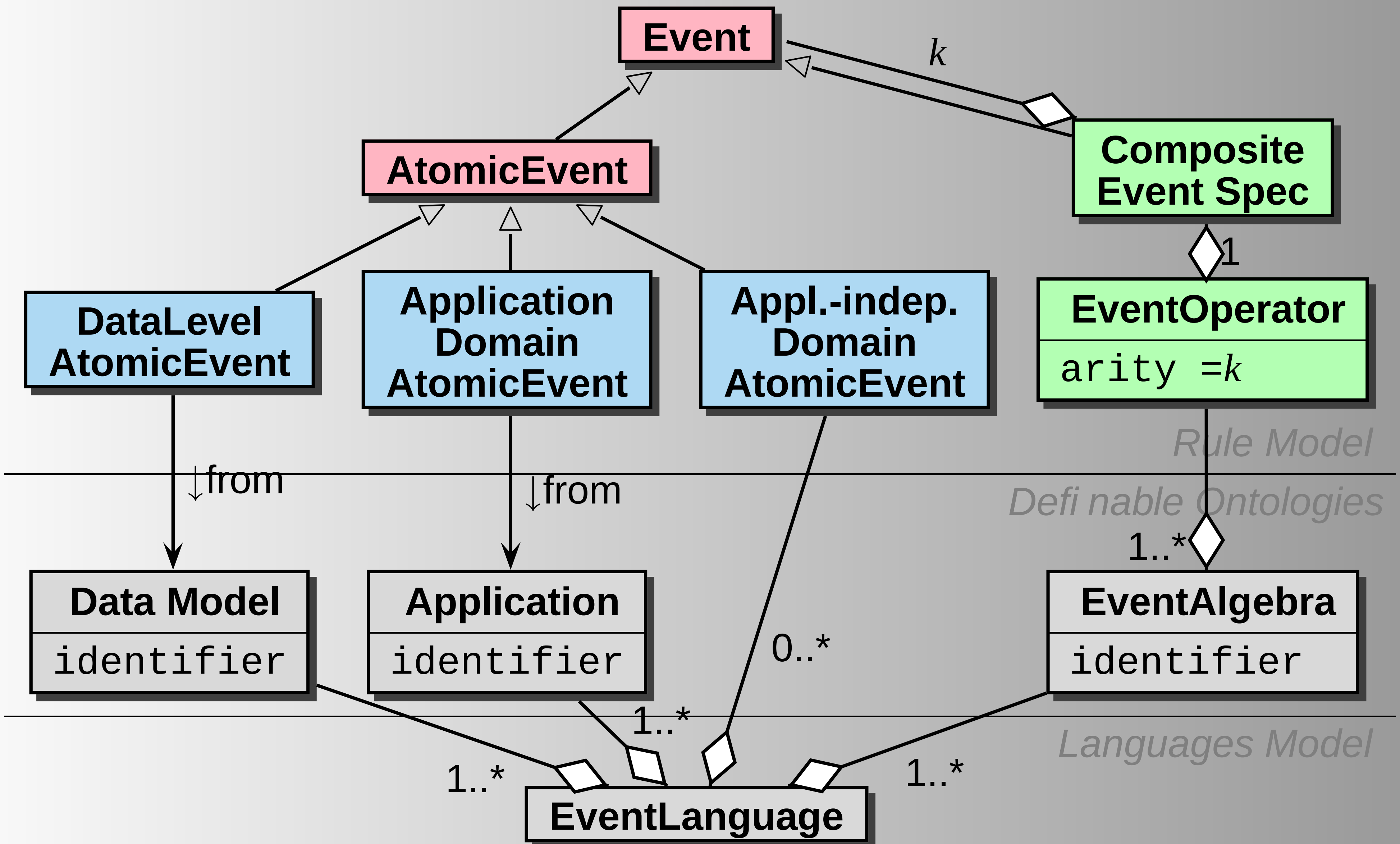
... up to now: only simple events.
Atomic events can be combined to form composite events. E.g.:

- (when) $E_1$ and some time afterwards $E_2$ (then do $A$)
- (when) $E_1$ happened and then $E_2$, but not $E_3$ after at least 10 minutes (then do $A$)

*Event Algebras* allow for the definition of composite events.

- specifying composite events as terms over atomic events.
- well-investigated in Active Databases
  (e.g., the SNOOP event algebra of the SENTINEL ADBMS)

# Events Subontology



# Sample Markup (Event Component)

```
<eca:rule xmlns:travel="...">
  <eca:variable name="theSeq">
    <eca:event xmlns:snoop="...">
    <snoop:sequence>
        <eca:atomic-event>
          <travel:delay-flight flight="$Flight" minutes="$Minutes"/>
        </eca:atomic-event>
        <eca:atomic-event>
          <travel:cancel-flight flight="$Flight"/>
        </eca: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

# Example: Travel Domain

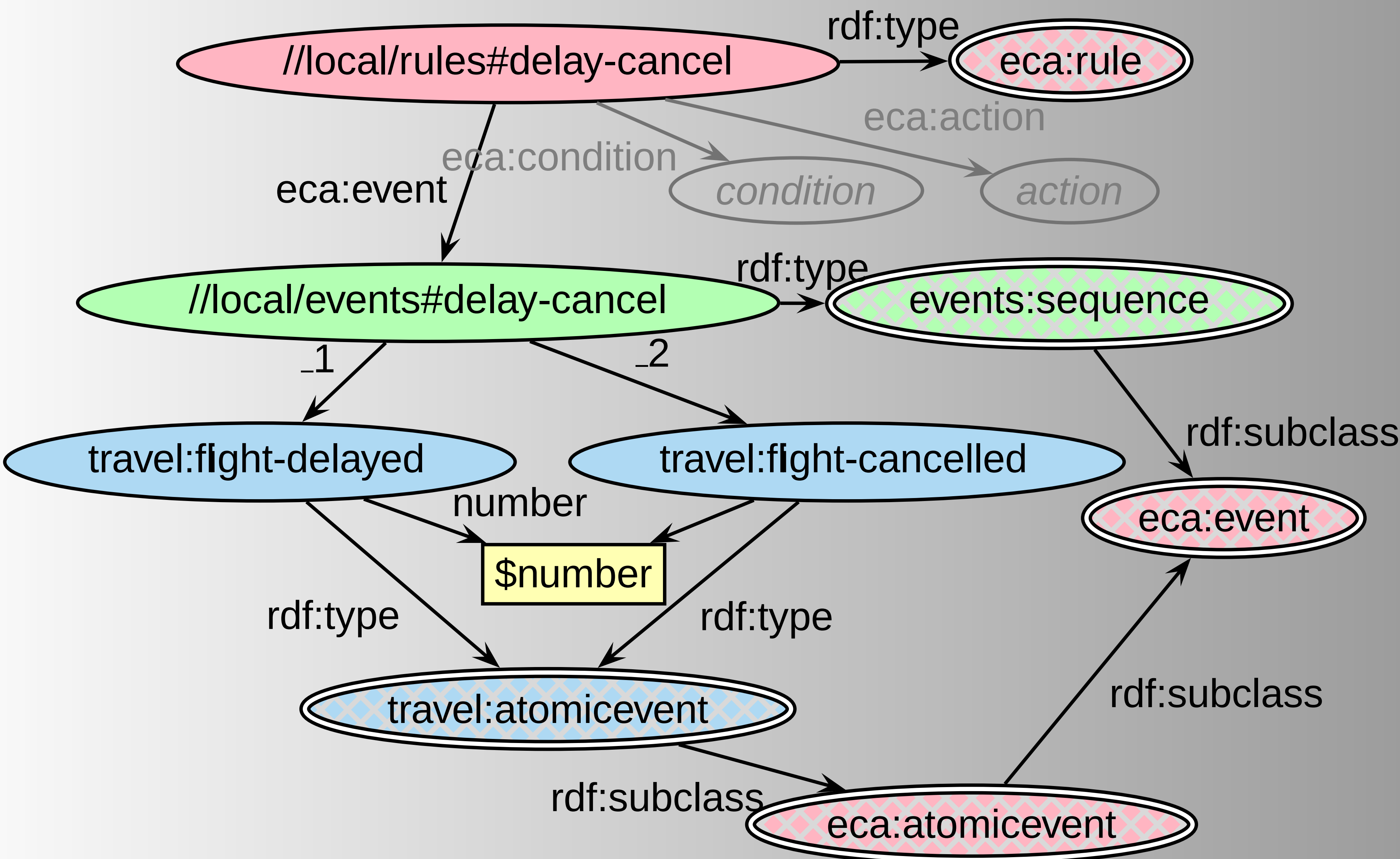
## Portal functionality:

- static: access to train/flight schedules, hotels etc.
- dynamic: communication of events and announcements like delayed or cancelled flights
- users: can register rules for their personal needs

## Example:

- “if a flight is first delayed and then cancelled, then ...”
  $\Rightarrow$ composite sequential event.

# Example


rdf:type
//local/rules#delay-cancel
eca:rule
eca:action
eca:condition
condition
action
eca:event
rdf:type
//local/events#delay-cancel
events:sequence
_1
_2
travel:flight-delayed
travel:flight-cancelled
rdf:subclass
number
eca:event
$number
rdf:type
rdf:type
travel:atomicevent
rdf:subclass
rdf:subclass
eca:atomicevent

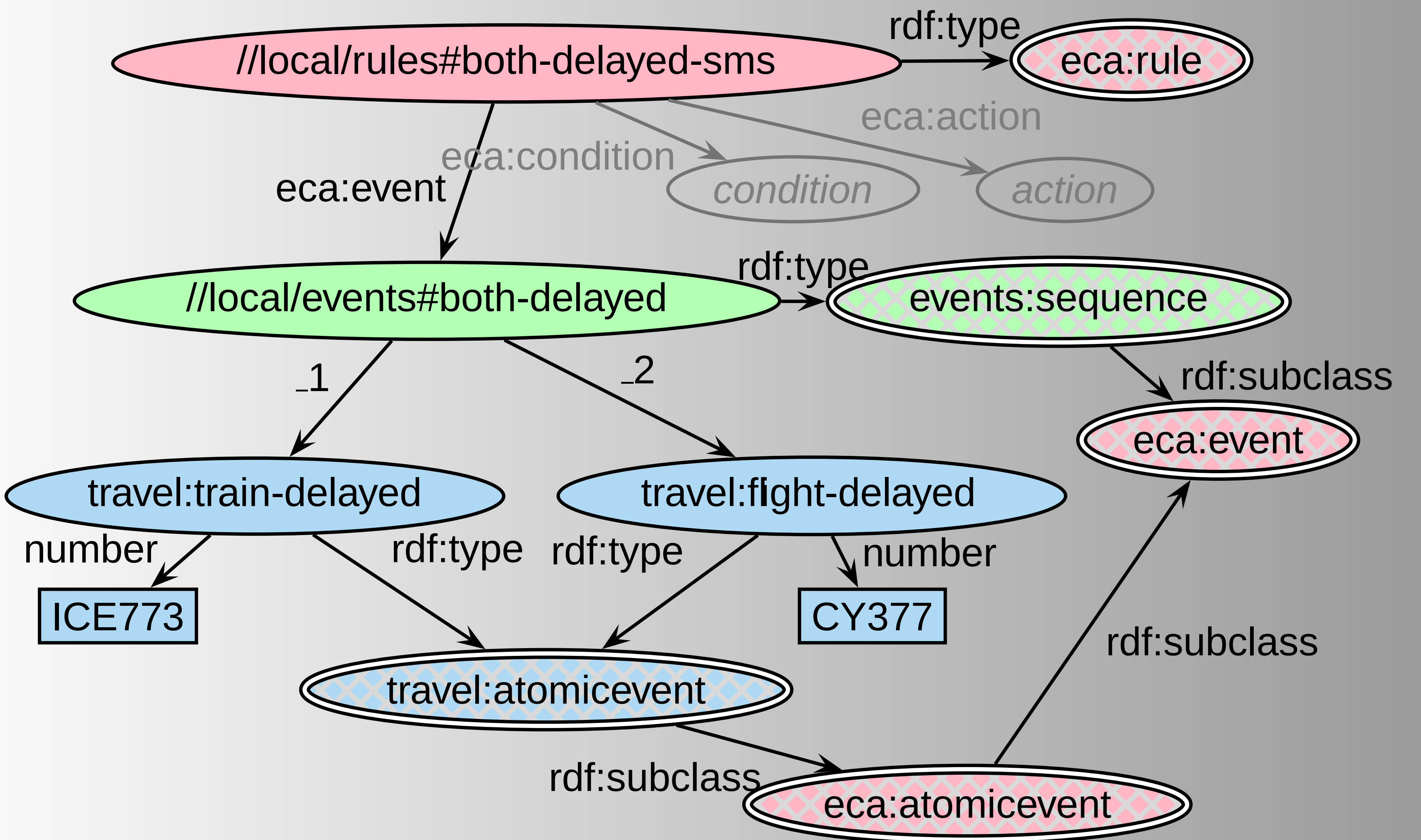
# Example: Travel Domain

## Portal functionality:

- static: access to train/flight schedules, hotels etc.
- dynamic: communication of events and announcements like delayed or cancelled flights
- users: can register rules for their personal needs

1. "if my train is delayed, send me an SMS".
   Simple, often not necessary – no danger.
2. "if my flight is delayed ... don't send me a mail"
   (same as above).
3. But: "if my train towards the airport is delayed, and the flight is also delayed, then send me an SMS (about the latter)"
   $\Rightarrow$ composite sequential event.

# Example



rdf:type
//local/rules#both-delayed-sms
eca:rule
eca:action
eca:condition
eca:event
condition
action
rdf:type
//local/events#both-delayed
events:sequence
_1
_2
rdf:subclass
eca:event
travel:train-delayed
travel:flight-delayed
number
rdf:type
rdf:type
number
ICE773
CY377
rdf:subclass
travel:atomicevent
rdf:subclass
eca:atomicevent

# Ontologies, Languages and Resources

- Rule components, subexpressions etc. are resources
- associated with languages corresponding to the ontologies (event languages, action languages, domain languages)
- each language is a resource, identified by a URI.
- DTD/XML Schema/RDF description of the language
- Algebraic languages:

  processing engine
- Domain Languages:

  Event Broker Services (subscribe) and processors for actions

# Detection of Atomic Events

- Atomic Data Level Events [database system ontology; local]
- Appl.-indep. Domain Events
  - receive message [common ontology; local]
    with contents [contents: own ontology] as parameter
  - transactional events [common ontology; local]
  - temporal events [common ontology]
    provided by services (upon registration)
- Application-Level Events [domain ontology]
  - derived/raised by appropriate ECE/ACE rules,
    (probably also derived from other facts)
- Composite Events: event detection algorithm; feeded with atomic events

# Event Component: Event Algebras

- a composite event is detected when its "final" subevent is detected:

$$(E_1 \nabla E_2)(x,t) \quad :\Leftrightarrow \quad E_1(x,t) \vee E_2(x,t) \ ,$$

$$(E_1;E_2)(x,y,t) \quad :\Leftrightarrow \quad \exists t_1 \leq t : E_1(x,t_1) \wedge E_2(y,t)$$

$\neg(E_2)[E_1,E_3](t)$ $:\Leftrightarrow$ if $E_1$ and then a first $E_3$ occurs, without occurring $E_2$ in between.

- "join" variables between atomic events
- "safety" conditions similar to Logic Programming rules
- Result:
  - the sequence that matched the event
  - optional: additional variable bindings

# Advanced Operators (Example: SNOOP)

- ANY$(m, E_1, \ldots, E_n)(t) \quad :\Leftrightarrow$
  $\exists t_1 \leq \ldots \leq t_{m-1} \leq t,\ 1 \leq i_1, \ldots, i_m \leq n$ pairwise
  distinct s.t. $E_{i_j}(t_j)$ for $1 \leq j < m$ and $E_{i_m}(t)$ ,
- "aperiodic event"
  $\mathcal{A}(E_1, E_2, E_3)(t) \ :\Leftrightarrow$ ,
  $E_2(t) \wedge (\exists t_1 : E_1(t_1) \wedge (\forall t_2 : t_1 \leq t_2 < t : \neg E_3(t_2)))$
  "after occurrence of $E_1$, report *each* $E_2$, until $E_3$ occurs"
- "Cumulative aperiodic event":
  $\mathcal{A}^*(E_1, E_2, E_3)(t) \ :\Leftrightarrow \ \exists t_1 \leq t : E_1(t_1) \wedge E_3(t)$
  "if $E_1$ occurs, then for each occurrence of an instance of $E_2$, collect its parameters and when $E_3$ occurs, report all collected parameters".
  (Same as before, but now only reporting at the end)

# Examples of Composite Events

- A deposit (resp. debit) of amount $V$ to account $A$:
  $E_1(A,V) := deposit(A,V)$ (resp. $E_2(A,V) := debit(A,V)$)
- A change in account $A$: $E_3 := E_1(A,V) \nabla E_2(A,V)$.
- The balance of account $A$ goes below 0 due to a debit:
  $E_4(A) := debit(A,V) \wedge balance(A) < 0$
  [note: not a clean way: includes a simple condition]
- A deposit followed by a debit in Bob's account:
  $E_5 := E_1(bob,V_1); E_2(bob,V_2)$.
- There were no deposits to an account $A$ for 100 days:
  $E_6(A) := (\ \neg(\exists X : deposit(A,X)))$

  $$[deposit(A,Am) \wedge t = date; date = t + 100 days]$$

# Examples of Composite Events (Cont’d)

- The balance of account $A$ goes negative and there is another debit without any deposit in-between:
  $E_7 := \mathcal{A}(E_4(A), E_2(A, V_1), E_1(A, V_2))$
- After the end of the month send an account statement with all entries:
  $E_8(A, list) := \mathcal{A}^*(first\_of\_month, E_3(A), first\_of\_next\_month)$

# Query Component

... obtain additional information:

- local, distributed, OWL-level
- Result:
  - the answer to the query
    XQuery, XPath, SQL
  - bindings of free variables
    Datalog, F-Logic, XPathLog, SparQL

# Test Component

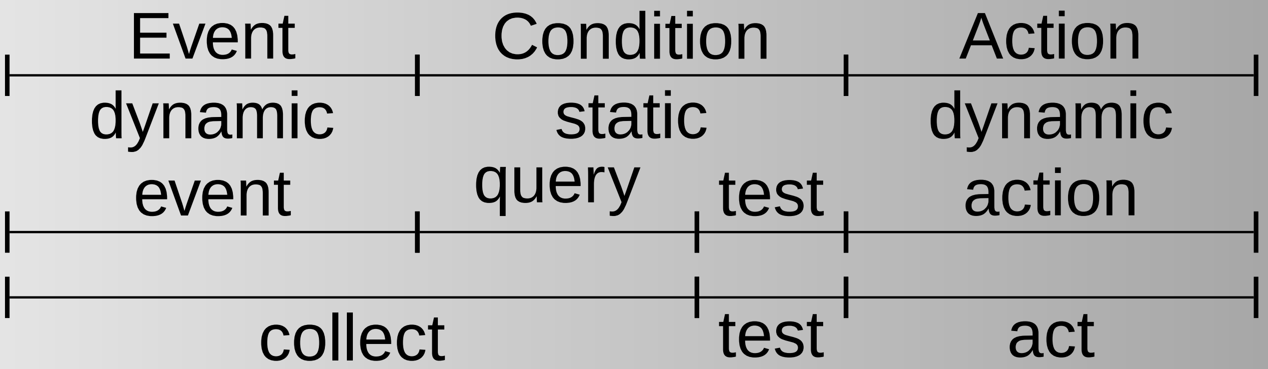
- evaluate (locally) a test over the collected information

# The Action Component

- Atomic actions:
  - ontology-level local actions
  - data model level updates of the local state
  - explicit calls of remote procedures/services
  - explicit sending of messages
  - ontology-level *intensional* actions (e.g. in *business processes*)
- Composite actions: e.g. CCS or (opaque) code
- transactions
  including queries against other sources

# Part III: The Architecture

# ECA Rules




- each ECA Rule language uses
  - a (composite) event language (mostly an event algebra)
  - a query language
  - a condition language
  - a language for specification of actions/transactions
- different languages, different expressiveness/complexity
- different locations where the evaluation takes place

$\Rightarrow$ Modular concepts with Web-wide services

# Engines – Service-Based Architecture

Language Processors as Web Services:

- ECA Rule Execution Engine employs other services for E/Q/T/A parts:
  nodes register their rules at the engines; processing is done by the engine
- dedicated services for each of the event/action languages e.g., composite event detection engines
- dedicated services for domain-specific issues:
  raising and communicating events, predicates, executing actions/updates
- query languages often implemented directly by the Web nodes (portals and data sources)

# Languages and Resources

Each language is a resource, identified by a URI.
Connected to the following resources:

## ECA and Generic Sublanguages

- DTD/XML Schema/RDF description of the language
- processing engine (according to a communication interface)
- [semantics description by a formal method for reasoning about it]

## Application Languages/Ontologies

- DTD/XML Schema/RDF description of the language
- Event Broker Services (subscribe)

# Communication

- register “things” (rules, events) at appropriate services
- communicate relevant events

⇒ different strategies

a) user/client registers rule and also provides relevant events
   service only implements the algorithms

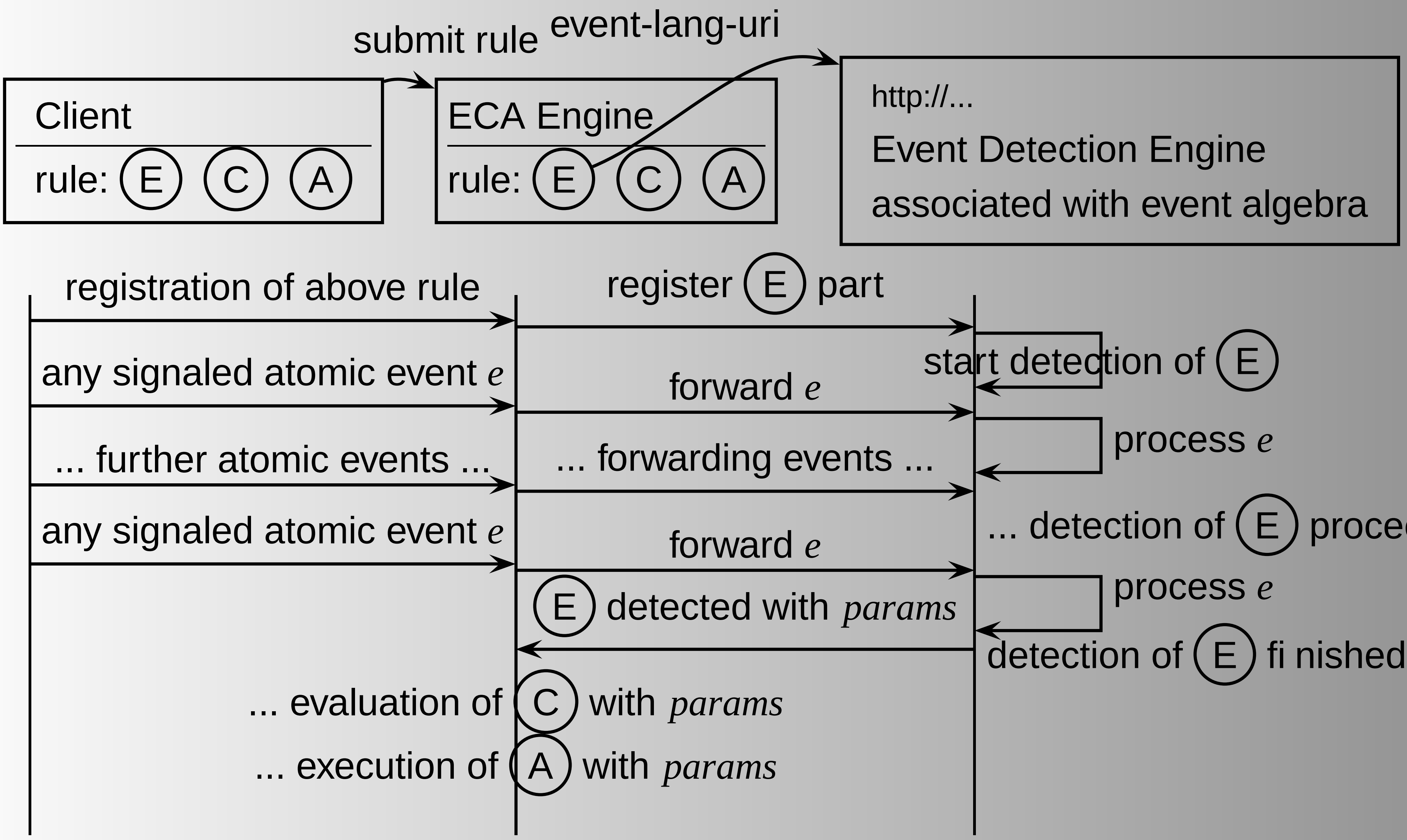
b) user/client registers rule and leaves the acquisition of events to the service:
   - event language/ontology: service
   - atomic events: provided by application-specific services

# Communication: Simple Pattern

Only the algorithmic part is outsourced:

- nodes that register an *ECA rule* at a service must forward all relevant events to the Rule Evaluation Service
- service that registers a *composite event specifi cation* at a service must forward events to the Event Detection Service

# Architecture



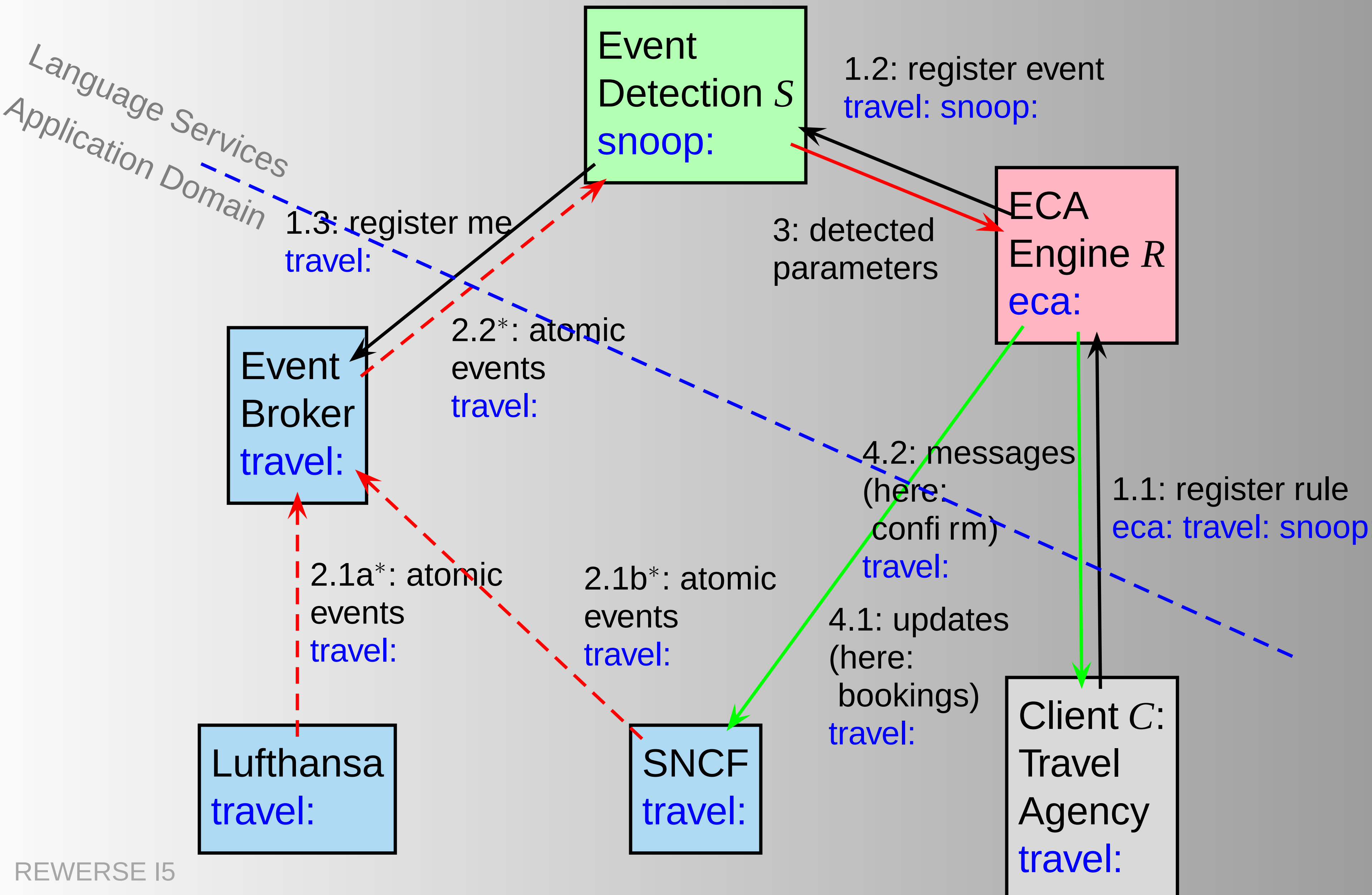
# Advanced Architecture

Complete event detection is outsourced:

- composite event detection service is also responsible for detecting appropriate atomic events
(e.g., specialized on a certain application area)

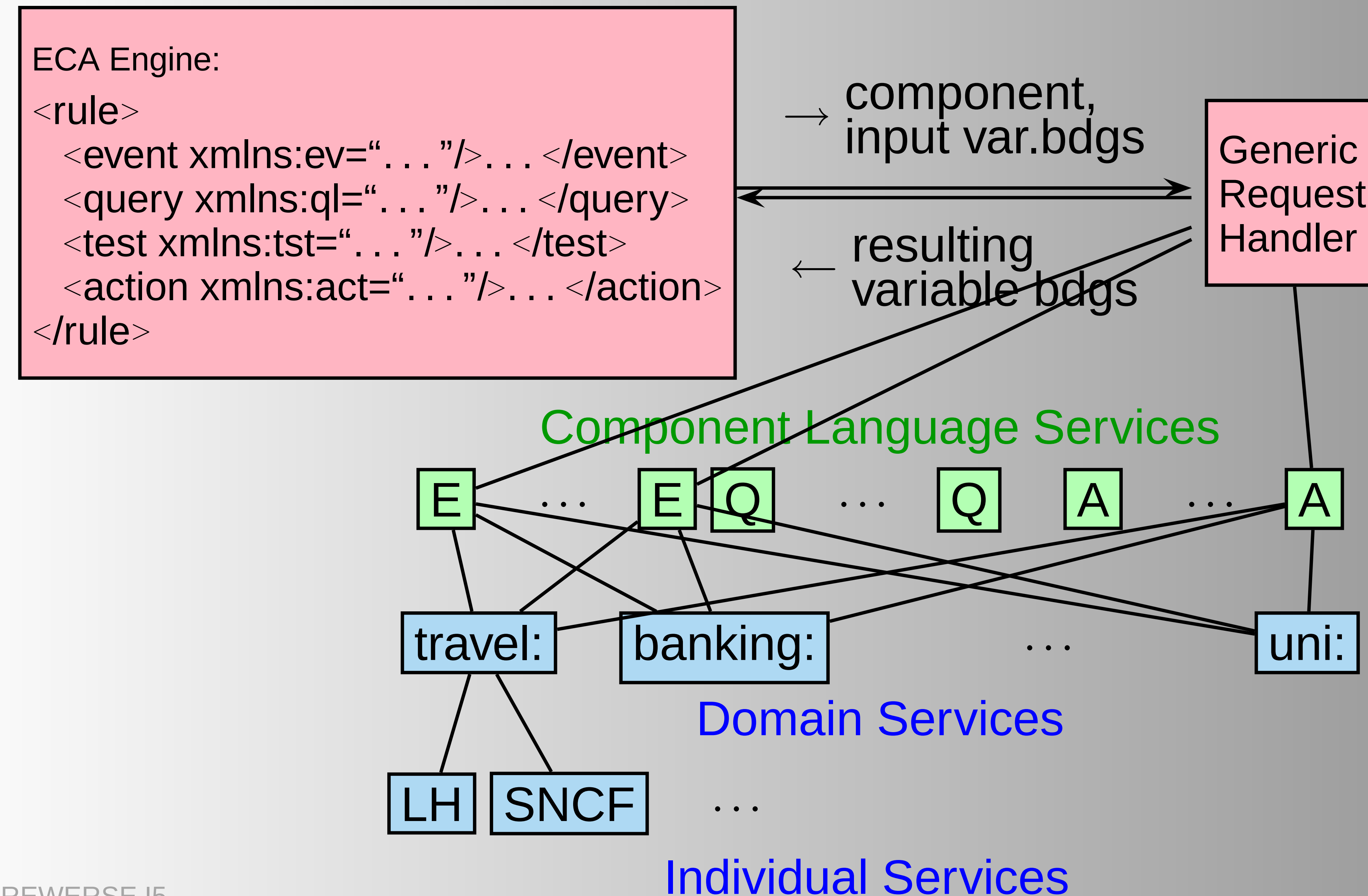
- Rule Execution Services
- Event Broker Services (application-specific)
- Algorithmic services (event detection, transactions)
- simple nodes that provide application-oriented functionality
(e.g., travel agencies)

# Architecture



# Part IV: Syntax Details and Implementation

# ECA Architecture



# Tasks

- ECA Engine: Rule Semantics
- Generic Request Handler: Mediator with Component Engines
- Component Engines: dedicated to certain Event Algebras, Query Languages, Action Languages
- Domain Services (Portals): atomic events, queries, atomic actions

# Communication of Variable Bindings

XML markup for communication of variable bindings:

```
<eca:variable-bindings>
 <eca:tuple>
  <eca:variable name="name" ref="URI"/>
  <eca:variable name="name"> any value </eca:variable>
   :
 </eca:tuple>
 <eca:tuple> . . . </eca:tuple>
   :
 <eca:tuple> . . . </eca:tuple>
</eca:variable-bindings>
```

# Communication ECA → GRH

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

```
[Sample: a query component]
<eca:query xmlns:ql="url"
  rule="rule-id" component="component-id">
  <!-- query component -->
< eca:query>
<eca:variable-bindings>
  <eca:tuple> ... </eca:tuple>
      :
  <eca:tuple> ... </eca:tuple>
<eca:variable-bindings>
```

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

# Generic Request Handler

- Submits component to appropriate service
- if necessary: does some wrapping tasks
  (for non-framework-aware services, mainly in opaque cases)
- receives results
- wraps them in a message that is sent back to the ECA engine

# Communication Component Engine → GRH

- result-bindings-pairs (semantics of expression)

```
<eca:answers rule="rule-id" component="component-id">
  <eca:answer>
    <eca:result>
      <!-- functional result -->
    </eca:result>
    <eca:variable-bindings>
      <eca:tuple> . . . </eca:tuple>
          :
      <eca:tuple> . . . </eca:tuple>
    </eca:variable-bindings>
  </eca:answer>
  <eca:answer> . . . </eca:answer>
      :
  <eca:answer> . . . </eca:answer>
</eca:answers>
```

# Communication GRH $\rightarrow$ 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

# Special Issue: Functional Results

Example: Event Component

```
<eca:query xmlns:ql="uri">
  <eca:variable name="name">
    event specification
  </eca:variable>
</eca:query>
```

- GRH submits *event specifi cation* to processor associated with *uri*
- GRH receives **answer(result,variable-bindings*)** elements from event detection engine
- binds <**result**> to *name* and extends <**variable-bindings**>

# Special Issue: Opaque Components

Example: wrapped, framework-aware XQuery engine

```
<eca:query>
  <eca:opaque lang="uri">
    code fragment in language lang
  </eca:opaque>
</eca:query>
```

- GRH submits *event specifi cation* to processor associated with *lang*
- GRH receives **answer(result,variable-bindings*)** elements from event detection engine
- and returns them to ECA engine

# Part V: Further Issues

# Special Aspects: Indirect Communication

Communication via intermediate services:

- indirect communication: publish/subscribe – *push/push*
  sources publish data/changes at a service, others register there to be informed
  + requires (less) activity by provider
- indirect communication: continuous queries – *pull/push*
  register query at a continuous query service
  + acceptable load also for "important" sources
  + shorter intervals possible

# Special Aspects: Intermediate Services

Intermediate services can add functionality:

- information integration from several services
- checking query containment
- caching
- acting as information brokers (possibly specialized to an application area)

# Further Issues

## Normal Form vs. Shortcut

- note that parts of the condition can often already checked earlier during event detection
- most event formalisms allow for small conditions already in the event part (e.g., state-dependent predicates and functions; cf. Transaction Logic)

# Summary

- first: diversity looked like a problem, lead to the Web (XML) and the Semantic Data Web (RDF and OWL data);
- heterogeneous data models and schemata:
  $\Rightarrow$ RDF/OWL as integrating semantic model in the Semantic Web
- extend these concepts to describing behavior
- describe events and actions of an application within its RDF/OWL model
- diversity + unified Semantic-Web-based framework has many advantages
- languages of different expressiveness/complexity available
- markup+ontologies make expressions accessible for reasoning about them

# Summary

- architecture: functionality provided by specialized nodes
- Local: triggers (SQL, XML, RDF/Jena, ...)
  - local updates
  - raise higher-level events
- Global: ECA rules
  - components
  - application-level atomic events and atomic actions
  - specific languages (event algebras, process algebras)
  - opaque (= non-markup, program code) allowed
- Communication: events, event broker services, registration
- Identification of services via namespaces

# Further Information

- REWERSE Deliverable I5-D4: “Models and Languages for Evolution and Reactivity”: Everything + examples
- Prototypes:
  - generic ECA engine with interfaces (GOE BSc)
  - Jena+Triggers (GOE/CLZ Diploma)
  - Cooperation within REWERSE I5 with RuleCore (U Skövde/Sweden) and XChange (LMU München/Germany)