Evolution and Reactivity on the Semantic Web

José Júlio Alferes Wolfgang May

CENTRIA, Universidade Nova de Lisboa, Portugal

Institut für Informatik, Universität Göttingen, Germany

Reasoning Web Summer School, Malta, July 25-29, 2005

GOALS OF THIS TALK

- to have a profound knowledge of underlying theoretical concepts and existing approaches,
- to be able to recognize, apply, combine and extend them,
- not to invent triangle wheels again,
- not to design things that at the end turn out to be not sufficient.

OVERVIEW

1. Foundations and analysis of the situation,
2. Logical formalisms,
3. A proposal,
4. Presentation of existing approaches, ...
5. ... leading to the discussion.

Note: the LNCS text contains a more detailed description of (1) and (2), including references to existing related work:

- “traditional” concepts that have to be considered when designing a Semantic Web framework,
- existing research for the Semantic Web,
- see also Overview/State-of-the-Art deliverables of REWERSE WGs.

Table of Contents

Chapter 1
Introduction

1.1 The Semantic Web

Today's Portals

- early Web: HTML ($\rightarrow wrapping$)
- underlying information on different servers in different data models (relational DM, HTML, XML)
- XML data wrt. different DTDs
- portal integrates a fixed set of different sources for presentation and answering queries using integration mappings

THE SEMANTIC WEB

... continue the idea of today's portals:

- information must be independent from its actual location
  ("talk *about* something")
- combination of information by "the Web" (not by the user)
  $\Rightarrow$ union instead of mappings
- information sources must support this (cooperation)
- semantics-based, not syntax/data-structure-based querying

The Semantic Web: RDF

- the "Web" structure of documents/data provides background infrastructure to distributed data (whose actual model is independent from their location)
- location of *services* still given by URLs

RDF: Resource Description Format

- Triples: (subject (resource), predicate, object (resource))
- Resources: "objects" of interest, described by URIs
  (Uniform Resource Identifiers)
- RDF: node- and edge-labeled graph data model (cf. OEM)
- RDF *can* be *represented* (serialized) by XML data
- physical distribution of this data independent from the described objects

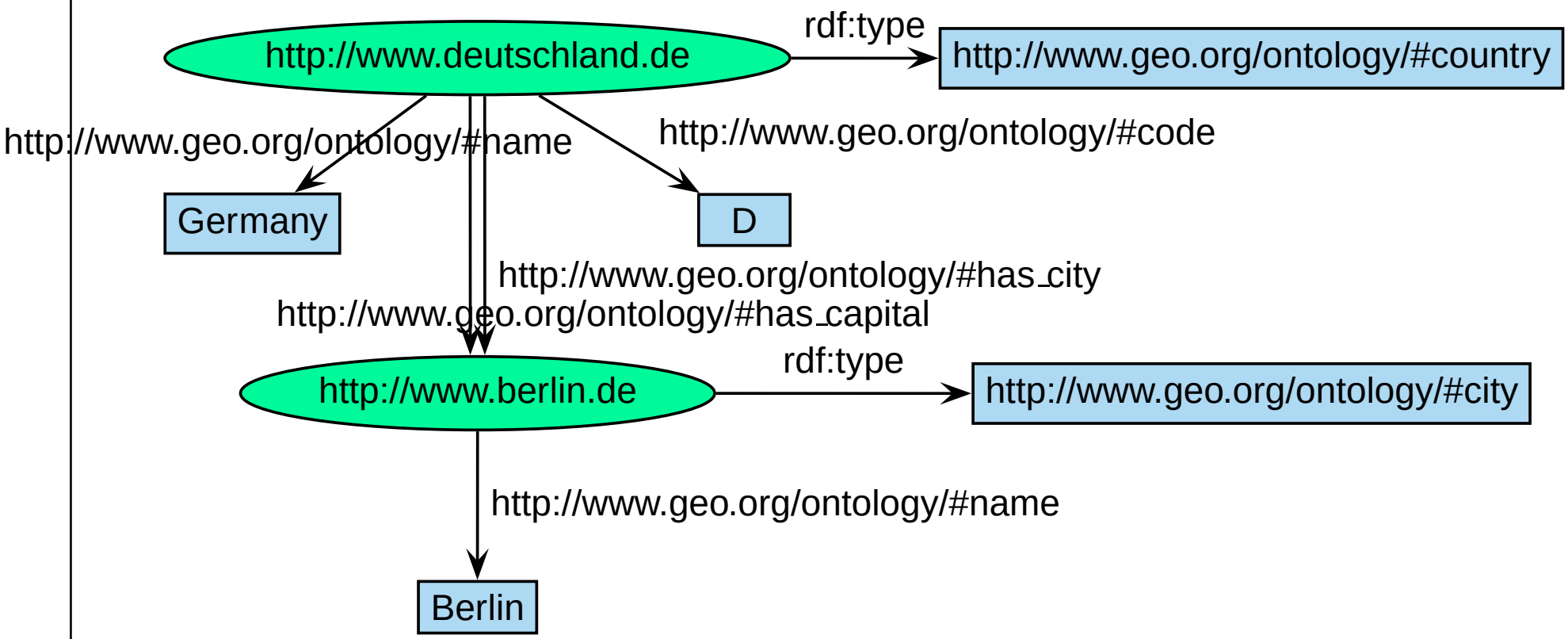
$\Rightarrow$ distributed, strongly intertwined, but very loosely coupled database

RDF: EXAMPLE

```
<rdf:RDF xmlns:geo= “http://www.geo.org/ontology#”>
  <rdf:Description rdf:about=“http://www.deutschland.de”>
    <rdf:type rdf:resource= “http://geo.org/ontology#country”/>
    <geo:name>Germany</geo:name>
    <geo:code>D</geo:code>
    <geo:has_city>
      <rdf:Description rdf:about= “http://www.berlin.de”>
        <rdf:type rdf:resource= “http://geo.org/ontology#city”/>
        <geo:name>Berlin</geo:name>
      </rdf:Description>
    </geo:has_city>
    <geo:has_capital rdf:resource=“http://www.berlin.de”/>
  </rdf:Description>
</rdf:RDF>
```

- there can be different RDF files that also describe Germany and Berlin, using the same URIs, and the same or even other (overlapping) ontologies.

RDF: Example Graph



THE SEMANTIC WEB: DATA MODEL AND INTEGRATION

- Global data model:
  - RDF format
  - RDF Schema
    defines the notions that are used with <rdf:type>,
    and the notions that are used as property names
    (with this, predicates also become resources)
  - OWL (Web Ontology Language (+Semantics))
- Note: not the XML representation is relevant, but only the RDF triples that are described by it!
- no integration necessary if sources agree on the used ontologies (names+URIs)
- Global architecture:
  - physical: *autonomous data sources* (located at URLs) that hold actual information about ...
  - ... *resources* that are in general not Web nodes, but represent *real* things (represented by URIs)

1.2 Analysis

- Why RDF, Ontologies, and OWL?
  Heterogeneity of data models and notions used by different data sources
  $\Rightarrow$ data integration
  RDF provides a unified simple model with seamless integration of ontologies
  - sometimes: mapping between ontologies
    (ontology level [synonyms, hyponyms etc], not XML level views/mappings as before in data integration)

BEHAVIOR IN A "PASSIVE" SEMANTIC WEB

... so far, there are no actual processes, but some behavior is already required:

Cooperative Query Answering in the Semantic Web

query answering (e.g., RDQL):
actual integration of RDF/OWL information is simple
task: *querying* this information
distributed (P2P) process
$\Rightarrow$ communication and collection of queries and answers

Note: also policies, trust, dealing with incomplete knowledge etc. needs reasoning that is often described and implemented in a procedural way.

Cooperative Evolution of the (Semantic) Web: Update Propagation

overlapping ontologies and information between different sources:

- consistency between data sources
  (although querying deals with a static notion of state, it requires that this state is consistent)
- $\Rightarrow$ local updates of data sources must be propagated
  while state change in a database is an update, state change in the (Semantic) Web requires “background” activity.
  - directed communication between dependent data sources (push, pull, ...)
    tightly coupled peers, e.g. in bioinformatics: sources are known
  - communication to unknown data sources?
    updates as “events” (e.g. “our address changes to ...”)

  $\Rightarrow$ *reactivity*

SEMANTIC UPDATES

... contained information is independent from its location.

- updates: in the same way as there are semantic query languages, there must be a semantic update language.

– updating RDF data: just tell (a portal) that a property of a resource changes
  intensional, global updates
  $\Rightarrow$ must be correctly realized in the Web!

- "the Web" must do something – it's "the Web", not a single page/site!

- communication and view maintenance
  rough ideas:
  - tell each source explicitly what changes ("push")
  - "send" update to a portal, the sources have to care for relevant updates

  both cases: *reactivity* – see such updates as *events* where sources must react upon.

COOPERATIVE EVOLUTION OF THE SEMANTIC WEB: PROCESSES

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

- Semantic Web as a base for processes
  - Business Rules, designed and implemented in participating nodes: banking, travel booking, . . .
  - 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"*

1.3 Semantic Web: Abstraction Levels

⇒ evolution of single pages (updates + reasoning)

⇒ evolution of the Web (updates + processes)

- local vs. remote/distributed
- physical data model (XML) vs. logical data model (RDF in a node)
- combination: Web-wide: global, RDF/OWL

Infrastructure and Principles

⇒ communication

⇒ reactivity: communication $\rightsquigarrow$ evolution

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, e.g. in bioinformatics: 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
    mainly: for queries, but also for "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 whole as a network of *communicating* nodes

Dependencies between different Web nodes require to propagate changes:

- global Semantic Web model is an integrating view,
  overlapping sources $\rightarrow$ consistency
- (the knowledge of) every node presents an excerpt of it
  - view-like with explicit reference to other “master” 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)

Updating Data on the Web

Usually, query languages are directly extended to update languages (e.g. SQL, XQuery + Updates)

RDF update language with several “levels”:

- updates of local data (XML; RDF)
- remote updates of certain sources by messages/method calls – mapped to local updates (XUpdate, XQuery+Updates, XChange; RDF-Update)
  (closely coupled nodes, cf. federated databases)
- RDF/OWL: intensional updates
  ... behavior of “the Web” as a whole
  - P2P infrastructure: communicate updates to nodes
  - nodes: map and implement them in local repository

Chapter 2
Development of a Solution

For most issues, concepts are already around:

- The internet (protocols like IP, HTTP, ...)
- Data models and query languages
- logical formalisms for individual states, including open-world semantics
- languages for expressing queries, updates, and behavior
- communication strategies
- logical formalisms that allow for modeling and reasoning about state sequences

$\Rightarrow$ combine them and adapt them to the Semantic Web

$\Rightarrow$ develop new concepts (languages, logics) where needed

2.1 Query and Update Languages

Query Languages

see course on Query Languages.

- XML level: XML-QL, XQuery, XPathLog, Xcerpt

  Common design of such languages:

  1. bind variables by FROM-like clause (XQuery: for-let)
  2. evaluate conditions in WHERE-like clause
  3. format/construct answer from variable bindings in SELECT-like clause (XQuery: return)
     - note: underlying idea are “rules”
     - functional style: nested queries, logic programming style: derived views + fixpoint semantics

- RDF level: RDQL
- semistructured data in general: OEM/Lorel, F-Logic

UPDATE LANGUAGES

Extensions of Query Languages

Common design of such languages:

1. bind variables by FROM-like clause (XQuery: for-let)
2. evaluate conditions in WHERE-like clause
3. apply updates to the elements bound to variables

- XQuery+Updates

```
update
for $c in document("mondial.xml")//country
replace $c/population/text() with
   $c/population * (1 + $c/population_growth div 100)
```

- XChange
- similar proposals for RDF
- declarative, but operational description of “what to change”

UPDATE LANGUAGES

XUpdate: XML:DB's proposal (1999)

Updates as "messages" in XML:

<xu:modifications xmlns:xu= "http://www.xmldb.org/xupdate">
<xu:*operation* select= "*xpath-expression*">
contents (e.g. new value)
</xu:*operation*>
</xu:modifications>

```
<?xml version="1.0"?>
<xu:modifications version="1.0" xmlns:xu="http://www.xmldb.org/xupdate">
  <xu:update select="/mondial/country[name='Germany']/population">
     80000000
  </xu:update>
</xu:modifications>
```

- can be generated as XML, can be sent via HTTP
- declarative description of "what changes"
- easy to derive a similar concept that uses RDF URIs

2.2 Evolution and Behavior

Behavior is ...

... doing something

- when it is required
  - upon interaction (user, or service call)
  - abstractly: as a reaction to a message
  - as a reaction to an internal event (temporal, update)
- continuously (process)
  - which in fact most of the time does nothing, but also reacts on events

Event-Condition-Action Rules are a well-known paradigm (not only in Active Database Systems).

Let's discuss simple cases first ...

LOCAL EVOLUTION OF WEB NODES

- Web nodes with local behavior
- react on
  - local updates
  - incoming messages
  - temporal events
  - possibly poll/query other sources

$\Rightarrow$ Trigger-like
update/message + condition $\rightsquigarrow$
update (possibly including a remote query)

TRIGGERS ON THE XML LEVEL

- similar to SQL triggers:
  ON *update* WHEN *condition* BEGIN *action* END
- *update* is an event on the XML level
  - 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 INSERT [IMMEDIATELY] [SIBLING] BEFORE|AFTER` *xsl-pattern*:
  insertion of a sibling

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

TRIGGERS ON THE RDF LEVEL

Events on the RDF Level

- `ON {DELETE|INSERT|UPDATE} OF` *property* `[OF` *class*`]`.
- `ON CREATE 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 UPDATE OF {CLASS` $class$ `| PROPERTY` $property$`}`,
- `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.

2.3 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

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

Intermediate services can add functionality:

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

Chapter 3
Evolution & (Re)Activity

- 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)
  - 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)

ECA Rules

- intended basic paradigm: reactivity
  - communication/messages
  - specification and implementation of (at first: local) behavior

---

$\Rightarrow$ homogeneous, modular framework

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

EVENTS

- physical/local events (updates on the local knowledge)
- communication events (wrapped as messages)
  - explicit queries (XML: XQuery etc.; Semantic Web: RQL etc)
  - answers (in XML/RDF)
  - other messages (especially on the RDF level)
    $\Rightarrow$ markup language for general messages (SOAP-like)
- system events, temporal 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”
  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”

EVENTS ON THE SEMANTICAL LEVEL

Declarative extension for evolution of “the Web”:
global application-level events “somewhere in the Web”

- “on change of VAT do ...”
- “if a flight is offered from FRA to LIS under 100E and I have no lectures these days then do ...”
- local event (update) to a certain Web source
  (but: applications not actually interested where it happens)
- updates can raise “intensional” events:

  action: “book a flight FRA-MAL, 23.7.2005”
  update: some changes in the Lufthansa database
  events: “a booking of seat 18A on flight LH0815, 23.7.2005”

  “LH0815, 23.7.2005 is fully booked”

  “there are no more tickets on 23.7. from Germany to MAL”

$\Rightarrow$ requires detection/communication strategies

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 if actually something has to be done.
  This probably requires to investigate data that has not been explicit in the event
  $\Rightarrow$ evaluate a ((Semantic) Web) query.
- Action: actually *does* something.

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:
  - OK: “a change of Alice’s bank account”
  - OK: “a debit of 200E from Alice’s bank account”
  - But: (if) “the balance of Alice’s bank account becomes below zero” ... already contains a condition

  ... either define it as a *derived event* in the ontology, or move parts into the condition component.
- Condition: “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
  - my lectures are surely not contained there

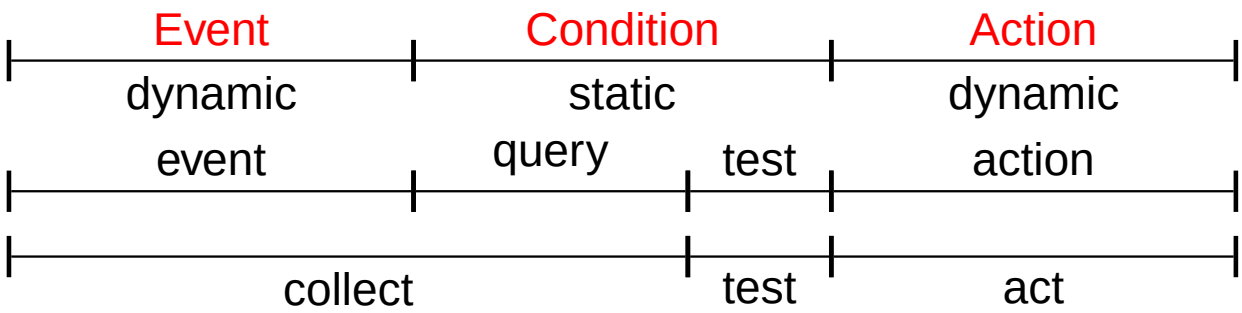
  $\Rightarrow$ includes another query before evaluating a “WHERE” (boolean) condition
  (SQL: would be in an `select ... into ...` and `if` in the action part.

GOAL

Focus on a *clean, declarative* design as a “Normal Form”:

- detect just the dynamic part of a situation,
- then obtain additional information by a query,
- then evaluate a *boolean* condition,
- then actually do something.

Rule Components:



- communication/propagation of information from E→Q→C→A component by *logical variables*.

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)

$$
\begin{aligned}
(E_1 \nabla E_2)(t) &:\Leftrightarrow E_1(t) \vee E_2(t) \ , \\
(E_1 \triangle E_2)(t) &:\Leftrightarrow E_1(t) \wedge E_2(t) \ , \\
(E_1 ; E_2)(t) &:\Leftrightarrow \exists t_1 \leq t : E_1(t_1) \wedge E_2(t) \ ,
\end{aligned}
$$

- usually, a composite event is detected when its “final” subevent is detected

ASIDE: ALGEBRAS

An *algebra* is simply: a *domain* (i.e. a set of "things"); a *set of operators* over the domain (i.e. mapping elements, pairs, triples, ... of elements of the domain to elements of the domain).

Algebra expressions are formed by elements of the domain and operators.

- term algebra: the "things" are numbers, and the operators are +, -, *, /
- boolean algebra: true/false, boolean operators
- word algebras: characters and concatenation
- event algebras: atomic events and some operators

Algebras as a Generic Concept

- "terms": tree structures
- for us: "standard" XML markup of term languages
  (later: RDF composite events)

EVENT ALGEBRAS: REPORTING PARAMETERS

- more complex semantics: with parameters
- “join” variables between atomic events
- “safety” conditions similar to Logic Programming rules

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

note: result parameters must be bound by both alternatives

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

Advanced Operators (Example: SNOOP)

- "negated" events, "cumulative" events
- ANY$(m, E_1, \ldots, E_n)(t) \;\; :\Leftrightarrow \;\; \exists t_1 \leq \ldots \leq t_{m-1} \leq t,\ 1 \leq i_1, \ldots, i_m \leq n$ pairwise distinct s.t. $E_{i_j}(t_j)$ for $1 \leq j < m$ and $E_{i_m}(t)$,
- $\neg(E_2)[E_1, E_3](t) \;\; :\Leftrightarrow \;\; E_3(t) \wedge (\exists t_1 : E_1(t_1) \wedge$
  $\wedge(\forall t_2 : t_1 \leq t_2 < t : \neg(E_2(t_2) \vee E_3(t_2))))$,
  "if E_1 and then a first E_3 occurs, without occurring E_2 in between"
- "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)
- note that parameters have to be dealt with separately

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:
  $E_6(A) := (\neg(\exists X : deposit(A, X)))[deposit(A, Am) \wedge t = date; date = t + 100days]$
- 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:
  $E_8(A, list) := \mathcal{A}^*(first\_of\_month, E_3(A), first\_of\_next\_month)$

EVENT DETECTION

- Local events, incoming messages, temporal events: easy
- composite events of event algebras:
  incrementally on-the-fly: automata/graphs, temporal formulas, tree of operator instances,
  extended with variable bindings (parameters)
  Note: there are no “event bases” that store past events and that can be queried
- intensional events, application-level events:
  (cf. “if a flight is offered from FRA to LIS under 100E”)
  - detection by derivation from another event
    (e.g., raising the intensional event by a trigger upon a basic event)
    (can be defined in the ontology - or add it to the distributed ontology)
  - translation to a query that is then monitored
    (“on change of something”)

CONDITIONS

check if after a dynamic situation that *probably* requires a reaction (“wake up”), a reaction is actually required (“get up”).

- maybe requires to obtain additional information ($\Rightarrow$ Query):
  - local
  - distributed/remote at a certain node
  - Semantic Web-level
- *Then* check a boolean condition (including predicates of the application domain)

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. later: Transaction Logic)

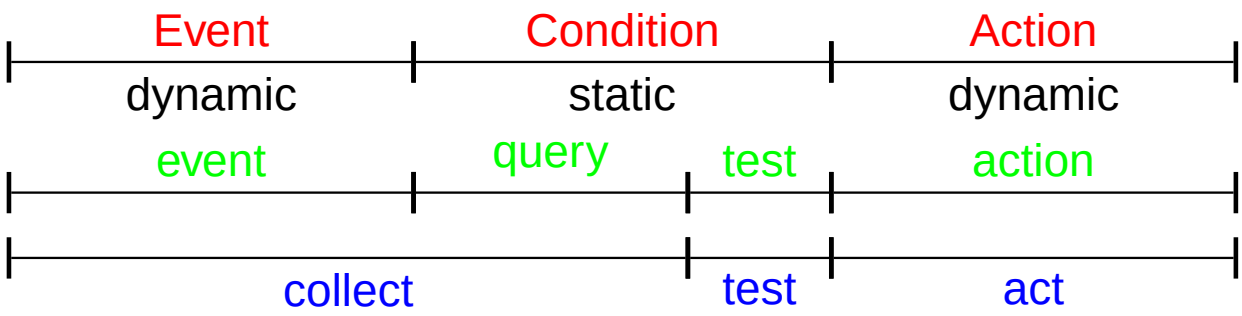
ACTIONS

- updates of the local state:
  - facts
  - knowledge derivation rules
  - rules describing the behavior of a node

  $\Rightarrow$ all kinds of knowledge are updatable
  local evolution
- calls of procedures/services
- sending messages
- transactions
  including queries against other sources
- intensional actions (e.g. in *business rules*)
  - ... then raise the share at company X to 51%
    (must in general be accomplished by buying some stocks, or buying another company that holds stocks of X)

ECA Rules

Rule Components:



- 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
    (process algebras – see later, or programming languages)
- propagation of values between the different parts of a rule by variables
  (bound in the collect part and used afterwards)

Chapter 4
Logical Formalisms for Reasoning about Evolution & Reactivity

Theoretical background and formal means for analyzing and describing concepts of Evolution and of Reactivity, in general.

More specifically:

- General models for evolution
- Temporal Logics
- Labelled Transition Systems
- Action Languages

4.1 Models for Evolution

Kripke Structures

An individual (static) state may be modelled by a (first-order) structure. An infrastructure is needed for considering various states (resulting from evolution) and their relationship. A *Kripke structure* is just that. Formally, it is $\mathcal{K} = (\mathcal{G}, \mathcal{R}, \mathcal{M})$, where:

- $\mathcal{G}$ is a set of (static) states;
- $\mathcal{M}$ maps each state to a first order structure;
- $\mathcal{R}$ is a relation between states.

An example of a relation might be one that imposes linearity of all states, this representing linear time (points).

A *path* in a Kripke structure is a sequence of states $g_0, g_1, \ldots$, such that $\forall i, (g_i, g_{i+1}) \in \mathcal{R}$.

4.2 Temporal Logics

Formulas for describing temporal concepts, that constrain the set of possible Kripke structures (Kripke-models).

Extend first order logics with modal operators for accessing the structure and "talking" about time. E.g.:

- $\Box F$ – "always" (F holds in all subsequent states);
- $\Diamond F$ – "sometimes" (F eventually holds in a subsequent state);
- $\circ F$ – "next" (F holds in the next state);
- F until G – "until" (there is a subsequent state where G holds, and in all states between now and this state, F holds).

There is a satisfaction relation, defining when a structure (at a state) satisfies a formula. *Kripke models* of a given set are those that satisfy all its formulas.

LINEAR TIME LOGICS

Only linear Kripke structures (representing linear time).

The basic operators are $\circ F$ and F until G.

"always" and "sometimes" are defined as $\Diamond P := \text{true until } P$ and $\Box P := \neg\Diamond\neg P$.

The meaning, at a state g of the basic operator (in the propositional case) is defined by:

$$
\begin{array}{lcl}
g \models A & :\Leftrightarrow & M(g) \models_{PL} A \ , \\
g, \models \neg F & :\Leftrightarrow & g \not\models F \ , \\
g \models F \wedge G & :\Leftrightarrow & g \models F \text{ and } g \models G \ , \\
g_i \models \circ F & :\Leftrightarrow & g_{i+1} \models F \ , \\
g_i \models F \text{ until } G & :\Leftrightarrow & \text{there is a } j \geq i \text{ s.t. } g_j \models G \\
 & & \text{and for all } k : i \leq k < j,\ g_k \models F \ .
\end{array}
$$

BRANCHING TIME LOGICS

For considering (and ‘talking” and reasoning about) different possible futures, linear structures are not enough!

Tree-like (branching) Kripke structures. In them, e.g.:

- $\Diamond F$ means there is a future in which F holds – “not never” (different from F holding in the future –“sometimes”).

CTL (Computation Tree Logic):

- considers this branching, and
- adds existential and universal path quantifiers EF meaning: “there exists a path in which F”

Several extensions of CTL exist:

- CTL$^+$ and CTL* (see Lecture Notes)
- Past Tense Logics add past-time temporal operators: $\bullet$ (previous state), $\blacklozenge$ (sometimes in the past), $\blacksquare$ (always in the past), and since (e.g., A since B), symmetrical to the future tense operators.

4.3 Labelled Transition Systems

The Temporal Logics (and Kripke structures) presented do not pay special attention to transitions.

Labelled Transition Systems extend Kripke structures by labelling the (transitions) relations between states with *actions*.

Given an action a, $\mathcal{R}(a)$ is the set of all transitions labelled by a.

Logics for Labelled Transition Systems (such as *Dynamic Logic*) extend Temporal Logics with modal operators labelled by actions: $\langle a \rangle_{DL}$ – possibly with a; $[a]_{DL}$ – always with a.

The formal meaning is:

$$[a]_{DL} F := \neg \langle a \rangle_{DL} \neg F$$

$$g \models \langle a \rangle_{DL} F : \quad \Leftrightarrow \ \exists \text{ state } h \mid (g, h) \in \mathcal{R}(a) \text{ and } h \models F.$$

An action can be atomic, or a program formed with operators. In Dynamic Logic: ";" – sequential composition, $\cup$ – alternative composition ("choice"); * – iteration.

EXAMPLE

Consider bank accounts whose balances are given by $balance(name) = value$. There are actions of $debit(name, value)$ and $deposit(name, value)$.

The meaning of $debit$ (similar for $deposit$):

$$\forall N, V_1, V_2 : (balance(N) = V_1) \rightarrow ([debit(N, V_2)]\ (balance(N) = V1 - V2))$$

A Kripke structure satisfying this:

$$\mathcal{M}(g_0) = \{balance(Alice) = 200, balance(Bob) = 100\}$$
$$\mathcal{M}(g_1) = \{balance(Alice) = 180, balance(Bob) = 100\}$$
$$\mathcal{M}(g_2) = \{balance(Alice) = 180, balance(Bob) = 120\}$$
$$(g_0, g_1) \in \mathcal{R}(debit(Alice, 20))\ ,\quad (g_1, g_2) \in \mathcal{R}(deposit(Bob, 20))\ .$$

For stating that a transfer between any two accounts does not change their total balance:

$$\forall S, A_1, A_2, T : \quad (Sum = balance(A_1) + balance(A_2)) \rightarrow$$
$$([deposit(A_1, A); debit(A_2, A)]\ (Sum = balance(A_1) + balance(A_2)))$$

PROCESS ALGEBRAS

Algebraic description of *processes*:

- atomic (elementary) processes;
- constructors on processes (to build complex processes).

CCS algebra: Processes are elementary processes from $\mathcal{A}$, (process) variables and:

- $a : P$, where $a \in \mathcal{A}$ and P is a process (*sequential composition*);
- $P \times Q$, where P and Q are processes (*parallel composition*);
- $\sum_{i \in I} P_i$, with a set of indices I, and processes $P_i : i \in I$ (*alternative composition*);
- $P \upharpoonright A$, where $A \subseteq \mathcal{A}$ and P is a process (*restriction to a set of visible actions*);
- $\mathrm{fix}_j \vec{X} \vec{P}$, where X_i are variables, and P_i process (*binds variables to processes*).

Operational semantics of CCS given by transition rules.

SEMANTICS OF CCS

Transition rules for CCS:

$$a : P \xrightarrow{a} P \qquad \frac{P_i \xrightarrow{a} P}{\sum_{i \in I} P_i \xrightarrow{a} P} \text{ (for } i \in I\text{)} \qquad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q'}{P \times Q \xrightarrow{ab} P' \times Q'} ,$$

$$\frac{P \xrightarrow{a} P'}{P \restriction A \xrightarrow{a} P' \restriction A} \text{ (for } a \in A\text{)} \qquad \frac{P_i\{\mathsf{fix}\, \vec{X}\vec{P}/\vec{X}\} \xrightarrow{a} P'}{\mathsf{fix}_i \vec{X}\vec{P} \xrightarrow{a} P'} .$$

(Some) extra constructs:

$$\begin{aligned} \partial P &:= \mathsf{fix}\, X(1 : X \ + \ P) \quad , \quad X \text{ not free in } P \\ P|Q &:= P \times \partial Q \ + \ \partial P \times Q \end{aligned}$$

CCS Example

Processes allow to describe how transitions are composed. E.g.:

- A simple process for money transfer:

$$transfer(Am, Acc_1, Acc_2) := debit(Acc_1, Am) : deposit(Acc_2, Am) \ .$$

- A process for transferring a given amount from one account to another one, every first of a month (assuming the receipt of a message as a communicating action):

$$\text{fix } X(rec\_msg(first\_of\_month) : debit(Acc_1, Am) : deposit(Acc_2, Am) : \partial X)$$

Process algebras allow for complex descriptions of processes (for actions).

Their semantics allow for proving properties about the processes (model checking; here especially the CSP [Communicating Sequential Processes] algebra).

REASONING VERSUS "EXECUTING" EVOLUTION

These logics are adequate for reasoning about temporal structures.

- specifying conditions, and studying properties, invariants, etc.

Mechanisms are also needed to specify which actions are indeed executed, upon which events have indeed occurred.

Both are essential for expressing ECA-Rules: *if something happens then do something*.

Reasoning about evolution is needed, but not enough!

For execution:

- Rule based languages for evolution and reactivity;
- Event Algebras for expressing complex events;

4.4 Action Languages

Used (mostly in AI) for modelling dynamic environments, representing actions and their effects.

Formalization of actions is usually based on labelled transition systems. States are first order structures, where predicates are divided into *static* predicates and *fluents* (those that may change).

Action programs are sets of rules specifying which fluents change upon the execution of which actions.

Allow to predict the consequences of (really) executing a sequence of sets of actions, and also to determine which actions should be executed given a goal (planning).

4.4.1 Situation Calculus

Is a methodology for representing actions, which is in the basis of most action languages.

Fluent predicates are true at *situations* (or states). For stating that fluent f is true at situation s: $holds(f, s)$.

A situation is either the initial situation s_0, or the situation resulting from performing an action a at a situation S, denoted by the term $do(a, S)$.

Actions can be described by specifying rules for $holds$. E.g.:

$$holds((balance(A) = V), S) \rightarrow holds((balance(A) = V + D), do(deposit(A, D), S)$$

This describes what changes. But:

Frame problem: How to describe what doesn't change?

Other problems, mainly relevant in AI:

- Qualification problem: How to completely qualify the preconditions for an action?
- Ramification problem: How to completely anticipate the consequences of an action?

FRAME AXIOMS

To deal with the frame problem:

- Have general axioms describing inertia:

$$\forall P, A, S \; not \; holds(\neg P, do(A, S)) \wedge holds(P, S) \rightarrow holds(P, do(A, S)$$

  where not is a non-monotonic operator. Requires non-monotonic logic (e.g. logic programming).

- Or make frame axioms for fluents explicit:

$$\begin{aligned} holds((balance(C) = V_2), do(A, S)) \leftrightarrow holds((balance(C) = V_1), do(A, S)) \wedge \\ ((A = deposit(C, D) \wedge V_2 = V_1 + D) \vee \\ (A = debit(C, D) \wedge V_2 = V_1 - D) \vee \\ (A \neq debit(\_, \_) \wedge A \neq deposit(\_, \_) \wedge V_2 = V_1)) \end{aligned}$$

4.4.2 Languages $\mathcal{B}$ and related approaches

Developed to model actions in dynamic environments.

In $\mathcal{B}$, action programs are sets of *dynamic rules* (describing effects of actions):

$$A \textbf{ causes } L \textbf{ if } F$$

where A is an action name, L a fluent literal and F a conjunction of (static or fluent) literals.

For defining intemporal rules, $\mathcal{B}$ has *static rules*:

$$L \textbf{ if } F$$

stating that, in every state, if F then L must be true.

For example:

$$deposit(A, D) \textbf{ causes } (balance(A) = V + D) \textbf{ if } (balance(A) = V)$$
$$warning(A) \textbf{ if } (balance(A) < 0)$$

$\mathcal{B}$ SEMANTICS

The semantics of $\mathcal{B}$ is based on labelled transition systems.

In the transition system:

- states are all interpretations closed under the static laws,
- and there is an arc from a state s to a state s' with label a iff all Ls of dynamic rules of the form **causes** L **if** F, where F holds in s, belong to s', and nothing else differs from s to s'.

Alternatively, the semantics can be defined by translating $\mathcal{B}$ programs into Logic Programs using Situation Calculus: frame axioms use the non-monotonic *negation by default* of Logic Programming.

$\mathcal{B}$ RELATED APPROACHES

- The $\mathcal{C}$ language generalizes $\mathcal{B}$ by allowing arbitrary formulas to be caused by actions. Moreover it deals with causality, distinguishing between a fluent *holding*, and *being caused* (or *having a causal explanation*).
- LP-update languages (such as LUPS or EVOLP) allow actions to change rules (rather than "simply" to change fluents). In other words, in them actions can (also) be the addition, retraction or modification of a rule of the program itself. They allow to model highly dynamic situation, where the "rules of the game" may also change.

None of these make a clear distinction between Events and Conditions.

Chapter 5
Transaction Logic

An integrated language logic-programming style language for

- programming (state-changing) actions and transactions with ACID properties (Atomicity, Consistency, Isolation, Durability);
- executing actions;
- reasoning about actions.

It works on top of a data (or knowledge) base, and makes no assumption about it. States can be first-order (incl. relational databases), logic programs, XML or RDF resources, etc.

It is parametric on:

- a *state data oracle* $\mathcal{O}^d$ that answers queries (possibly with free variables) for every individual state, and
- a *state transition oracle* $\mathcal{O}^t$ which maps pairs of states to sets of transition formulas (typically assertions or retractions).

TRANSACTION LOGIC SYNTAX

Formulas are made with the connectives:

- $\alpha \otimes \beta$ – execute α and then execute β;
- $\alpha \wedge \beta$ – execute α and execute β along the same path;
- $\alpha \vee \beta$ – (nondeterministically) execute α or execute β;
- $\neg\alpha$ – execute whatever, provided that it is not a valid execution of α;
- $\alpha \leftarrow \beta (:= \alpha \vee \neg\beta)$ – one way to execute α is to execute β.

For example:

$$deposit(A, D) \leftarrow (balance(A) = V) \otimes balance.del(A, V) \otimes balance.ins(A, V + D)$$

$$safe\_debit(A, D) \leftarrow decrease\_balance(A, D) \otimes (balance(A) \geq 0)$$

MORE CONNECTIVES

These also include extension to *Concurrent Transaction Logic*:

- $\alpha \mid \beta$ – execute α and β in parallel;
- $\odot\alpha$ – execute α in isolation (like in databases);
- $\diamond\alpha$ – check if execution of α is possible;
- $\alpha \oplus \beta$ – (serial) disjunction of α and β.
- path – special constant always true (at every path)

For example:

$$transfer(V, A_1, A_2) \leftarrow debit(A_1, V) \mid deposit(A_2, V)$$
$$debit(A, V) \leftarrow \odot(balance(A, B) \otimes B \geq V \otimes change\_balance(A, B - V))$$
$$deposit(A, V) \leftarrow \odot(balance(A, B) \otimes change\_balance(A, B + V))$$

SEMANTICS OF TRANSACTION LOGICS

States are determined by the *data oracle* $\mathcal{O}^d$.

A path is a finite sequence of states $\pi = \langle s_1, \ldots s_n \rangle$ $(n \geq 1)$.
$\pi_1 \circ \pi_2 = \langle s_1, \ldots s_i \rangle \circ \langle s_i, \ldots s_n \rangle$ $(1 \leq i \leq n)$ is called a *split of* π.

Formulas are true over paths (not over states):

- ϕ is true in $\pi = \langle s_1, s_2, \ldots s_n \rangle$ means that ϕ can execute at s_1, changing it to s_2, ..., to s_n, terminating in s_n.

Compliance with oracles:

- if $\mathcal{O}^d(S) \models F$ for some state S, then F must be true at the path of length 1 $\langle S \rangle$;
- if $\mathcal{O}^t$ with formula F (e.g. $p.insert$ or $p.delete$) maps S_1 to S_2 then F must be true at path $\langle S_1, S_2 \rangle$.

Atomic formulas are true at a path π if it (classically) complies with the oracles in a model M.

MEANING OF CONNECTIVES

- $M, \pi \models \neg\alpha$ iff $M, \pi \not\models \alpha$ (cannot execute α over path π);
- $M, \pi \models \alpha \wedge \beta$ iff $M, \pi \models \alpha$ and $M, \pi \models \beta$ (α and β execute along the same path π);
- $M, \pi \models \alpha \vee \beta$ iff $M, \pi \models \alpha$ or $M, \pi \models \beta$ (either α or β execute along path π);
- $M, \pi \models \alpha \otimes \beta$ iff there is a split $\pi = \pi_1 \circ \pi_2$ such that $M, \pi_1 \models \alpha$ and $M, \pi_2 \models \beta$ (α executes at a prefix of π, and β at the remainder – α and then β);
- $M, \pi \models \alpha \oplus \beta$ iff for every split $\pi = \pi_1 \circ \pi_2$, $M, \pi_1 \models \alpha$ or $M, \pi_2 \models \beta$;
- $M, \langle s \rangle \models \diamond\alpha$ iff there is a path $\pi = \langle s, \ldots, s_n \rangle$ such that $M, \pi \models \alpha$ (α is possible at "state" s if there is a path starting in s where α executes);
- $M, \pi \models \mathsf{path}$ for every path π.

The semantics of Concurrent Transaction Logic is not discussed in this course.

TRANSACTION LOGIC PROGRAMS

Serial Horn Fragment of Transaction Logic – Sets of rules:

$$a_0 \leftarrow a_1 \otimes \ldots \otimes a_n$$

Read: one way to execute a_0, is to start by executing a_1 then . . . then execute a_n.

The execution of a_0 (by this rule) succeeds if all of $a_1 \otimes \ldots \otimes a_n$ succeed (transaction).

Various rules for a_0 determine various (nondeterministic) possibilities for executing it.

Each a_i can be:

- a predicate with rules like these – can be seen as triggering an event;
- or a "call" to the data oracle testing whether something is true at the state – condition testing;
- or a "call" to the transition oracle (e.g. $p.insert$ or $p.delete$) which changes the state accordingly (resp. inserting or deleting p) – action execution.

One possible use:

$$event(\overline{X}) \leftarrow pre\_conditions(\overline{X}, \overline{Y}) \otimes action(\overline{X}, \overline{Y}) \otimes post\_conditions(\overline{X}, \overline{Y})$$

EXECUTION OF PROGRAMS

M is a *model* of a transaction program P iff $M, \pi \models R$ for every path π and every rule $R \in P$.

If G is a formula (goal) and $S_0, \ldots, S_n$ a sequence of states then *executional entailment* is a statement:

$$P, S_0, \ldots, S_n \models G$$

which means that, for every model M of P, $M, \langle S_0, \ldots, S_n \rangle \models G$.

There exists a proof theory (not presented here) for executional entailment, and a corresponding implementation for Transaction Logic Programs.

In it there is a notion of "current state", and of queries/goals:

- Given a "current state" s_1, goal succeeds G succeeds iff $P, s_1 \ldots s_n \models G$.
- If G succeeds with $\langle s_1, \ldots s_n \rangle$ then the new "current state" is s_n.
- If G fails then the "current state" remains unchanged.

BANK EXAMPLE

$$transfer(V, A_1, A_2) \leftarrow debit(A_1, V) \otimes deposit(A_2, V)$$
$$debit(A, V) \leftarrow change\_balance(A, B, B - V) \otimes non\_negative\_acc(A)$$
$$deposit(A, V) \leftarrow balance(A, B) \otimes change\_balance(A, B, B + V)$$
$$change\_balance(A, B_1, B_2) \leftarrow balance(A, B_1).delete \otimes balance(A, B_2).insert$$
$$non\_negative\_acc(A) \leftarrow balance(A, B) \wedge B \geq 0$$

$\mathcal{O}^t$ includes $balance(A, B).insert$ (resp. $balance(A, B).delete$): true in every path $\langle S_1, S_1 \cup \{balance(A, B)\}\rangle$ (resp. $\langle S_1, S_1 - \{balance(A, B)\}\rangle$.

States are sets of facts for $balance(A, B)$, and the data oracle tests for membership in the set.

Let the initial state be $s_0 = \{balance(Bob, 100), balance(Alice, 50)\}$.

$? - transfer(50, Bob, Alice)$ succeeds, with final state $\{balance(Bob, 50), balance(Alice, 100)\}$.

$? - transfer(150, Bob, Alice)$ fails, leaving the state unchanged (note here the all-or-nothing, typical of transactions – atomicity).

(Prolog wouldn't do the same! In it assertions/retractions are not undone in backtracking)

BLOCKS WORLD EXAMPLE

States are defined with facts for $on(Block_1, Block_2)$, $clear(Block)$, and $wider(Block_1, Block_2)$.

The transition oracle includes insertion and deletion of these facts, as before.

The following defines actions which change the state in the blocks world:

$$
\begin{aligned}
pickup(X) &\leftarrow clear(X) \otimes on(X,Y) \otimes on(X,Y).delete \otimes clear(Y).insert \\
putdown(X,Y) &\leftarrow wider(X,Y) \otimes clear(Y) \otimes on(X,Y).insert \otimes clear(Y).delete \\
move(X,Y) &\leftarrow pickup(X) \otimes putdown(X,Y)
\end{aligned}
$$

$$
\begin{aligned}
stack(N,X) &\leftarrow N > 0 \otimes move(Y,X) \otimes stack(N-1,Y) \\
stack(0,X).
\end{aligned}
$$

$?- stack(5,a)$ *nondeterministically* chooses 5 (consecutively) clear narrower blocks and stacks them on top of block a.

$?- on(a,b) \wedge on(b,c)$ either fails (in case it is not possible to come to a situation where a is on top of b and b on top of c), or succeeds after executing a sequence of actions that makes $on(a,b)$ and $on(b,c)$ true.

Robot Example

States are defined with facts for $at(Room)$, $door(Room_1, Room_2)$.

The transition oracle includes insertion and deletion of facts $at(\_)$, as before.

$$conn(R_1, R_2) \leftarrow door(R_1, R_2) \vee door(R_2, R_1)$$

$$leadsTo(R_1, R_2) \leftarrow conn(R_1, R_2)$$
$$leadsTo(R_1, R_2) \leftarrow conn(R_1, R_3) \wedge leadsTo(R_3, R_2).$$

$$oneStepsInto(R, R_1, R_2) \leftarrow conn(R, R_1) \wedge leadsTo(R, R_2)$$

$$goTo(R_1) \leftarrow at(R_2) \otimes conn(R_1, R_2) \otimes at.delete(R_2) \otimes at.insert(R_1)$$
$$goTo(R_1) \leftarrow at(R_2) \otimes oneStepInto(R, R_2, R_1) \otimes at.delete(R_2) \otimes at.insert(R) \otimes goTo(R_1)$$

$$goThru(R) \leftarrow \mathsf{path} \otimes at(R) \otimes \mathsf{path}$$

$?- goTo(r1) \wedge goThru(r2) \wedge goThru(r3)$ – Go to $r1$ passing through $r2$ and $r3$ along the way.

$?- goTo(r1) \wedge \neg goThru(r2)$ – Go to $r1$ without passing through $r2$.

TRANSACTION LOGIC AND ECA RULES

Transaction Logic does not distinguishes between Events, Conditions and Actions.

By writing rules as (actions in the body):

$$event(\overline{X}) \leftarrow pre\_conditions(\overline{X}, \overline{Y}) \otimes action(\overline{X}, \overline{Y}) \otimes post\_conditions(\overline{X}, \overline{Y})$$

events are seen as queries (much like in logic programming) that trigger a proof (execution) process. In this (nondeterministic) process, other queries (events) may be triggered, conditions may be tested, and update actions executed.

These rules cannot be used for imposing temporal constraints such as: "Whenever E, if C is true then A must be executed".

For such constraints, one can use (right) *serial implication* (actions in the head):

$$\alpha \Rightarrow \beta \ :\Leftrightarrow \ \neg\alpha \oplus \beta$$

i.e.

$M, \pi \models \alpha \Rightarrow \beta$ iff for every split $\pi = \pi_1 \circ \pi_2$, $M, \pi_1 \not\models \alpha$ or $M, \pi_2 \models \beta$

For example, *"whenever there is a debit and the resulting balance is below zero, send a*

message":

$$debit(A, V) \otimes balance(A) < 0 \Rightarrow sendmsg(\ldots)$$

Chapter 6
ECA Rules in the Semantic Web

Static Aspects, Data Integration

- heterogeneous data models and schemata:
  $\Rightarrow$ RDF/OWL as integrating model in the Semantic Web

Dynamic Aspects

- describe events and actions of an application within its RDF/OWL model
- heterogeneous event/condition/action languages
  $\Rightarrow$ RDF/OWL-based ECA framework
  (that wraps these languages and relates their atomic items to the application 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., as "if *(composite) event* and *condition* then (raise) *derived event*") – ECE

  $\Rightarrow$ EC part/event detection not only in ECA rules, but also in ECE derivation rules for events

6.1 Languages and Markup

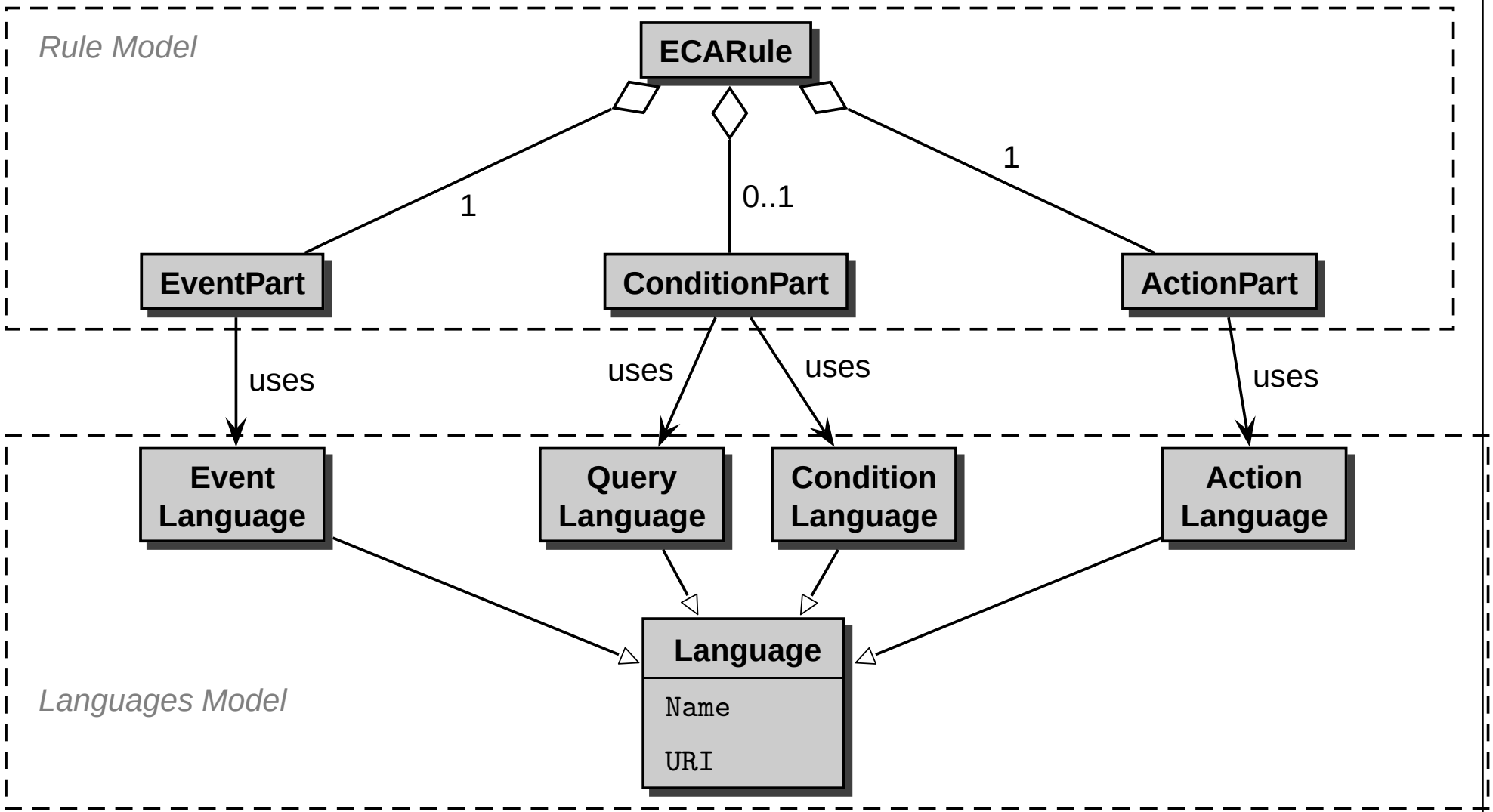
Heterogeneity on the Language level

common framework: ECA active rules, ECE derivation rules

- different levels of rules:
  - basic triggers,
  - simple ECA rules,
  - business rules on the semantic level
- rules over distributed data and distributed events with distributed actions
- intensional events and updates/actions
- rules using different E/Q/C/A languages for the same application, probably on the same data
- nodes that are not able to execute rules

$\Rightarrow$ Modular concepts with Web-wide services

Modular ECA Concept: Rule Ontology



RULE MARKUP: ECA-ML

<eca:rule *rule-specific attributes*>
declaration of variables
<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:condition *identification of the language* >
condition specification; using variables
</eca:condition>
<eca:action *identification of the language* >
action specification; using variables, possibly also binding local ones
</eca:action>
</eca:rule>

SUBCONCEPTS AND SUBLANGUAGES: EVENTS AND EVENT MARKUP

- e/q/c/a subelements contain a language identification, and appropriate contents
- algebraic languages (event algebras, query languages, boolean conditions, process algebras) have a natural term markup.
- every such language “lives” in an own namespace,
- (sub)terms must have a well-defined result.

SAMPLE EVENT MARKUP

Event part can be

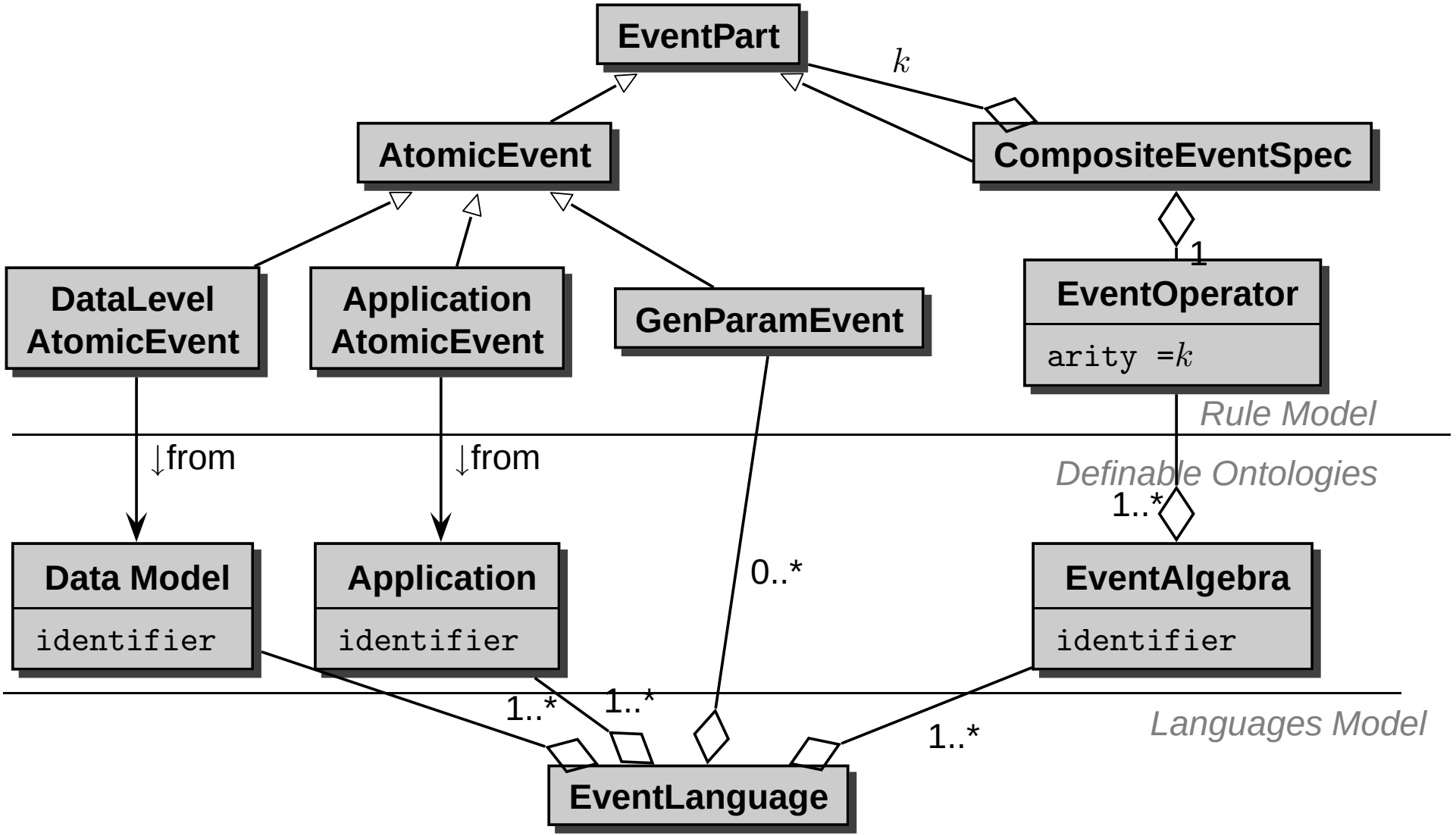
- an *atomic event* (e.g., in a trigger), or
- a composite event composed by an event algebra operator and a sequence of event arguments.

<**eca:rule**>
  :
 <**eca:event** *identification of the language (here: snoop)*>
  <snoop:sequence>
   <**eca:atomic-event**> *atomic event spec* <**/eca:atomic-event**>
   <**eca:atomic-event**> *atomic event spec* <**/eca:atomic-event**>
  </snoop:sequence>
 <**/eca:event**>
  :
<**/eca:rule**>

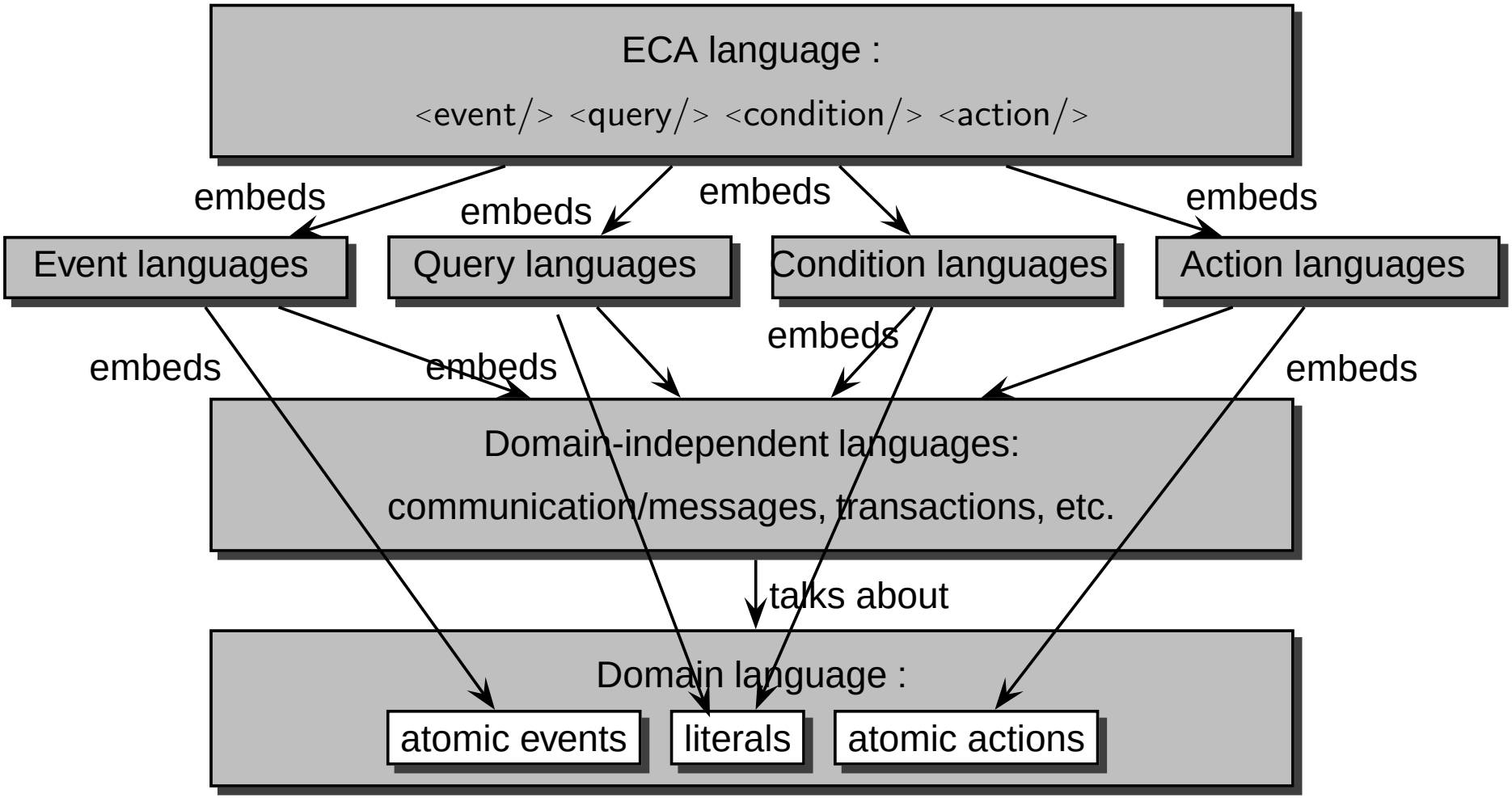
TYPES OF ATOMIC EVENTS

- Detection of atomic events depends on their types and
- their ontologies:
- Atomic Data Level Events [database system ontology]
  detected locally
- Generic Parameterized Events
  - receive message [common ontology]
    with contents [contents: own ontology] as parameter)
    detected locally
  - transactional events [common ontology]
    detected locally
  - temporal events [common ontology]
    provided by services (upon registration)
- Application-Level Events [own ontology]
  - raised by ECE rules from others
  - detected/derived by the Semantic Web

EVENT PART SUBONTOLOGY



LANGUAGE HIERARCHY



6.2 Logical Variables

- communication/propagation of information between E→Q→C→A components by *logical variables*.

Binding Strategies in the Collect Part

concerns: (Composite) Events and Queries

- Logic Programming (Datalog, F-Logic): binding variables by patterns
  proposal: markup of E and Q languages uses XSLT-style
  <variable name="*name*">
- functional style (SQL, OQL, XQuery): expressions return a value/fragment
  ⇒ must be bound to a variable to be kept and reused proposal: markup of E and Q languages use XSLT-style
  <eca:bind-variable name="*name*">-element or
  <eca:event bind-variable="*name*"/>-element

Using Variables

- as usual in XSLT or XQuery: $*name*

6.3 Engines and Processors – Service-Based Architecture

Language Processors as Web Services:

- ECA Rule Execution Engines that employ other services for E/Q/C/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:
  implement detection of one or more event algebras
- dedicated services for application-specific issues:
  - raising and communicating events
  - predicates
  - executing actions/updates
- query languages often implemented directly by the Web nodes (portals and data sources),
- evaluation of boolean conditions in the ECA engine.

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)
- [downloadable classes for processing]
- [description by a formal method for reasoning about it]

Application Languages/Ontologies

- DTD/XML Schema/RDF description of the language
- Atomic Event Broker Services (subscribe)
- [ECE event derivation functionality]

COMMUNICATION

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

$\Rightarrow$ different strategies

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

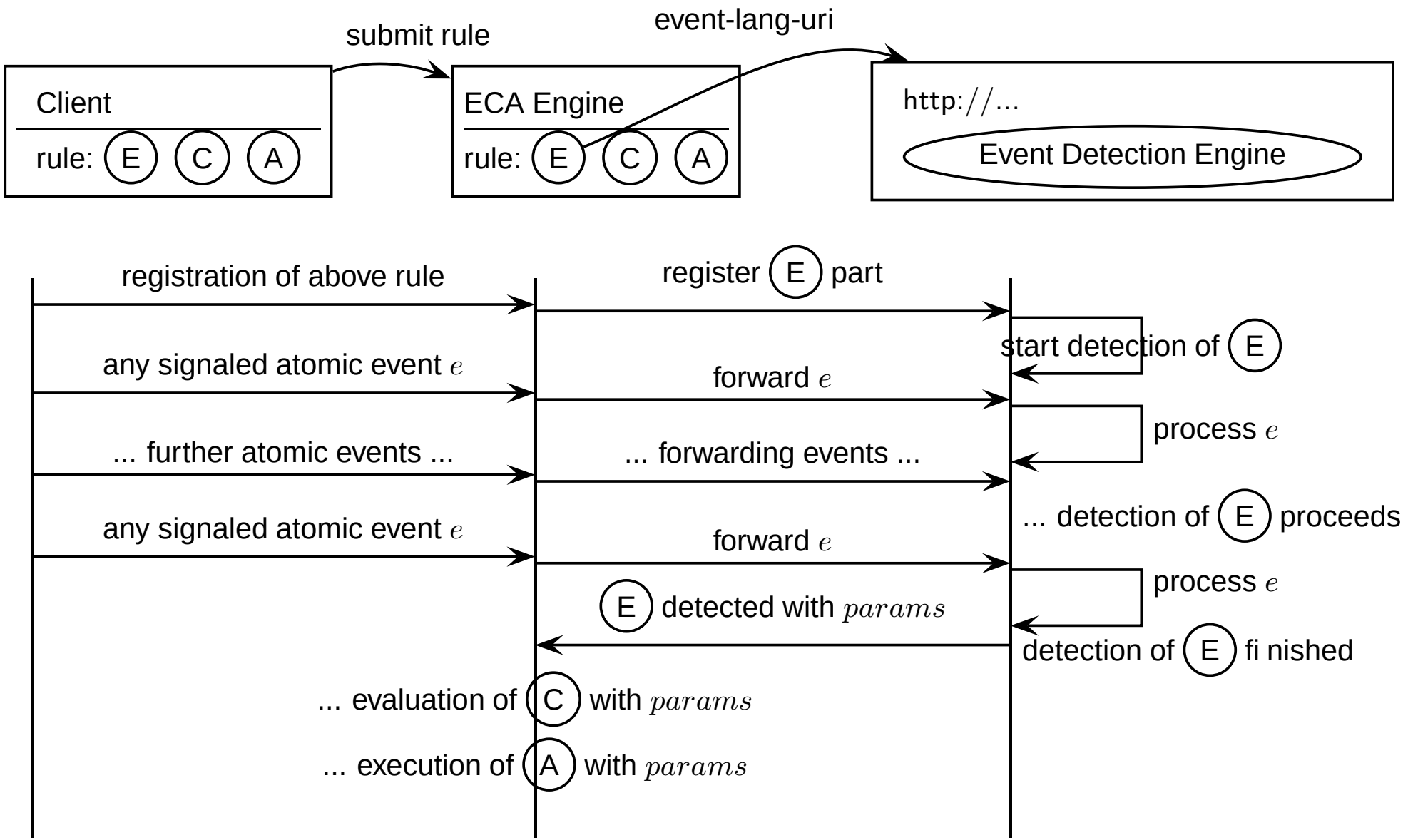
b) user/client registers rule and leaves the acquisition of events to the service:
   - $*$ event language/ontology: service
   - $*$ atomic events: provided by application-specific services

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 specification* at a service must forward events to the Event Detection Service

ARCHITECTURE

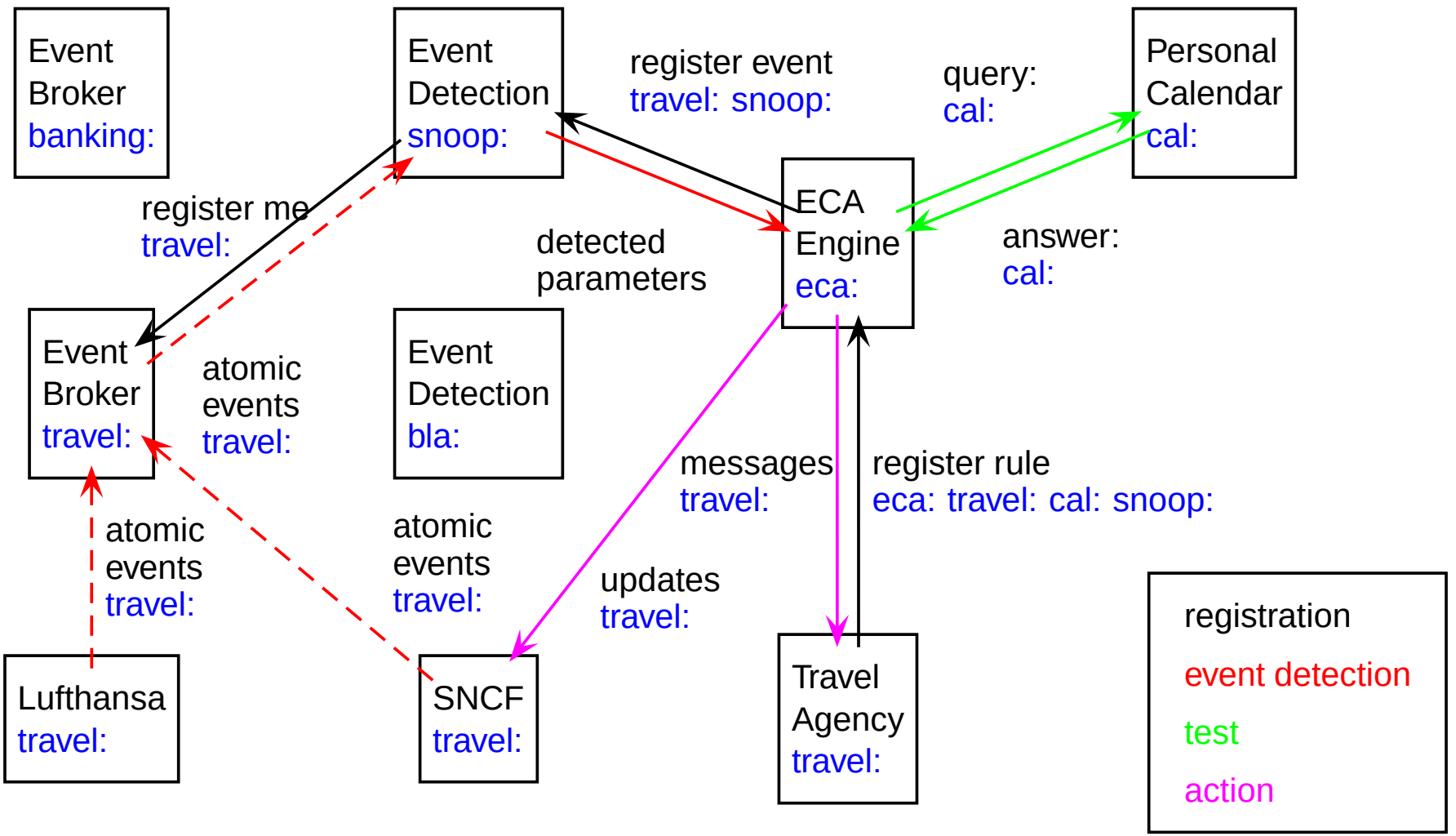


ADVANCED ARCHITECTURE

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



SUMMARY (EVOLUTION AND REACTIVITY)

- first: diversity looked like a problem, lead to the Semantic Data Web;
- now: diversity + unified Semantic-Web-based framework has many advantages
- languages of different expressiveness/complexity available
- functionality provided by specialized nodes
- sufficient if event detection services are informed about events;
  communication services can also be provided by specialized nodes

Chapter 7
Some existing languages and discussion

- Present some existing ECA-language for the Web:
  - “Active Rules for XML”;
  - Active XQuery;
  - E-C-A for XML;
  - RDFTL;
  - XChange.
- Discuss them in light of the previously exposed general model and framework.

"ACTIVE RULES FOR XML"

A. Bonifati, S. Ceri and S. Paraboschi. Pushing Reactive Services to XML Repositories Using Active Rules. *Computer Networks* 39(5): 645-660, 2002 (Previous version at WWW'01). Also available at: `http://www.icar.cnr.it/angela/pubs/ComputerNetworks.pdf`

- ECA-rules with XML markup
- react on atomic events in the local (XML) database;
- test conditions on `$new`, `$old` and static data;
- conditions are queries with empty or nonempty answer;
- actions are invocations of SOAP methods.

"ACTIVE RULES FOR XML"

- Event part:
  - mutating events: *insert, delete, update*
  - describe nodes to be monitored (via XPath);
  - every time such a node *changes* the event is raised, and `$new/$old` variables bound.
- Condition part:
  - XQuery query that binds variables in a FOR clause, test in the WHERE, and then (for each successful binding) "jump" to the action part.
- Action part:
  - SOAP method invocation that uses the variable bindings of the condition part.

"Active Rules for XML" – Example

Upon insertion of a `<cd>` *element, test whether the* `price` *of the new element is less than 20, and if the* `author` *contains "Milli". If so, invoke a SOAP method to notifies a given server with the* `<cdfound>` *inserted CD.*

```
<event> insert(//cd)</event>
<condition> FOR $a IN //cd
            WHERE $a=$new AND $a/price < 20 AND
            contains($a/author,"Milli") </condition>
<action>
  <soap-header> <uri>/notification</uri><host>131.....</host></soap-header>
  <SOAP-ENV:Envelope ...>
    <SOAP-ENV:Body> ... <cdfound> $a//* </cdfound> </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
</action>
```

ACTIVE X-QUERY

A. Bonifati, D. Braga, A. Campi and S. Ceri. Active XQuery. In *ICDE'02*, San Jose (California), February 2002. Also available at: `http://www.icar.cnr.it/angela/pubs/ICDE02.pdf`

- trigger (SQL-like) rules associated with an XML resource
- react after or before changes in the XML data
- changes are insertions, deletions and updates of nodes
- actions are to be executed for each changed node, or only once for a trigger statement
- possible LET clause to bind variables before evaluating the condition
- conditions may be tested by an XQuery WHERE clause
- actions are XQuery-update operations, or external actions.

ACTIVE X-QUERY – SYNTAX

```
CREATE TRIGGER name
[WITH PRIORITY number]
(BEFORE|AFTER) (INSERT|DELETE|REPLACE|RENAME)+ OF XPathExpression+
[FOR EACH (NODE|STATEMENT)]
[XQuery-LET-clause]
[WHEN XQuery-WHERE-clause]
DO (XQuery-UpdateOP|ExternalOp)
```

Note: The LET-clause differs from SQL $\Rightarrow$ variable binding before the condition part.

Variables OLD_NODE and NEW_NODE are available

(PRIORITY to define the execution order of “simultaneous” triggers.)

ACTIVE X-QUERY – EXAMPLE

Whenever a book is inserted for a new author, add the author to the author index.

```
CREATE TRIGGER AddNewAuthor
AFTER INSERT OF document("Lib.xml")//Book
FOR EACH NODE
LET $AuthorNotInList := {
   FOR $n IN NEW_NODE/Author
   WHERE empty(//AuthorIndex/AuthorEntry[Name=$n]) RETURN $n}
WHEN not(empty($AuthorNotInList))
DO ( LET $ai := document("Lib.xml")//AuthorIndex
     FOR $na in $AuthorNotInList
     UPDATE $ai {INSERT
       <AuthorEntry> <Name>{$na/text()}</Name>
                     <PCount>1</PCount>
       </AuthorEntry>} )
```

E-C-A FOR XML

J. Bailey, A. Poulovassilis and P. T. Wood. An Event-Condition-Action Language for XML. In *WWW'02*, pages 486-495. Hawaii, May 2002. Also available at:
`http://www.dcs.bbk.ac.uk/~ap/pubs/wwwTechRep.pdf`

- ECA-rules (re)acting on XML data;
- Events can only be insertions or deletions of (sets of) XML nodes (denoted by a *simple* XPath expression);
- All changes are bound to variable `$delta`;
- Condition and action parts are evaluated separately for each instantiation of `$delta`;
- Conditions are *simple* XPath expression (that return true/false – empty/nonempty) connected by boolean operators;
- Conditions do not bind extra variables;
- Action are sequences of updates of XML data (insertion/deletion of XML nodes)

E-C-A FOR XML – SYNTAX

`ON event IF condition DO actions`

- Event (ON) part:
  - `INSERT` e or `DELETE` e, where e is a (simple) XPath expression;
  - A rule is triggered when a node in e has been inserted (resp. deleted);
  - When a rule is triggered, the variable `$delta` contains the inserted (resp. deleted) nodes in e.
- Condition (IF) part:
  - (Simple) XPath expression connected by `and`, `or` or `not`, or the constant `TRUE`;
  - Truth value of XPath expression determined by (non)emptiness.
- Action (DO) part is a sequence of:
  - `INSERT r BELOW e [BEFORE|AFTER q]` or `DELETE e`, where `r` and `e` are XPath expressions, and `q` an XPath qualifier or `TRUE`.

E-C-A FOR XML – EXAMPLE

On insertion of a new product into store.xml, add this information to products.xml:

```
on INSERT document("store.xml")/stores/store/product
if not(document("products.xml")/products/
              product[@id=$delta/@id]/store[@id=$delta/../@id])
do INSERT <store id='{$delta/../@id}'/>
       BELOW document("products.xml")/products/product[@id=$delta/@id]
       AFTER TRUE
```

The condition and action part is “executed” for each node returned by the event.

RDFTL

G Papamarkos, A. Poulovassilis and P. T. Wood. RDFTL: An Event-Condition-Action Rule Language for RDF. In *HDMS'04*, 2004. Also available at:
`http://www.dcs.bbk.ac.uk/selene/reports/webDyn3.pdf`

- ECA-rules (re)acting on RDF data;
- Same high-level syntax of E-C-A for XML (on ... if ... do ...);
- Variable `$delta` and its meaning as in E-C-A for XML;
- Events are inserting or deleting an instance of a class or a triple, or updating a triple;
- Conditions are RDFTL's Path expressions (path expressions on RDF graphs, with filters) connected by boolean operators;
- Action are sequences of updates of RDF data;
- Updates can be inserting or deleting of a resource, or inserting, deleting or updating an arc.

RDFTL - EXAMPLE

If a new Learning Object in inserted, whose subject is among those of interest for user 128, then add a new arc linking the inserted learning object into the `new_LOs` collection in user 128's personal message:

```
ON INSERT resource() AS INSTANCE OF LearningObj
IF $delta/target(dc:subject) = resource(http://.../users/128)
   /target(ext1:interest)/element()/target(ext1:interest_typename)
DO LET $new_los = resource(http://.../users/128)
                  /target(ext3:message)/target(ext3:new_LOs) IN
   INSERT ($new_los,seq++,$delta)
```

XCHANGE

F. Bry and P.-L. Pătrânjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. In *20th ACM Symp. Applied Computing*. ACM, 2005. Also available at: http://www.pms.ifi.lmu.de/mitarbeiter/patranjan/paper-web-reactivity.pdf

- ECA-like rules for executing (trans)actions or raising other events (in a push manner);
- Events are represented as XML instances, that can be communicated and queried (XChange event messages);
- Possibility of having composite events, via event queries (with operators similar to an event algebra);
- Conditions are queries to persistent data that collect variable bindings;
- The condition part relies on (pattern-based) Xcerpt queries;
- Use of logical variables to communicate values between the parts;
- Actions may be complex updates of Web data (XML or RDF);
- Actions may be executed as transaction (all-or-nothing).

XCHANGE – EVENTS

- atomic events can be event message or updates, system, or transactional (e.g. commit/rollback) events;
- (XML) event messages have sender, recipient, raising/reception time, id parameters, plus the event data;
- Example: detecting a flight cancellation followed (within 2 hours) by the notification that accommodation is not granted.

```
andthen [
    xchange:event {{
        xchange:sender {"http://airline.com"},
        cancellation-notification {{ flight {{ number { var Number } }} }}
                  }},
    xchange:event {{
        xchange:sender {"http://airline.com"},
        important {"Accomodation is not granted!"} }}
        ] within 2 hour
```

XCHANGE – ACTIONS

- Update terms: Xcerpt query augmented with update operations (insertions, deletions, replacements);
- The query term gives a pattern for the data to be modified;
- Complex updates with sequences, conjunctions, disjunctions of updates, and transactions.

- Example: Insert, at the last position of a catalogue, a discount of 10% for all products not of type `New Arrival`:

```
catalogue[[
    product {{
        id { var PId },
        without type { "New Arrival"}
    }},
    at -1 insert discount { products { all var PId },
                            percent { "10" } }
    ]]
```

XCHANGE – RAISE EVENT EXAMPLE

Upon detection of cancellation of flight AI2021 notify the organiser of Mrs. Smith:

```
RAISE xchange:event {
            xchange:recipient {"http://organiser.com/Smith" },
            cancellation-notification { var F }
                    }
ON xchange:event {{
            xchange:sender {"http://airline.com/control-point20/" },
            cancellation {{
                var F -> flight {{ number { "AI2021" },
                                   date { "2005-08-21" }
                                }}
                        }}
                 }}
END
```

XCHANGE – TRANSACTION RULE EXAMPLE

In case the return flight of Mrs. Smith is cancelled, look for another suitable flight, and make that reservation:

```
TRANSACTION
    in { resource { "http://airline.com/reservations/" },
         reservations {{
            insert reservation { var F, name { "Christina Smith" } }
                      }} }
ON xchange:event {{
            xchange:sender { "http://airline.com" },
            cancellation-notification {{
                    flight {{ number { "AI2021" }, date {"2005-08-21" } }}
                                      }}
                  }}
FROM in { resource { "http://airline.com" },
          flights {{ var F -> flight {{ from { "Paris" }, to { "Munich" },
                                        date { "2005-08-21" } }}
                   }} }
END
```