Int. Late-Breaking News Event

Reaction RuleML

Nov. 9th, 2006

Athens, GA, USA

at ISWC’06/RuleML‘06

**Int. Late-Breaking News Event**

# Reaction RuleML

## http://ibis.in.tum.de/research/ReactionRuleML

## Nov. 9th, 17.30 – 18.30

## Athens, GA, USA at ISWC’06/RuleML’06

**Adrian Paschke (Co-Chair Reaction RuleML) and Alexander Kozlenkov (Co-Chair RuleML)**

**Reaction RuleML Technical Group**

RuleML Realize your Knowledge

# Agenda

1. Reaction RuleML: Introduction + News — 15 minutes
2. Reaction RuleML 0.1 — 5 minutes
3. Talks:

| | | |
|---|---|---|
| ECA-LP and Prova Agent Architecture | by Adrian Paschke and A. Kozlenkov | 5 minutes |
| Transaction Logics and Active Rules | by Michael Kifer | 5 minutes |
| Production Rule Programs | by Benjamin Grosof | 5 minutes |
| XChange | by Michael Eckert and Paula Patrajan | 5 minutes |

3. Discussion — 20 minutes

# Reaction RuleML is ...

- An open, general, practical, compact and user-friendly **XML-serialization language** for the **family of reaction rules** including:
  - ECA rules and variants such as ECAP rules and triggers (EA rules)
  - Production rules (CA rules)
  - Active rules (rule execution sequences)
  - Event notification and messaging rules including agent communications, negotiation and coordination protocol rules
  - Temporal event / action and state/fluent processing logics
  - Dynamic, update, transaction, process and transition logics

… but not limited to, due to extensible language design

# Reaction RuleML is intended for e.g., …

- Event Processing Networks
- Event Driven Architectures (EDAs)
- Reactive, rule-based Service-Oriented Architectures (SOAs)
- Active Semantic Web Applications
- Real-Time Enterprise (RTE)
- Business Activity Management (BAM)
- Business Performance Management (BPM)
- Service Level Management (SLM) with active monitoring and enforcing of Service Level Agreements (SLAs) or e-Contracts
- Supply Chain Event Management
- Policies
- …

## … where reaction rules of the various kinds need to be …

- serialized in a homogeneous combination with other rule types such as conditional derivation rules, normative rules, exceptional, default, prioritizied rules or integrity constraints;
- managed, maintained and interchanged in a common rule markup and interchange language;
- internally layered to capture sublanguages such as production rules, ECA rules, event notification rules, KR event/action/state processing and reasoning rules;
- managed and maintained distributed in closed or open environments such as the (Semantic) Web including different domain-specific vocabularies which must be dynamically mapped into domain-independent rule specifications during runtime
- interchanged, translated and executed in different target environments with different operational, execution and declarative semantics;
- engineered collaboratively and verified/validated statically and dynamically according to extensional but also intensional knowledge update actions which dynamically change the behavioral logic of the event-driven rules systems

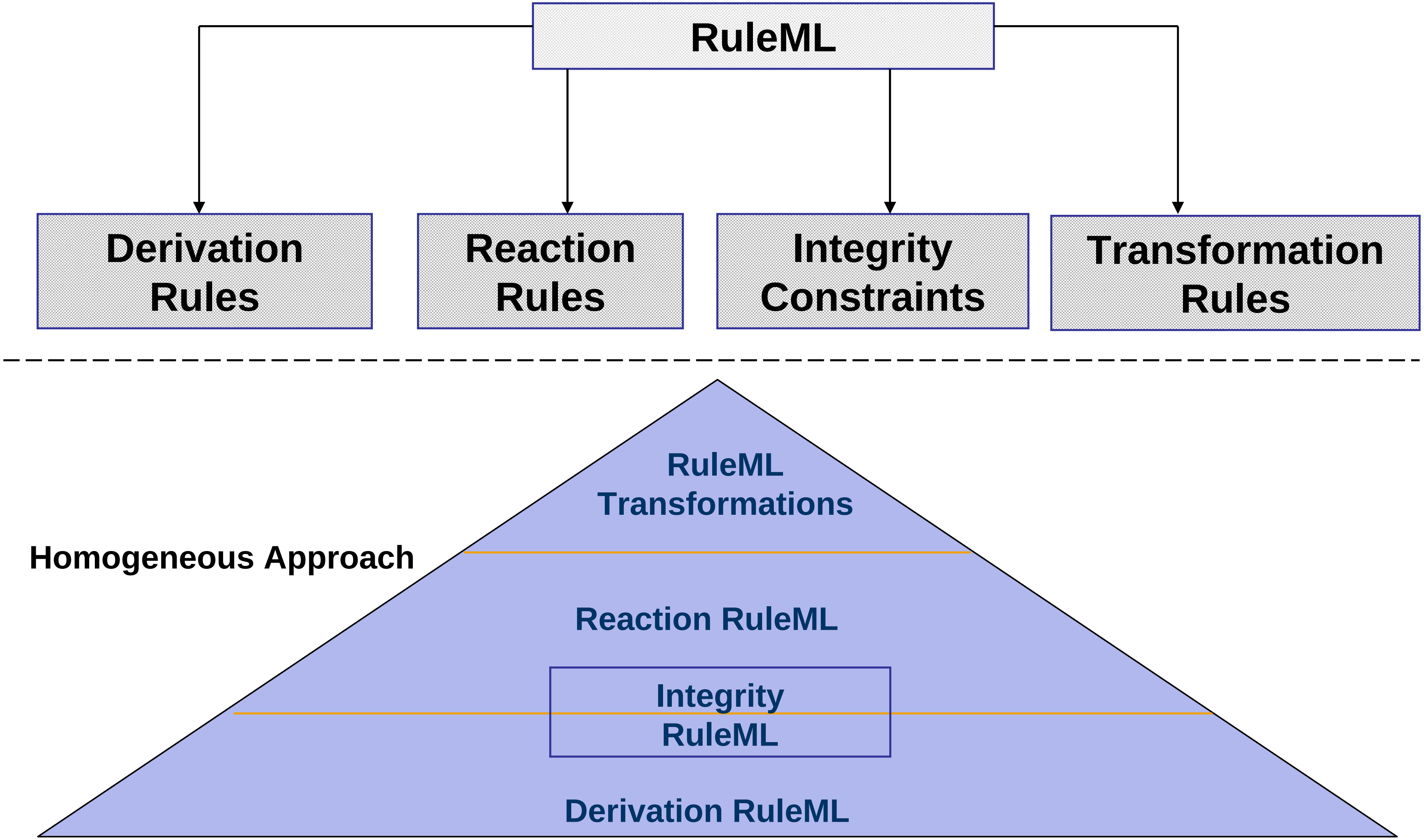
# Our goals are ...

- to enable interoperation between various domains of event/action definition and processing such as:
  - Active Databases, Production Rules Systems, (Multi) Agent Systems, KR Event/Action Logics and Transactional Dynamic Update Logics, Transition and State Process Systems
- to be an general and open intermediate between various "specialized" vendors, applications, industrial and research working groups and standardization efforts such as:
  - OMG PRR
  - W3C RIF
  - Rewerse (e.g. XChange, R2ML, Rewerse ECA)

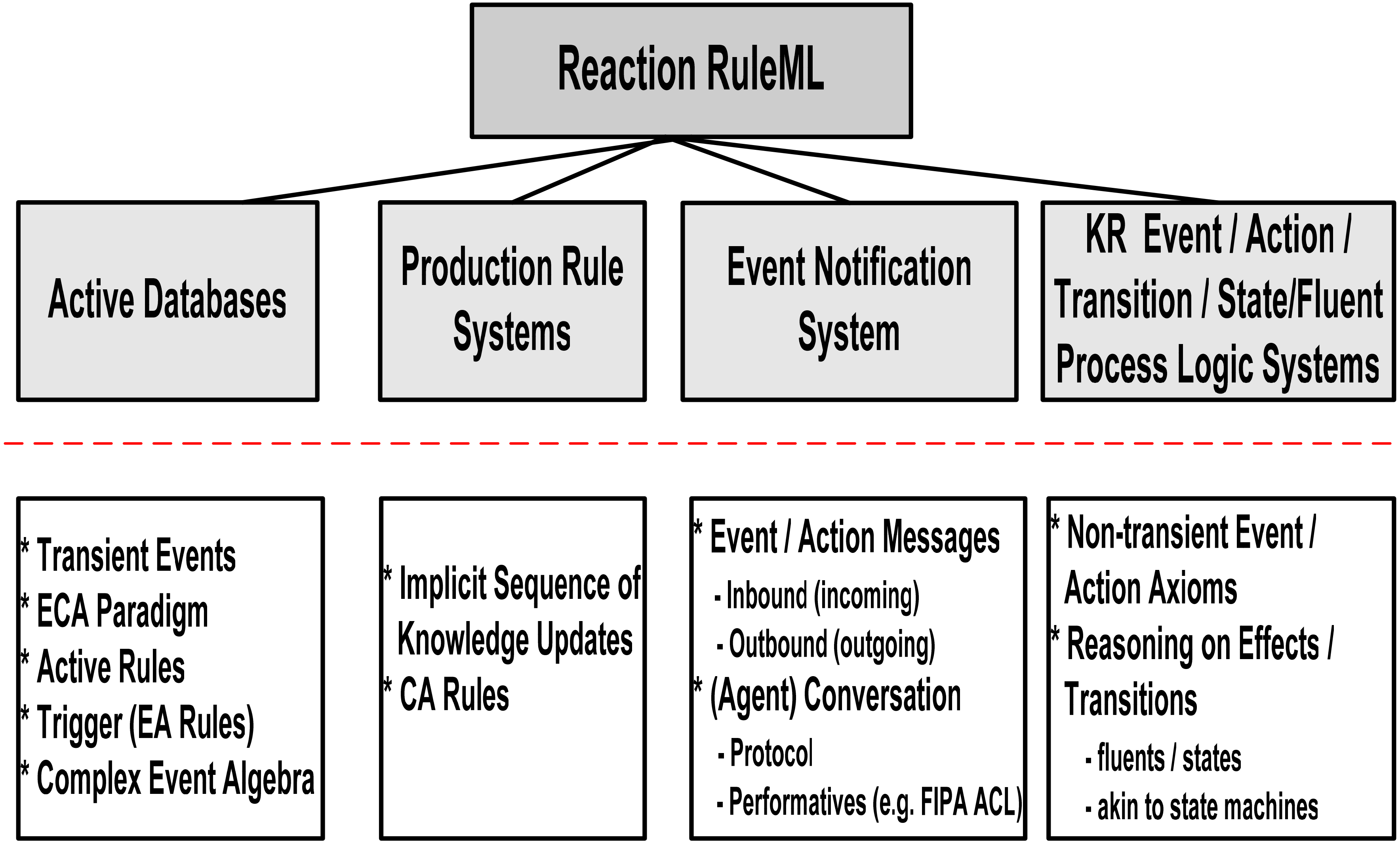
*Reaction RuleML as "GLUE" between different separated approaches on event/action/state definitions and processing/reasoning techniques*

*Bridging the gap between the divergent notations and terminologies via a general syntactical and semantically design*

# How does Reaction RuleML relate to RuleML?



# Scope of Reaction RuleML (1)



# Classification of Event Space – 1. Dimension

- Processing (a.k.a. situation detection or event/action computation resp. reasoning)
  - **Short term**: Transient, non-persistent, real-time selection and consumption (e.g. triggers, ECA rules): ***immediate reaction***
  - **Long term**: Transient, persistent events, typically processed in retrospective e.g. via KR event reasoning or event algebra computations on event sequence history; but also prospective planning / proactive, e.g. KR abductive planning: ***defered or retrospective/prospective***
  - **Complex event processing**: computation of complex events from event sequence histories of previously detected raw or other computed complex event (event selection and possible consumption) or transitions (e.g. dynamic LPs or state machines); typically by means of event algebra operators (event definition) (e.g. ECA rules and active rules, i.e. sequences of rules which trigger other rules via knowledge/state updates leading to knowledge state transitions)
  - **Deterministic vs. non-deterministic**: simultaneous occurred events give rise to only one model or two or more models
  - **Active vs. Passive**: actively detect / compute / reason event (e.g. via monitoring, sensing akin to periodic pull model or on-demand retrieve queries) vs. passively listen / wait for incoming events or internal changes (akin to push models e.g. publish-subscribe)

Derived from: Paschke, A.: **ECA-RuleML: An Approach combining ECA Rules with temporal interval-based KR Event/Action Logics and Transactional Update Logics**, Internet-based Information Systems, Technical University Munich, Technical Report 11 / 2005.

# Classification of Event Space – 2. Dimension

- Type
  - **Flat vs. semi-structured compound data structure/type**, e.g. simple String representations or complex objects with or without attributes, functions and variables
  - **Primitive vs. complex**, e.g. atomic, raw event or complex derived/computed event
  - **Temporal**: Absolute (e.g. calendar dates, clock times), relative/delayed (e.g. 5 minutes after …), durable (occurs over an interval), durable with continuous, gradual change (e.g. clocks, countdowns, flows)
  - **State or Situation**: flow oriented event (e.g. "server started", "fire alarm stopped")
  - **Spatio / Location**: durable with continuous, gradual change (approaching an object, e.g. 5 meters before wall, "bottle half empty" )
  - **Knowledge Producing**: changes agents knowledge belief and not the state of the external world, e.g. look at the program → effect

# Classification of Event Space – 3. Dimension

- Source
  - **Implicit** (changing conditions according to self-updates) vs. **explicit** (**internal** or **external** occurred/computed/detected events) (e.g. production rules vs. ECA rules)
  - **By request** (query on database/knowledge base or call to external system) vs. **by trigger** (e.g. incoming event message, publish-subscribe, agent protocol / coordination)
  - **Internal database/KB update events** (e.g. add, remove, update, retrieve) or **external explicit events** (inbound event messages, events detected by external systems): **belief update and revision**
  - **Generated/Produced (e.g. phenomenon, derived action effects) vs. occurred (detected or received event)**

# Classification of the Action Space (1)

- **Similar dimensions as for events**

- Temporal KR event/action perspective: (e.g. Event, Situation, Fluent Calculus, TAL)
  - Actions with effects on changeable properties / states, i.e. actions ~ events
  - ➔ Focus: reasoning on effects of events/actions on knowledge states and properties

- KR transaction, update, transition and (state) processing perspective: (e.g. transaction logics, dynamic LPs, LP update logics, transition logics, process algebra formalism)
  - Internal knowledge self-updates of extensional KB (facts / data) and intensional KB (rules)
  - Transactional updates possibly safeguarded by post-conditional integrity constraints / test case tests
  - Complex actions (sequences of actions) modeled by action algebras (~event algebras), e.g. delayed reactions, sequences of bulk updates, concurrent actions
  - ➔ Focus: declarative semantics for internal transactional knowledge self-update sequences (dynamic programs)
  - External actions on external systems via (procedural) calls, outbound messages, triggering/effecting

# Classification of the Action Space (2)

- Event Messaging / Notification System perspective
  - Event/action messages (inbound / outbound messages)
  - Often: agent / automated web) service communication; sometimes with broker, distributed environment, language primitives (e.g. FIPA ACL) and protocols; event notification systems, publish / subscribe
  - Focus: often follow some protocol (negotiation and coordination protocols such as contract net) or publish-subscribe mechanism

# Classification of the Action Space (3)

- Production rules (OPS5, Clips, Jess, JBoss Rules/Drools, Fair Isaac Blaze Advisor, ILog Rules, CA Aion, Haley, ESI Logist )
  - Mostly forward-directed non-deterministic operational semantics for Condition-Action rules
  - Primitive update actions (assert, retract); update actions (interpreted as implicit events) lead to changing conditions which trigger further actions, leading to sequences of triggering production rules
  - But: approaches to integrate negation-as-failure and declarative semantics exist:
    - E.g. for subclasses of production rules systems such as stratified production rules with priority assignments or transformation of the PR program into a normal LP
    - Related to serial Horn Rule Programs

# Classification of the Action Space (4)

- Active Database perspective (e.g. ACCOOD, Chimera, ADL, COMPOSE, NAOS, HiPac)
  - ECA paradigm: "*on Event and Condition do Action*"; mostly operational semantics
  - Instantaneous, transient events/actions according to their detection time
  - Complex events: event algebra (e.g. Snoop, SAMOS, COMPOSE) and active rules (sequences of self-triggering ECA rules)

# Classification of the Event / Action / State Definition and Processing / Reasoning Space (1)

1. Event/Action Definition
   - Definition of event/action pattern by event algebra
   - Based on declarative formalization or procedural implementation
   - Defined over an atomic instant or an interval of time, events/actions, situation, transition etc.
2. Event/Action Selection
   - Defines selection function to select one event from several occurred events (stored in an event instance sequence e.g. in memory, database/KB) of a particular type, e.g. "*first*", "*last*"
   - Crucial for the outcome of a reaction rule, since the events may contain different (context) information, e.g. different message payloads or sensing information
   - **KR view**: Derivation over event/action history of happened or future planned events/actions
3. Event/Action Consumption / Execution
   - Defines which events are consumed after the detection of a complex event
   - An event may contribute to the detection of several complex events, if it is not consumed
   - Distinction in event messaging between "multiple receive" and "single receive"
   - Events which can no longer contribute, e.g. are outdated, should be removed
   - **KR view**: events/actions are not consumed but persist in the fact base

# Classification of the Event / Action / State Definition and Processing / Reasoning Space (2)

4. State / Transition Processing
   - Actions might have an internal effect i.e. change the knowledge state leading to state transition from (pre)-condition state to post-condition state.
   - The effect might be hypothetical (e.g. a hypothetical state via a computation) or persistent (update of the knowledge base),
   - Actions might have an external side effect

➔ Separation of this phases is crucial for the outcome of a reaction rule base since typically event occur in a context and interchange context date to the condition or action (e.g. via variables, data fields).

➔ Declarative configuration and semantics of different selection and consumption policies is desirably (also on the syntax layer)

# Design Principles of Reaction RuleML (1)

- XML Schema + EBNF Syntax
- Full RDF compatibility via type and role tags (akin to triple syntax) which can be omitted
- XML Schema Modularization: Layered and uniform design
  - easier to learn the language and to understand the relationship
  - facilitates reusability and complex language assemblings from modules
  - provides certain guidance to vendors who might be interested only in a particular subset of the features
  - easier to maintain, manage and extend in a distributed environment
- Not organized around complexity, but add different modeling expressiveness

# Design Principles of Reaction RuleML (2)

- Reaction RuleML is a **declarative programming language for state / event / action processing rules** and not just a specification language;
  - but might be reduced to it for the business practitioner via predefined functionalities (implemented by a rule engineer and stored in a repository)
- Fulfils typical criteria for good language design such as *minimality*, *symmetry* and *orthogonality*
- Satisfies typical KR adequacy criteria such as *epistemological adequacy* in view of expressiveness of the language
- Reaction RuleML is intended to be transformed into a target execution languages of an underlying rule-based or event/action-driven systems

# Reaction RuleML brings the following benefits …

- Compared to traditional event-driven systems, this approach has the following major advantages:
  - rules are externalized and easily shared among multiple applications (avoiding vendor lock-in) ;
  - encourages reuse and shortens development time;
  - changes can be made faster and with less risk;
  - lowers cost incurred in the modification of business and reaction logic;
  - Allows to continuously adapt the rule-based behavioral logic to a rapidly changing business environments, and overcomes the restricting nature of slow change IT application cycles;

*“Reaction rules constitute the next step in the application of flexible information system (IS) and decision support systems (DSS) technology aimed at automating reactions to events occurring in open service-oriented Web applications (SOAs)”*

# Part II: Reaction RuleML 0.1

# General Concepts (1)

- General reaction rule form that can be specialized as needed
  - Three general execution styles:
    - **Active**: 'actively' polls/detects occurred events, e.g. by a ping on a service/system or a query on an internal or external event database
    - **Passive**: 'passively' waits for incoming events, e.g. an event message
    - **Reasoning**: KR event/action logic reasoning and transitions (as e.g. in Event Calculus, Situation Calculus, ACTL formalizations)
  - Appearance
    - **Global**: ‘globally’ defined reaction rule
    - **Local**: ‘locally’ defined (inline) reaction rule nested in a outer rule
  - Event: event of reaction rule
    - Production rule systems: Event implicit in starting next cycle
    - Active execution: Actively detect / listen to events (possibly clocked by a time function / monitoring schedule)
    - Passive execution: Passively wait / listen for matching event pattern (e.g. event message)

# General Concepts (2)

- Condition
  - Production rule system: trigger for action
  - Backward reasoning: top-down goal proof attempt based on derivation rules
  - **Strong condition**: on failure completely terminates the execution, e.g. the message sequence or the derivation process
  - **Weak condition**: on failure proceeds with the derivation or waits for further messages without execution of the action
- Action
  - Executes action either as internal knowledge self-update or externally, e.g. as sendMessage or procedural call on an external system
- Postcondition
  - Evaluated after action has been performed
  - **Transactional postcondition**: rolls back action (knowledge update) if failed
- Alternative Action
  - Executes alternative action if condition or action fails (akin to “if then else” logic)

# Reaction RuleML Syntax – Basic Constructs

- **<Reaction>** General reaction rule construct
- @**exec** = "*active* | *passive* | *reasoning*"; default = "*passive"*
  - Attribute denoting "active", "passive" or "reasoning" **exec**ution style
- @**kind** Attribute denoting the **kind** of the reaction rule, i.e. its combination of constituent parts, e.g. „*eca*", „*ca*", „*ecap*"
- @**eval** Attribute denoting the interpretation of a rule: "*strong* | *weak*"
- **<event>,<body>,<action>,<postcond>, <alternative>**
  - role tags; may be omitted when they can be uniquely reconstructed from positions
- **<Message>** Defines an inbound or outbound message
- @**mode** = *inbound* | *outbound*
  - Attribute defining the type of a message
- @**directive** = [directive, e.g. FIPA ACL]
- **<Assert>** | **<Retract>** Performatives for internal knowledge updates

… glossary on further constructs such as complex event/action algebra on website

# General Syntax for Reaction Rules

```
<Reaction exec="active" kind="ecapa" eval="strong">

      <event>
               <!-- event -->
       </event>

      <body>
               <!-- condition -->
      </body>

      <action>
              <!--  action -->
      </action>

     <postcond>
              <!-- postcondition -->
     </postcond>

     <alternative>
              <!-- alternative/else action -->
     </alternative>
</Reaction>
```

# Example 1: Active Global Reaction Rule (ECA) (1)

```
<Reaction kind="eca" exec="active">
   <event> <!– the role tag might be omitted -->
      <Reaction kind="ea">
        <event>
           <Atom>
               <Rel>everyMinute</Rel>
             <Var>T</Var>
           </Atom>
        </event>
        <action>
           <Atom>
             <Rel>detect</Rel>
             <Var type="event:EventType1"
                  mode="-">TroubleTicket</Var>
             <Var>T</Var>
           </Atom>
        </action>
      </Reaction>
    </event>
... next slide
```

# Example 1: Active Global Reaction Rule (ECA) (1)

```
    <body>
         <Atom>
            <Rel>maintenance</Rel>
            <Var>T</Var>
         </Atom>
    </body>
     <action>
       <!– Boolean-valued procedural attachment on
           incident management system -->
        <Atom>
             <!-- class/object -->
             <oid><Ind uri="rbsla.utils.TroubleSystem„/></oid>
             <!-- method -->
             <Rel in="effect" lang="java">processTicket</Rel>
             <!-- parameter -->
             <Var type="event:EventType1“
                  mode="+">TroubleTicket</Var>
        </Atom>
     </action>
</Reaction>
```

# Example 2: Active Global Reaction Rule (CA / Production) (1)

```
<Reaction kind="ca" exec="active">
   <body>
        <Atom>
           <Rel>occurs</Rel>
            <Expr in="no">
                 <Fun>heartbeat</Fun><Var>Service</Var>
           </Expr>
           <Var>T</Var>
        </Atom>
   </body>
   <action>
    <Assert>
      <oid><Ind>availability values</Ind></oid> <!– OID of update -->
         <Atom>
              <Rel>alive</Rel>
              <Var>Service</Var>
               <Var>T</Var>
         </Atom>
    </Assert>
   </action>
</Reaction>
```

# Example 3: Passive Global Notification Reaction Rule

```
<Reaction kind="ea" exec="passive" eval="strong">

   <event>
      <Message mode="inbound" directive="ACL:inform">
         <oid><Var>XID</Var></oid>
         <protocol><Var>Protocol</Var>
         <sender><Var>From</Var></sender>
        <content><Var>Payload</Var></content> <!—message payload-->
      </Message>
   </event>

   <action>
        <Assert>
           <oid><Ind>opinions</Ind></oid> <!-- OID of update -->
           <Atom>
                <Rel>opinion</Rel>
                <Var>From</Var>
                <Var>Payload</Var>
           </Atom>
        </Assert>
  </action>
</Reaction>
```

# Web Site Demonstration

http://ibis.in.tum.de/research/ReactionRuleML/

# Part III: Talks

# ContractLog ECA-LP: An Event-Condition-Action Logic Programming Language

by Adrian Paschke

and

Prova Agent Architecture

by Alexander Kozlenkov

## **ECA-LP**: A Homogeneous Event-Condition-Action Logic Programming Language

<u>ECA rule</u>: *eca (<Time>,<Event>, <Condition>, <Action>, <Post-Cond.>, <Else Action>)* *

* All ECA rule parts are optional, except of action; An ECA rule is interpreted as top query

*(Time)*: Pre-conditional time function used as clock / timer

*(Event):* Actively detect/listen to internal and external (complex) events (clocked by time function)

*(Condition)* : Conditional test

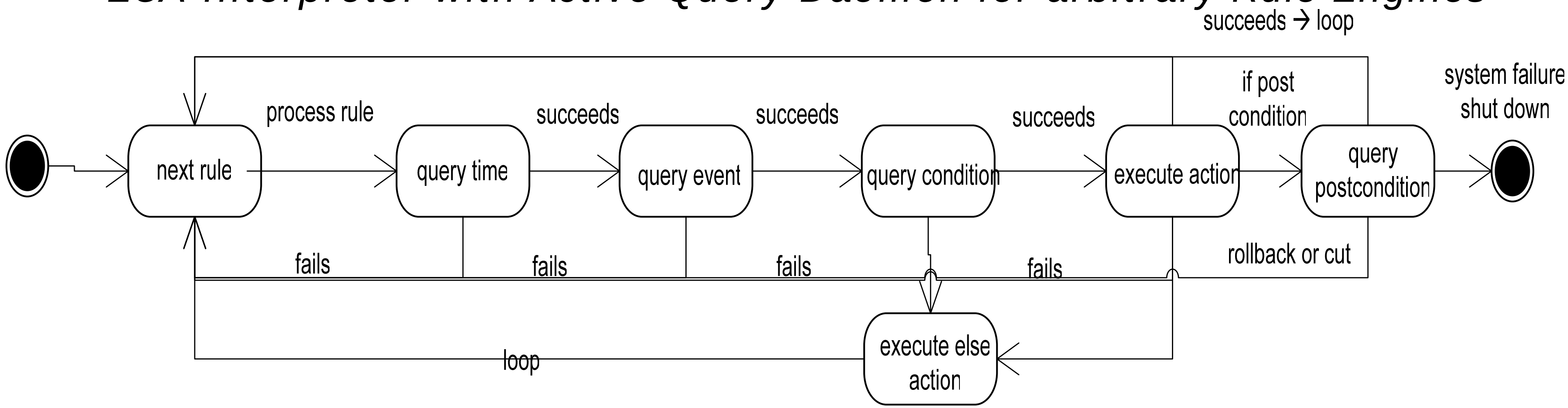
*(Action)*: Internal self-update action or external action with side effects; might be complex and transactional

(*Post-Condition*): Post-conditional test; might commit or rollback action; supports cuts and variable quantifications

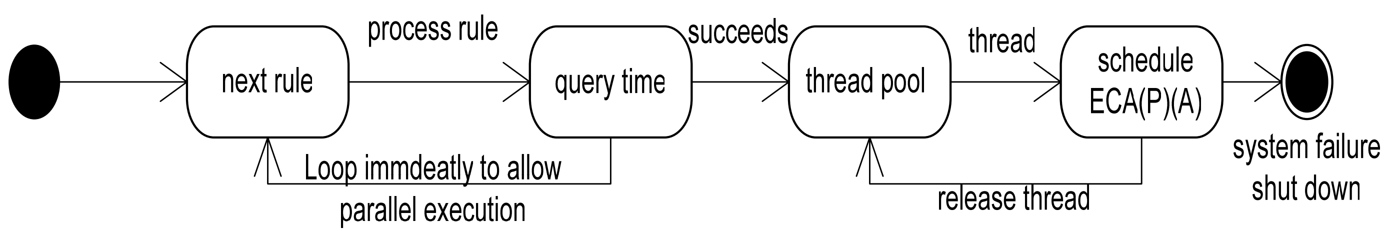
(Else Action): Executes alternative action if condition or action fails (akin to “if then else” logic)

# ECA-LP: Operational Semantics

## *ECA Interpreter with Active Query Daemon for arbitrary Rule Engines*




## Multi-Threading Parallel Scheduling of Reaction Rules



# ECA-LP: Decl. Logic Programming Semantics for ECA rules

- ECA rule is top query: $T \wedge E \wedge ((C \wedge A \wedge P) \vee EL)$ ?.
- Declarative Logic Programming semantics for **PROGRAMMING** of ECA functionalities in terms of derivation rules or Boolean-valued procedural attachments (assigning truth values):
  - Model-theoretic semantics based on 3-valued truth-valued semantics of LP language, e.g. extended WFS: $SEM\ (ECA\ LP) \subseteq MOD_{3\text{-val}}^{Herb}\ (ECA\ LP)$
- Post-conditional integrity constraints and test cases to dynamically test transactional self-update actions and do rollbacks / commits
  - $U^{pos\ /\ neg}_{oid} := \{rule^N : H \leftarrow B, fact_M : A \leftarrow\}\ oid$ , where $N=0,..,n$, $M=0,..,m$ and $oid$=*update label (~module)*
  - $P_i = P_{i-1} \cup U_{oid}^{pos}$ or $P_i = P_{i-1} \setminus U_{oid}^{neg}$ ;
  - Sequence of transitions: $< P,E,U > \rightarrow <P',U,U'> \rightarrow <P'',U',U''> \rightarrow .. \rightarrow <Pn+1,Un, A>$
- Interval Based Event Calculus
  - Rich expressive events/actions definitions
  - State/fluent processing / KR reasoning
  - complex interval-based event / action algebra (KR EC semantics):
    - Paschke, A.: **ECA-RuleML: An Approach combining ECA Rules with temporal interval-based KR Event/Action Logics and Transactional Update Logics**, Internet-based Information Systems, Technical University Munich, Technical Report 11 / 2005.
- 3-Phases for event processing: (1) definition (2) selection (3) consumption
  - Configurable selection and consumption policies
- Transactional complex updates or external actions
  - Dynamic OID-based transactional LP updates
  - Sequence of transitions with post-conditional integrity tests and possible rollbacks
  - External actions with side effects via highly expressive attachments

# Example 1: Active Global Reaction Rule (ECA) (1)

```
<Reaction kind="eca" exec="active">
    <event> <!– the role tag might be omitted if still unambigous -->
       <Reaction kind="ea">
         <event>
            <Atom><Rel>everyMinute</Rel><Var>T</Var></Atom>
         </event>
         <action>
            <Atom>
               <Rel>detect</Rel> <Var type="event:EventType1" mode="-">TroubleTicket</Var>
               <Var>T</Var>
            </Atom>
         </action>
       </Reaction>
    </event>

    <body>
        <Atom>
           <Rel>maintenance</Rel>
           <Var>T</Var>
        </Atom>
    </body>
    <action>
       <!– Boolean-valued procedural attachment -->
       <Atom>
          <oid><Ind uri="rbsla.utils.TroubleSystem„/></oid> <!-- class/object -->
          <Rel in="effect" lang="java">processTicket</Rel> <!-- method -->
          <Var type="event:EventType1" mode="+">TroubleTicket</Var> <!-- parameter -->
       </Atom>
    </action>
</Reaction>
```

# Example 1: Active Global Reaction Rule (ECA) (2)

- ECA-LP/Prova Syntax (related to ISO Prolog notation)

```
eca(
  everyMinute(T),                %time  precond(clock)
  detect(TroubleTicket,T),       % event
  maintenance(T),                % condition
  rbsla.utils.TroubleSystem.ProcessTicket(   % action
      TroubleTicket
  )                                  ).


% Formalization of time function „everyMinute(T)“
everyMinute(T):-
   sysTime(T),   % get actual system time/date
   interval(timespan(0,0,1,0), T).% interval function
```

# Example 1: Active Global Reaction Rule (ECA) (3)

```
% Formalization of event detection
detect(TroubleTicket:event_EventType1,T) :-
      occurs(TroubleTicket:event_EventType1,T),
      consume(TroubleTicket:event_EventType1,T).


% Formalization of condition
maintenance(T) :- neg(holdsAt(maintenance,T).

% Event Calculus state processing rules
initiates(startingMaintenance,maintenance,T).
terminates(stopingMaintenance,maintenance,T).
```

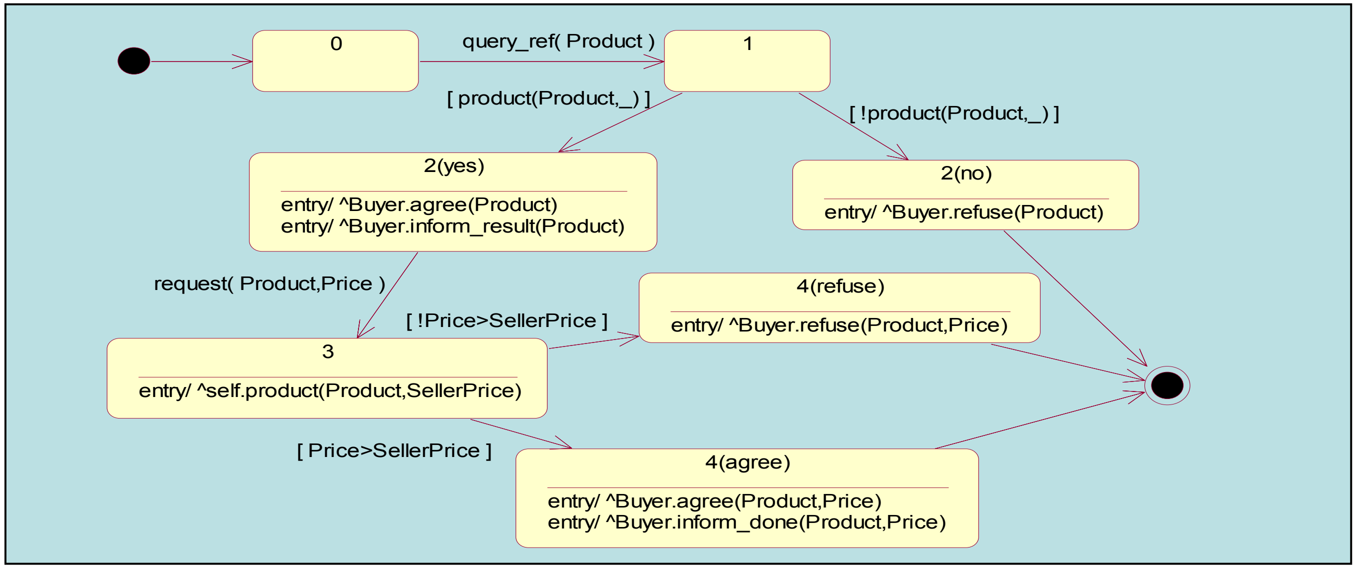
# Agents Architecture Prova-AA

*Prova Agents Architecture is an intrinsic part of the Prova rule language providing reactive agent functionality.*

***Prova-AA*** offers

- ✓ reaction rules;
- ✓ message sending and receiving for local and remote communication actions;
- ✓ uniform handling of communication protocols (JMS, JADE, plus any Mule ESB).
- ✓ message payload as complex terms containing typed or untyped Java variables and serializable Java objects;
- ✓ state machine, Petri nets, or pi-calculus based conversation protocols;
- ✓context-dependent inline reactions for asynchronous message exchange;
- ✓ ability to distribute mobile rulebases to remote agents;
- ✓ *Communicator* class for simplified embedding of Prova agents in Java and Web applications;
- ✓ Prova Universal Message Object gateway for reactive agents on Mule ESB.
- ✓ Multi-threaded Swing programming with Prova Java calls and reaction rules.

# State machines based conversations

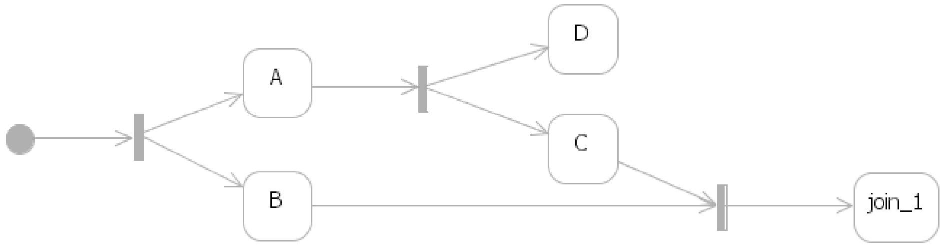



```
directbuy_seller_1(XID,Protocol,From,Product) :-
     product(Product|_),
     !,
     sendMsg(XID,Protocol,From,agree,Product,seller),
     sendMsg(XID,Protocol,From,inform_result,Product,seller),
     directbuy_seller_2(yes,XID,Protocol,From,Product).
directbuy_seller_1(XID,Protocol,From,Product) :-
     sendMsg(XID,Protocol,From,refuse,Product,seller),
     directbuy_seller_2(no,XID,Protocol,From,Product).
directbuy_seller_2(yes,XID,Protocol,From,Product) :-
     !,
     rcvMsg(XID,Protocol,From,request,[Product,Price],buyer),
     product(Product,SellerPrice),
     directbuy_seller_3(XID,Protocol,From,Product,Price,SellerPrice).
directbuy_seller_2(no,XID,Protocol,From,Product).
```

# Prova as a pattern- and rule-based workflow language (1/2)

- State machines cannot represent some business processes involving parallelism;
- More focused workflow languages are required;
- Typical semantics are based on Petri nets and pi-calculus;
- Prova offers
  - unification of message patterns and performatives;
  - *fork* via rule non-determinism;
  - *join* via a built-in predicate with join patterns;
  - conversations distinguished by conversation-ids;
  - easy programmability and ability to run a memory-limited number of conversations in parallel.
  - branching logic with BPEL-like links for selective termination reaction;

# Prova as a pattern- and rule-based workflow language (2/2)




```
process_join() :-
     iam(Me),
     init_join(XID,join_1,[c(_),b(_)]),
     fork_a_b(Me,XID).
fork_a_b(Me,XID) :-
     rcvMsg(XID,self,Me,reply,a(1)),
     fork_c_d(Me,XID).
fork_a_b(Me,XID) :-
     rcvMsg(XID,self,Me,reply,b(1)),
     join(Me,XID,join_1,b(1)).
fork_c_d(Me,XID) :-
     rcvMsg(XID,self,Me,reply,c(1)),
     % Tell the join join_1 that a new pattern is ready
     join(Me,XID,join_1,c(1)).

% The following rule is invoked by join once all the inputs are assembled.
join_1(Me,XID,Inputs) :-
     println(["Joined for XID=",XID," with inputs: ",Inputs]).

% Prints
% Joined for XID=agent@hostname001 with inputs [[b,1],[c,1]]
```



betfair advanced technology group

# Example 3: Passive Global Notification Reaction Rule (1)

```
<Reaction kind="ea" exec="passive" eval="strong">

   <event>
      <Message mode="inbound" directive="ACL:inform">
             <oid><Var>XID</Var></oid>
             <protocol><Var>Protocol</Var>
             <sender><Var>From</Var></sender>
             <content><Var>Payload</Var></content> <!—message payload-->
      </Message>
   </event>

   <action>
        <Assert>
                <oid><Ind>opinions</Ind></oid> <!-- OID of update -->
                <Atom>
                        <Rel>opinion</Rel>
                        <Var>From</Var>
                        <Var>Payload</Var>
               </Atom>
        </Assert>
  </action>
</Reaction>
```

## Example 3: Passive Global Notification Reaction Rule (2)

- Prova AA Syntax (related to ISO Prolog notation)

```
rcvMsg(XID,Protocol,From,“inform",Payload) :-
    add(opinions,"opinion(_0,_1).",[From,Payload]).
```

# Prova AA and ECA-LP Demonstration

# Transaction Logics

by

Michael Kifer

# Production Logic Programs

by

Benjamin Grosof

XChange

by

Michael Eckert and Paula-Lavinia Patranjan

# Part IV: Discussion