NRC Publications Archive Archives des publications du CNRC

Rules Capturing Events and Reactivity

Paschke, Adrian; Boley, Harold

Publisher's version / Version de l'éditeur:

*Handbook of Research on Emerging Rule-Based Languages and Technologies:
Open Solutions and Approaches, 2009-01-01*

NRC Publications Record / Notice d'Archives des publications de CNRC:

<https://nrc-publications.canada.ca/eng/view/object/?id=9212ddc0-c08d-494c-a52c-4482d4e547d6>
<https://publications-cnrc.canada.ca/fra/voir/objet/?id=9212ddc0-c08d-494c-a52c-4482d4e547d6>

Access and use of this website and the material on it are subject to the Terms and Conditions set forth at

<https://nrc-publications.canada.ca/eng/copyright>

READ THESE TERMS AND CONDITIONS CAREFULLY BEFORE USING THIS WEBSITE.

L'accès à ce site Web et l'utilisation de son contenu sont assujettis aux conditions présentées dans le site

<https://publications-cnrc.canada.ca/fra/droits>

LISEZ CES CONDITIONS ATTENTIVEMENT AVANT D'UTILISER CE SITE WEB.

Questions? Contact the NRC Publications Archive team at

PublicationsArchive-ArchivesPublications@nrc-cnrc.gc.ca. If you wish to email the authors directly, please see the first page of the publication for their contact information.

Vous avez des questions? Nous pouvons vous aider. Pour communiquer directement avec un auteur, consultez la première page de la revue dans laquelle son article a été publié afin de trouver ses coordonnées. Si vous n'arrivez pas à les repérer, communiquez avec nous à PublicationsArchive-ArchivesPublications@nrc-cnrc.gc.ca.

National Research
Council Canada

Institute for
Information Technology

Conseil national
de recherches Canada

Institut de technologie
de l'information

NRC · CNRC

Rules Capturing Events and Reactivity.*

Paschke, A., and Boley, H.
2009

* Published at the Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches.

Copyright 2009 by
National Research Council of Canada

Permission is granted to quote short excerpts and to reproduce figures and tables from this report, provided that the source of such material is fully acknowledged.

RULES CAPTURING EVENTS AND REACTIVITY

Adrian Paschke

Harold Boley

Free University Berlin, Germany

National Research Council Canada

ABSTRACT

Event-driven reactive functionalities are urgently needed in present-day distributed systems and dynamic Web-based environments. Reaction rules constitute a promising approach to specify and program such reactive systems in a declarative manner. In particular, they provide the ability to reason over events, actions and their effects, and allow detecting events and responding to them automatically. Various reaction rule approaches have been developed, which for the most part have been advanced separately, hence led to different views and terminologies. This chapter surveys and classifies the wide variety of rule-based calculi approaches, engines and languages for event, action and state processing, and describes their main features. Founded on the original formalisms, major lines of development are traced to the present and extrapolated to the future.

INTRODUCTION AND MOTIVATION

Event-driven applications based on reactive rules and in particular ECA rules, which trigger actions in response to the detection of events, have been extensively studied during the 1990s. Originating from the early days of programming language where system events were used for interrupt and exception handling, active event-driven rules have received great attention in different areas such as active databases, which started in the late 1980s (Paton, 1999) (Widom and Ceri, 1996), real-time applications, and system and network management tools which emerged in the early 1990s, as well as publish-subscribe systems, which appeared in the late 1990s (Baldoni et al, 2003). Recently, a strong demand for reactive functionalities has come from the Web community, where data sources are enriched with reactive behavior to make them active and allow dynamic behavior (as opposed to the conventional query-driven passive Web). Additionally, there is an increased interest in industry and academia in event-driven mechanisms such as Complex Event Processing (CEP) (Luckham 2002) and high-level Event-Driven Architectures (EDA) (Chandy, 2006). (Pro-)active real-time and just-in-time reactions to events and relevant situations are a key factor in upcoming agile and flexible IT infrastructures and distributed service oriented environments. Among many other industry developments, Real-Time Enterprise (RTE), Business Activity Management

(BAM) and Business Performance Management, as well as closely related areas such as IT Service Level Management (SLM) (Paschke, 2007a), are business drivers for this renewed interest.

A great variety of approaches have been developed for reaction rules, which have for the most part evolved separately in their respective domains, and which have led to different views and terminologies. Some detect events as they occur and directly react to them, as e.g. in active databases or implicitly in production rule systems which are fired by changing conditions in the working memory. Others, e.g., KR event / action logics, focus on the development of axioms to formalize the notions of actions and causality, and on the inferences that can be made from the fact that certain events are known to have occurred or are planned to happen in future.

This chapter defines the basic concepts of reaction rules and event processing technologies. It introduces a general multi-dimensional classification scheme for the major families of reaction rule languages, with the four dimensions *event*, *condition*, *action*, and *processing and reasoning* for the classification of the space of (possible) language features. It also gives a survey of the major lines of reaction rule approaches and systems of the past decades. The objectives are:

- to provide a common set of foundational terms and definitions;
- to classify and survey the wide variety of reaction rule languages and event, action and state processing approaches;
- to provide a common vocabulary to describe the language features of different reaction rule languages.
- to provide a common frame of reference as a basis for understanding and analyzing the core properties of a reaction rule language and a rule-based reactive system architecture;
- to support the selection of an appropriate reaction rule approach and language for engineering a rule-based system in a particular application domain.

This chapter is structured as follows: Sections 2 and 3 introduce foundational definitions and a multi-dimensional classification scheme for the major families of reaction rule languages. The central section 4 gives a survey of the main families of reaction rule languages, their core properties and features, and the major developments in each field in the past decades till now. Section 5 highlights future trends in event processing and reaction rule technology. Finally, section 6 concludes this chapter.

FOUNDATIONAL DEFINITIONS

This section introduces the foundational notions and definitions of the reaction rule and rule-based event processing technology.¹

Atomic Event

An atomic event (also raw event or primitive event) is defined as an instantaneous (at a specific point in time), significant (relates to a context), indivisible (cannot be further decomposed and happens completely or not at all) occurrence of a happening.

¹ A related glossary, which is specific to complex event processing, is defined by the Event Processing Technical Society, see (CEP, 2008)

Complex Event

A complex event is composed (composite event) or derived (derived event) from occurred atomic or other complex event instances, as described, e.g., according to the operators of an event algebra or as a result of applying an algorithmic function or process to one or more other events. The included events are called components while the resulting complex event is the parent event. The first event instance contributing to the detection of a complex event is called initiator, where the last is the terminator; all other components are called interiors. Often complex events occur in a *temporal, spatial, state or semantic context* that is often relevant to the execution of the other parts of the reaction rule.

Event Pattern

An event pattern (also event class, event definition, event schema, or event type) classifies the structure and properties of an (atomic or complex) event. It describes, on an abstract level, the essential factors that uniquely identify the occurrence of an event of that type, i.e. its detection condition(s) and its properties with respect to instantiation, selection and consumption.

Event Instance

A concrete instantiation of an event pattern is a specific event instance (also event object or event individual). This can be, e.g., a particular fact becoming true, an external event in a monitored system, a communication event message within a conversation, a state transition such as a knowledge update, a transaction (e.g., begin, commit, abort, rollback), a temporal event (e.g., an alarm raised at 6 p.m., every 10 minutes), etc.

Situation

A situation (aka as fluent or state in KR event/action logics) is initiated and possibly terminated by one or more (complex) events (+ context), and typically holds over a certain time interval, but might also be unbounded. Situation detection in event processing means detecting transitions in the universe of interest that require action, either “reactive” or “proactive”, so that actions can be triggered as a consequence of the detected situation.

Situation Individuals

Situation individuals are discrete individuals of situations with a fixed context allocation of time, date, location, participant, etc. They are not repeatable and are temporally connected. Reification of these situation individuals is possible. Situation individuals relate to complex event instances, i.e. they relate to a state which holds between one or more complex event instances.

Situation Descriptions

Situation descriptions assign content and temporal properties to situation individuals, e.g. dating of situation individuals by mapping them into the context, and connect situation types with validity intervals.

They might be based on differences between situation individuals (sorting, e.g. sequential order of individuals) or based on the differences between situation types with respect to their interaction with the time structure (monotonicity, homogeneity). Different situations might have corresponding properties.

Situation Types

Situation types might have several context allocations (validity intervals) and can be repeated. Validity intervals might not necessarily be sequential and eventually unbounded.

Complex Action

A complex action consists of atomic actions or other complex actions. A complex action might be transactional, i.e. is executed completely (effects are committed) or effects need to be rolled-back.

Action Type

An action type (also action class, action definition, action schema) describes the structure and properties (e.g. transactional) of an (atomic or complex) action, .e.g. the flow of atomic actions of a complex action.

Action Instance

A concrete execution of an action type is a specific action instance (also action object) (whose effects might be rolled-back in case of a transactional action).

Reaction Rule

A reaction rule consists of parts for event/situation processing (e.g. detection, computation), condition verification, action invocation, and post-condition verification, where the condition and (especially) the post-condition parts are optional. The event/situation part can also specialize to the detection of changing conditions, as in a production rule. Accordingly, the most general form of a reaction rule consists of the following parts:

```
on           [event]
if           [condition]
then        [conclusion ]
do          [action]
after       [post-condition]
else        [else conclusion]
elseDo      [else/alternative action]
elseAfter   [else post condition]
```

According to the selected and omitted rule parts, a rule specializes, e.g., to a trigger rule (*on-do*), a production rule (*if-do*), an ECA rule (*on-if-do*) and special cases such as ECAP rule (*on-if-do-after*) or rules with alternative actions or failure compensating actions such as (*if-do-elseDo*), e.g., to send an event message (e.g. to a log system) in case the action execution fails.

Example “Book Flight” Reaction Rule:

*“On an incoming request by a customer to book a flight to a certain destination (**event**), a database look-up tries to select a non-empty list of all flights to this destination (**condition**), and attempts to book the first flight (**action**). In case this action fails, the system will backtrack and book the next flight in the list, etc.; otherwise it succeeds (**post-condition cut**), sending a “flight successfully booked” notification. If no flight can be found or booked to this destination, i.e. the condition or all actions fail, a “no flight could be booked” notification is sent to the customer (**else action**).”*

A reaction rule is triggered when a (complex) event is detected or computed (e.g. a timer which triggers the reaction rule) and/or when a state changes (e.g. in production rules when a new condition is added to the working memory leading to a new state which triggers the rule) and is fired when all rule conditions can be verified, but is rolled-back when any post-condition after the action execution cannot be verified. Typically a reaction rule has an action part, but special forms are also possible, e.g. messaging reaction rules which receive event messages or event calculi formalisms which decouple the axioms which formalize the event occurrence from the axioms describing their effects on the changeable state / situation (see e.g. Event Calculus formalism). Reaction rule languages focusing on processing for the detection of (complex) events are also called *rule-based Event Processing Languages* (rule-based EPLs).

A MULTI-DIMENSIONAL CLASSIFICATION SCHEME FOR REACTION RULE LANGUAGES

This section introduces a multi-dimensional classification scheme for the major families of reaction rule languages, namely

1. (Temporal) KR event/action logics

- Members e.g. Event, Situation, Fluent Calculus, variants based e.g. on Interval Calculus, TAL, etc.
- Actions with effects on changeable properties / states / fluents, i.e. actions are treated similar as events
- Focus: reasoning on effects of events/actions on knowledge states and properties

2. KR evolutionary transaction, update, state transition logics

- Members e.g. transaction logics, dynamic LPs, LP update logics, transition logics,
- Knowledge self-updates of extensional KB (facts / data) and intensional KB (rules)
- Transactional updates possibly safeguarded by post-conditional integrity constraints / tests
- Complex actions (sequences of actions)
- Focus: declarative semantics for internal transactional knowledge self-update sequences (dynamic programs)

3. Process Calculi, Event Messaging and distributed rule-based Complex Event Processing

- Members, e.g. process calculi (CSP, CCS, pi-calculus, join-calculus), event/action messaging reaction rules (inbound / outbound messages), rule-based intelligent CEP with rule-based Event Processing Languages (e.g. Prova, Reaction RuleML, AMIT, Rule Core)
- Focus: process calculi focus on the actions, event messaging and CEP on the detection of complex events; often follow some workflow pattern, protocol (negotiation and coordination protocols) or CEP pattern

4. Condition-Action / Production rules

- Members, e.g. OPS5, Clips, Jess, JBoss Rules/Drools, Fair Isaac Blaze Advisor, ILog Rules, CA Aion, Haley, ESI Logist, Reaction RuleML (including Production RuleML)
- Mostly forward-directed committed-choice / ("don't care") non-deterministic operational semantics for Condition-Action rules
- Focus: primitive actions (assert, retract) and other update actions (assignment etc.), interpreted as implicit events, lead to changing conditions which trigger further actions, themselves leading to sequences of triggering production rules

5. Active Database ECA rules

- Members, e.g. ACCOOD, Chimera, ADL, COMPOSE, NAOS, HiPac, Reaction RuleML, Prova, XChange
- ECA paradigm: “*on Event if Condition do Action*”; mostly operational semantics
- Instantaneous, transient events and actions detected according to their detection time
- Focus: Complex events: event algebra (e.g. Snoop, SAMOS, COMPOSE) and active rules (sequences of self-triggering ECA rules)

For the classification of the space of possible reaction rule language features the scheme defines the five dimensions *general*, *event*, *condition*, *action*, and *processing and reasoning*.

1) General

- **Homogenous vs. heterogeneous language**: same language for all components (event, condition, action) of a rule or different languages
- **Separation of concerns**: different rule parts (event, condition, action) are clearly separated or not (e.g. no explicit distinction between events and actions)
- **Fixed order vs. arbitrary combinations** of event, condition, action rule parts, e.g. ECA rules with fixed event-condition-action paradigm, or messaging reaction rules with arbitrary combinations of inbound events and outbound action messages and intermediate conditions
- **Rules representation language and representation level**: logic base (e.g. propositional, attributive logic, first-order logic, Datalog-like flat logic) vs. operational, procedural vs. process and transition calculi based
- **Variables** in the event and condition can be used in the action part including possible backtracking to different variable bindings

II) Classification of the Event Space

1. Processing

- **Atomic vs. complex event processing:** only support for atomic events or computation of complex events
- **A-priory know event patterns vs. unknown event patterns:** detection and computation of complex event according to known event patterns or detection of unknown event patterns from occurred (complex) events
- **Short term vs. long term:**
 - *Short term- immediate reaction:* transient, non-persistent, real-time selection and consumption of events (e.g. triggers, ECA rules)
 - *Long term- retrospective, deferred, or prospective:* Persistent events, typically processed in retrospective e.g. via KR event calculi reasoning or event algebra computations on an event instance history; but also proactive detection, e.g. aspects of KR abductive planning
- **Real-time vs. any-time:** Real-time reaction rules systems respond to events within stringent timing constraints, where any-time rule systems impose no constraints on the reaction time wrt to the processed events. Often rule-based intelligent CEP systems are real-time reaction rule systems.
- **Qualitative** (e.g. James Allen's interval calculus) vs. **quantitative representations of time**
- **Separation of event definition, selection and consumption:**
 - Event definition: algebraic, temporal logic, event/action calculus, Petri nets, etc.
 - Event selection (from instance sequence): first, last, all, etc.
 - Event consumption: consume once, do not consume, etc.
- **Deterministic vs. non-deterministic:** simultaneous occurred events trigger one rule (deterministic) or more than one rule (non-deterministic), giving rise to only one model ("don't care" non-deterministic, e.g. by priority-based selection of rules) or two or more models ("don't know" non-deterministic)
- **Active vs. passive:** actively query / compute / reason / detect events (e.g. via monitoring, querying / 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)
- **Local vs. global:** events are processed locally within a context (e.g. a process, conversation, branch) or globally (apply global on all rules)

2. Type

- **Flat vs. semi-structured compound data structure/type**, e.g. simple flat representations (e.g. propositional) or complex objects with or without attributes, functions and variables
- **Primitive vs. complex**, e.g. atomic, raw event types or complex event types
- **Homogenous vs. heterogeneous** Sub events are different from the complex event
- **Temporal:**
 - absolute (e.g. calendar dates, clock times),
 - relative/delayed (e.g. 5 minutes after ...),
 - durable (occurs within an interval),

- durable with continuous, gradual change (e.g. clocks, countdowns, flows)
- **State or Situation:**
 - without explicit boundaries
 - Situation State (e.g. “the fire alarm stopped”): selective or interval-oriented relating to point or period; homogenous (the state holds in all subintervals and points), without dynamic change
 - Process (e.g. “he runs”): interval-oriented, restricted homogenous (the process is executed in subintervals), continuous dynamic change
 - Iterative event (e.g. “the alarm rings again and again”): selective or interval-oriented, homogenous
 - with explicit boundaries
 - dynamic change (e.g. the light changes the colour)
 - interval-based (within the next 5 minutes)
 - frequency (e.g. the telephone rung three times)
- **Spatial / Location:** durable with continuous, gradual change (approaching an object, e.g. 5 meters before wall, “bottle half empty”)
- **Knowledge Producing:** changes agents/systems knowledge/belief and not the state of the external world, e.g. look at the schedule → internal effect, e.g. **belief update and revision**

3. Source

- **Implicit** (changing conditions according to self-updates) vs. **explicit 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 at external entities)
- **Generated, produced** (e.g. "spontaneous" phenomenon, derived action effect) vs. **occurred** (e.g. detected or received event)

III) Classification of the Condition Space

- **Logical conditions vs. pattern-based test constraints:**
 - logical conditions act as goals on complex conditional logic represented using (derivation) rules as in, e.g., backward-reasoning logic programming; might support, e.g., variables (new free and bound event variables) + backtracking over different variable bindings, logical connectives (conjunction, disjunction and negations)
 - ground pattern tests, e.g. on changed conditions (~ implicit events in production rules), and pattern matching tests
- **Conditions on the state before/during/after the action change**
- **Scoped conditions:** a scoped condition is evaluated only with respect to an explicitly defined scope (e.g. part of the knowledge base, e.g. a module) or working memory
- **External data integration:** conditions can define constructive views (queries) over external data sources and assign values/objects to variables

IV) Classification of the Action Space

1. Processing

- **Atomic vs. complex action processing:** only support for atomic actions or execution of complex actions (e.g. complex workflows, execution paths)
- **Transactional vs. non-transactional processing:** Transactional complex actions possibly safeguarded by post-conditional integrity constraints / test case tests might be rolled-back in case of failures, or might be committed
- **Hypothetical vs. real execution:** actions have a real effect on the internal (e.g. add new fact) or external world, or are performed hypothetically, i.e. the effect is derived or computed implicitly
- **Concurrent vs. sequential execution:** actions can be processed concurrently (e.g. in parallel branches / threads) or sequentially
- **Local vs. global execution:** actions are executed in a global context or locally in a (workflow) process, scope or conversation state (e.g. outbound action messages in conversations).
- **Variable iteration:** actions iterate over variable bindings of event and conditional variables, possibly by backtracking over binding variants

2. Type

- **Internal knowledge self-update actions** with internal effects on extensional KB or working memory (update facts / data) and intensional KB / rule base (update rules)
- **State actions** with effects on changeable properties / states / fluents, often actions directly relate to events and are treated similarly (as e.g. in KR event and fluent calculi)
- **External actions** with side effects on external systems via calls (by procedural attachments), outbound messages, triggering/effecting
- **Messaging actions:** outbound action messages, usually in a conversational context
- **Process actions:** process calculi and workflow pattern style actions such as join, merge, split, etc.
- **Complex actions** (e.g. delayed reactions, actions with duration, sequences of bulk updates, concurrent actions, sequences of actions) modelled by e.g. action algebras (~event algebras) or process calculi

V) Classification of the Event / Action / State Processing and Reasoning Space

1. Event/Action Definition Phase

- Definition of event/action patterns by event algebra
- Based on declarative formalization or procedural implementation
- Defined over an atomic instant or an interval of time or situation context

2. Event/Action Selection Phase

- Defines selection function to select, e.g. “*first*”, “*last*”, event from several occurred events (stored in an event instance sequence e.g. in memory, database/KB, queue) of a particular type
- 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 Phase

- 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. have outdated timestamps, should be removed
- KR view: events/actions are not consumed but persist in the fact base

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
- Inactive partially executed complex actions should be removed

Remark: The separation of these phases is crucial for the outcome of a reaction rule based system since typically events occur in a context and interchange context data to the condition and action (e.g. via variables, data fields). Declarative configuration and semantics of different selection and consumption policies is desirable.

SURVEY OF REACTION RULE LANGUAGES AND RULE-BASED EVENT, ACTION AND STATE PROCESSING APPROACHES

This section surveys the major fields of reaction rule approaches and discusses the main properties and contributions.

Event/Action Logics, Transition Logics and Process Calculi

Reasoning about events, actions and change is a fundamental area of research in artificial intelligence since the events/actions are pervasive aspects of the world in which agents / systems operate enabling retrospective reasoning but also prospective planning. A large number of knowledge representation (KR) formalisms (event logics, action logics, transition logics) for deductive reasoning but also planning and abductive reasoning about events, actions and change (state transitions) have been developed. The focus of these formalisms is on the development of axioms to formalize the notions of actions as well as events and causality, where events are characterized in terms of necessary and sufficient conditions for their

occurrences and where events/actions have an effect on the actual knowledge states, i.e. they transit states into other states and initiate / terminate changeable properties called **fluents**. Instead of detecting and consuming events as they occur, as e.g. in the active database domain as triggers, the KR approach to events/actions focuses on the inferences that can be made from the fact that certain events are known to have occurred or are planned to happen in future.

The common denominator of all these formalisms and systems is the notion of states, also known as fluents, which are changed or transit due to occurred or planned events/actions. That is, by executing actions (respectively by events that happen) the state of the knowledge changes. Among them are e.g. the event calculus (Kowalski and Sergot, 1986) and variants, the situation calculus (McCarthy and Hayes, 1969), features and fluents (Sandewall, 1989), various (temporal) action languages (TAL) (Gelfond and Lifschitz, 1993) (Fikes and Nilsson, 1971) (Giunchiglia and Lifschitz, 1999) (Giunchiglia and Lifschitz, 1999) (Doherty, 1998), fluent calculi (Hölldobler and Schneeberger, 1990) (Thielscher, 1999) and versatile event logics (Bennett and Galton, 2004). The formalisms differ in:

- How to describe the qualification of events / actions (Qualification)
- How to describe the effects of actions / events (Transitions)
- How to describe indirect effects and interdependencies between fluents (Ramification)
- How to describe which fluents remain unchanged over a transition (Frame Problem)

They also differ in the expressive features they provide such as continuous time, branching states/time, continuous change, causal constraints, concurrent events, durational fluents, events/actions with duration, functional fluents, inertial fluents, state constraints, etc.

Related and also based on the notion of (complex) events, actions (e.g. interaction, communication, and synchronization actions/events) and (process) states, with abstract models for transitions and parallel execution processes, are various process calculi like TCC (Saraswat and Gupta, 1996), CSS (Milner, 1989), ACP (Baeten and Weijland, 1990) and CSP (Hoare, 1985), as well as more recent approaches such as π -calculus (Sangiorgi and Walker, 2001), the ambient calculus (Cardelli and Gordon, 1998), PEPA (Hillston, 1996), fusion calculus (Björn and Parrow, 1998), join calculus (Fournet and Gonthier, 2000), (labeled) transition logics (LTL), and (action) computation tree logics (ACTL) (Meolic, 2000) (Meolic, 2003). In contrast to the event / action logics, the process calculi typically have a focus on the formal modelling of concurrent communicating systems instead of specifying the effects of events/actions on the knowledge state. For instance, action process algebras typically elude the frame problem¹ rather than solving it and have no explicit representation of "situation states" (changeable context-dependent properties of the world) which are initiated or terminated by (complex) events following the "law of inertia".

In the following, the two major event logics namely the Situation Calculus (McCarthy and Hayes, 1969) and the Event Calculus (Kowalski and Sergot, 1986) are described.

The Situation Calculus (SC) (McCarthy and Hayes, 1969) is one of the first KR approaches for formalizing actions and state transitions. It is basically a first-order formalism for describing the world as a sequence of

1 the problem of expressing a dynamic domain in logic without explicitly specifying which conditions *are not* affected by an action

situation changes caused by actions: if an action A occurs in a situation S , a new situation $result(A, S)$ results in which a property P will be true, $holds(P, S)$, (respectively false, $not\ holds(P, S)$) if A in S initiates P , $initiates(A, S, P)$ (respectively, $terminates(A, S, P)$). The frame axioms of the SC are (stated as a simplified Logic Programming formulation):

```
holds(P, s0) :- initially (P) .  
holds(P, result(A, S)) :- initiates(A, S, P) .  
holds(P, result(A, S)) :- holds(P, S), not ( terminates(A, S, P) ) .
```

The SC might be comparable to the Event Calculus (EC); however, unlike the EC, the SC has no notion of an explicit linear time line which in many event-driven systems is crucial, e.g. to define timestamps, timeout/deadline principles, temporal constraints, or time-based context definitions. The major differences between the SC and the EC are:

- branching time in SC vs. linear time in EC
- explicit notion of previous state of the world/situation in SC
- state transitions are functions in SC

Kowalski and Sergot's EC (Kowalski and Sergot, 1986) is a formalism for temporal reasoning about events and their effects on a logic programming system as a computation of earlier events (long-term "historical" perspective). It defines a model of change in which events happen at time points and initiate and/or terminate time intervals over which certain properties (time-varying fluents) of the world hold. Time points are unique points in time at which events take place instantaneously. The basic idea is to state that fluents are true at particular time points if they have been initiated by an event at some earlier time point and not terminated by another event in the meantime. Conversely, a fluent is false at a particular time point, if it has been (never initiated or) previously terminated and not initiated in the meantime. That is, the EC embodies a notion of default persistence according to which fluents are assumed to persist until an event occurs which terminates them. This principle follows the axiom of inertia first proposed by McCarthy and Hayes which says: "Things normally tend to stay the same" (McCarthy and Hayes, 1969). A central feature of the EC is that it establishes or assumes a narrative time structure which is independent of any event occurrences. The time structure is usually assumed or stated to be linear although the underlying ideas can equally be applied to other temporal notions, e.g., branching structures, i.e., event occurrences can be provided with different temporal qualifications. Variants range from the normal structures with absolute times and total ordering to loose ones with only relative times and partial ordering. Given a history of events (set of event occurrences), the EC is able to infer the set of maximal validity intervals (MVIs) over which the fluents initiated and/or terminated by the events maximally hold. Therefore, a central feature of the Event Calculus, in comparison to other formalisms such as Situation Calculus, is that an explicit linear time structure, which is independent of any events, is assumed.

The core axioms of the classical EC are:

```
happens(Ev, Ti) event Ev happens at time point Ti
```

```
initiates(Ev,Fl,Ti) event Ev initiates fluent Fl for all time>Ti
terminates(Ev,Fl,Ti) event Ev terminates fluent Fl for all time>Ti
holdsAt(Fl,Ti) fluent Fl holds at time point Ti
```

The Event Calculus was originally formulated as a logic program in (Kowalski and Sergot, 1986) and many alternative logic program formulations and extensions which also formalize non-deterministic actions, concurrent actions, actions with delayed effects, gradual changes, actions with duration, continuous change, and non-inertial fluents have been subsequently proposed, see e.g. (Shanahan, 1990) (Denecker et al, 1992) (Kowalski, 1992) (Sadri and Kowalski, 1995) (Kakas and Miller, 1997) (Shanahan, 1997a) (Kakas and Miller, 1999). The EC has also been formulated in modal logic; see e.g. (Cervesato et al, 1996). Extensions of the EC with time intervals and event intervals have been proposed (Paschke, 2006a). For instance, the interval algebra, proposed by J. Allen (Allen, 1983), which defines a taxonomy of relations between temporal intervals that is complete for time intervals over a linear set of time points. There are 13 possible relationships between two intervals *I* and *J*:

- I during J* - *I* starts after *J*, and finishes before *J*.
- I overlaps J* - *I* starts before *J*, and *J* ends after *I*
- I starts J* - *I* has the same starting point as *J*, but ends earlier than *J*
- I precedes J* - *I* happens before *J* and there is non-empty interval separating them
- I meets J* - *I* happens before *J*, but there is no non-empty interval between them
- I equals J* - *I* and *J* start and end simultaneously
- I finishes J* - *I* starts after *J*, but has the same end point

The interval relationships are mutual exclusive

Action languages such as *A* (Gelfond and Lifschitz, 1993), *AR* (Giunchiglia and Lifschitz, 1997), *AK* (Son and Baral, 2001), *B*, *C* (Giunchiglia and Lifschitz, 1998) and its successors *C+* (Giunchiglia et al, 2004), *K* (Eiter et al, 2004) and *TAL* (Doherty, 1998) have been developed for declarative reasoning about action and change, used in particular planning problems.

Action language *A* (Gelfond and Lifschitz, 1993) provides comparable expressiveness to the propositional fragment of planning languages such as *STRIPS* (Fikes and Nilsson, 1971) or *ADL* (Pednault, 1994) extended with conditional effects. The main axioms state that an action *A* is executable if the set of all conditional fluents $[F1,..Fn]$ holds and causes a final fluent *L*:

```
executable A if [F1,..Fn].
A causes L if [F1,..Fn].
```

A domain description *D* consists of a set of such propositions, the description of the positive (or negative)

effects of actions $eff(A, S)$, and goal conditions of the form F after $A_1, \dots, A_n$.

The action language AR (Giunchiglia and Lifschitz, 1997) extends A by allowing modelling indirect effects and non-deterministic actions.

The language B instead of *always* in AR uses so called static laws, of the form l if F where a successor state is defined by a minimal consistent set of literals such that all these static laws hold.

Language AK (Son and Baral, 2001) formalizes sensing actions. It provides axioms of the form A determines F , meaning that the value of F is determined by the execution of A .

As in B the language C (Giunchiglia and Lifschitz, 1998) distinguishes dynamic and static laws, but provides more expressiveness with respect to causation laws defined by arbitrary propositional formulae over fluent literals, constraints and qualifications, and supports explicit frame axioms.

$C+$ (Giunchiglia et al, 2004) additionally allows for multi-valued, additive fluents.

The temporal action logics TAL (Doherty, 1998) as descendants of the action languages and the feature and fluents framework, use different language features than the Event Calculus, but provide nearly comparable expressiveness. For instance, indirect effects in TAL are expressed by dependency constraints, whereas EC uses effect constraints and causal constraints. Non-deterministic effects in TAL are represented by disjunctions in reassignment operators. The EC uses special determining fluents.

Process calculi (or process algebras) (Baeten, 2005) are a diverse family of related approaches to formally modelling concurrent systems. Process calculi address the study of the behavior of parallel or distributed systems by algebraic means. They provide algebraic laws that allow process descriptions to be manipulated and analysed, and permit formal reasoning about equivalences between processes. Typical laws are structural or static laws. Atomic actions can be composed into more complicated processes by basic operators, such as $+$ denoting alternative composition, $;$ denoting sequential composition and $\parallel$ denoting parallel composition.

The three most well-known process algebras are CSS (Milner, 1989), ACP (Baeten and Weijland, 1990) and CSP (Hoare, 1985). Historically, CCS was the first with a complete theory. CSP adds a distinguishing equational theory. ACP emphasizes the algebraic aspect: there is an equational theory with a range of semantic models. Also, ACP has a more general communication scheme, whereas in CCS communication is combined with abstraction and in CSP communication is combined with restriction. Many other process algebras and hybrid logic calculi have been developed in the last two decades; each making its own set of choices in the different possibilities such as algebras permitting continuously changing variables other than time or involving differential algebraic equations, and connections with dynamic control theory. Such advanced process calculi are used e.g. in modern ECA rule approaches to specify complex actions (Behrends et al, 2008) or in messaging reaction rules to model the communication behavior of inbound and outbound message links in rules (Paschke et al, 2007b).

Dynamic Logics, Update Logics and Transaction Logics

Dynamic Logic Programs, LP update languages, and transaction logics address updates of logic programs where the updates can be considered as actions that transit the initial program (knowledge state/base) to a

new extended or reduced state hence leading to a sequence of evolving knowledge states. The main aim of these logical update languages is to provide a declarative semantics for logic languages with update actions. Update actions have also been studied in different areas.

Several Datalog extensions such as the LDL language of Naqvi and Krishnamurthy (Naqvi and Krishnamurthy, 1988) which extends Datalog with update operators including an operational semantics for bulk updates or the family of update language of Abiteboul and Vianu (Abiteboul and Vianu, 1991) which has a Datalog-style have been proposed. Further work has been done on adding a notion of state to Datalog programs where updates are modeled as state transitions close to situation calculus.

Reiter (Reiter, 1995) has further extended the situation calculus with an induction axiom specified in second-order logic for reasoning about action sequences. Applying this, Reiter has developed a logical theory of database evolution, where a database state is identified with a sequence of actions.

Transaction Logics (TR) (Bonner and Kifer, 1995) is a general logic of state changes that accounts for database updates and transactions supporting order of update operations, transaction abort and rollback, savepoints and dynamic constraints. It provides a logical language for programming transactions, for updating database views and for specifying active rules, also including means for procedural knowledge and object-oriented method execution with side effects. It is an extension of classical predicate calculus and comes with its own proof theory and model theory. The TR model-theoretic semantics is built on the concept of sequences of database states named paths. Like in modal logic semantics, a structure consists of a set of states and each state represents a database. However, the truths in TR are not defined on arcs between knowledge states, but on execution paths (executorial entailment). The proof procedure executes logic programs and updates to databases and generates query answers, all as a result of proving theorems. TR also has a restricted serial-Horn version for Horn logic programs with update transactions.

Multi-dimensional dynamic logic programming (DyLPs) (Leite et al, 2001), LUPS (Alferes, 2002), EPI (Eiter, 2001), Kabul (Leite, 2003), and ECA-LP (Paschke, 2006b) are all update languages which enable intensional updates to rules and define a declarative and operational semantics of sequences of dynamic LPs. While LUPS and EPI support updates to derivation rules, Kabul is concerned with updates of reaction rules. ECA-LP supports a tight homogenous combination of both reaction rules and derivation rules enabling complex (update) actions and events, transactional updates in both rule types with post-conditional (ECAP) verification, validation, integrity tests (V&V&I) on the evolved knowledge state after the update action(s).

ECA-LP Example:

```
eca (  
    every10Sec(),  
    detect(request(Customer, Destination), T),  
    find(Destination, Flight),  
    book(Customer, Flight),  
    !,  
    notify(Customer, bookedUp(Destination))
```

```
).  
  
% time derivation rule  
every10Sec() :- sysTime(T), interval( timespan(0,0,0,10),T).  
  
% event derivation rule  
detect(request(Customer, FlightDestination),T):-  
    occurs(request(Customer,FlightDestination),T),  
    consume(request(Customer,FlightDestination)).  
  
% condition derivation rule  
find(Destination,Flight) :-  
    on_exception(java.sql.SQLException,on_db_exception()), dbopen("flights",DB),  
    sql_select(DB,"flights", [flight, Flight], [where, "dest=Destination"]).  
  
% action derivation rule  
book(Cust, Flight) :- flight.BookingSystem.book(Flight, Cust),  
    notify(Cust,flightBooked(Flight)).  
  
% alternative action derivation rule  
notify(Customer, Message):-  
    sendMessage(Customer, Message).
```

The example shows an ECA-LP formalization of the “Book flight” reaction rule example given in the section where the term reaction rule was introduced. The example shows how ECA-LP combines reaction rules with derivation rules. Each part in the ECA rule specifies a query on the derivation rules, implementing the respective functionality. Accordingly, the basic declarative ECA rule syntax remains compact and enables reusing the separated, declarative rule specifications several times within different ECA rules. The example includes possible backtracking to different variable bindings. If the action succeeds further backtracking is prevented by the post-conditional cut. If no flight can be found for the customer request, the else action is executed which notifies the customer about this. The condition derivation rule accesses an external data source via an SQL query.

In summary, the update languages described in this section primarily deal with knowledge base updates and transactions but not with general event processing functionalities in the style of ECA rules such as situation detection, context testing in terms of state awareness or external inputs/outputs in terms of event/action notifications or external calls with side effects. However, their declarative semantics for internal (transactional) updates and (knowledge) state transitions forms a basis also for more general reaction rules such as production rules and ECA rules, as implemented e.g. in ECA-LP (Paschke, 2006b).

Production Rule Systems

The classical **production rules system paradigm** in artificial intelligence (AI) research (Hayes-Roth, 1985) and most database implementations of production rules (Declambre and Etheredge, 1988) (Sellis et al,

1993) (Widom and Finkelstein, 1990) (Hanson and Widom, 1993) specify an operational semantics or execution semantics formalizing state changes, e.g. on the basis of a transition system formalism. A production rule is a statement of rule programming logic that specifies the execution of one or more actions in the case that its conditions are satisfied, i.e.: “*if Condition do Action*”.

Two approaches of the operational semantics are distinguished:

- **forward-chaining** order-independent execution (e.g. based on variants of the RETE algorithm (Forgy, 1982)) where each change in the conditions due to update actions such as “*assert*” or “*retract*” on the internal database is considered as an implicit event which can “fire” a production rule hence lead to further update actions etc., resulting in a cycle of firing production rules.
- **Procedural** order-dependent sequential execution in a (compiled) execution environment (rule engine).

There are many forward-chaining implementations in the area of deductive databases (Hanson and Widom, 1993) and many well-known forward-reasoning engines for production rules such as ILOG’s commercial jRules system, Fair Isaac/Blaze Advisor, CA Aion, Haley, and ESI Logist as well as popular open source solutions such as OPS5, CLIPS, Jess, Jeops and Drools, jRules, which are based on variants of the **RETE algorithm** (Forgy, 1982), which was developed by Charles L. Forgy in the late 70s for OPS5 (Official Production System) shell. RETE matches facts against the patterns in rules to determine which rules have had their conditions satisfied. If the matching process occurs only once, the inference engine simply examines each rule and then searches the set of facts to see if the rule’s patterns have been satisfied – if so it is place on the agenda. The matching process takes place repeatedly and the fact list is modified on each cycle of the execution. Such changes can cause previously unsatisfied patterns to be satisfied or vice versa – as facts are added/removed, the set of rules must be updated/maintained. Checking rules against facts each cycle is a slow process. RETE avoids unnecessary computation by remembering what has already been matched from cycle to cycle and computing only necessary changes. The Rete algorithm uses temporal redundancy to save the state of the matching process from cycle to cycle, recomputing changes in this state only for the change that occurred in the fact list. It also takes advantage of structural similarity in rules. In summary, the RETE algorithm keeps the derivation structure in memory and propagates changes in the fact and rule base, i.e. it stores information about the antecedents in a network. In every cycle, it only checks for changes in the networks, which greatly improves efficiency. This algorithm can be very effective, e.g. to just find out what new facts are true or when there is a small set of initial facts and there tend to be lots of different rules which allow to draw the same conclusion. This might be one of the reasons why production rules have become very popular as a widely used technique to implement large expert systems in the 1980s for diverse domains such as troubleshooting in telecommunication networks and computer configuration systems.

While (“pure”) production rules could simulate derivation rules via asserting a conclusion as a consequence of the proved condition, i.e. “*if Condition then assert Conclusion*”, the classical (“full”) production rule languages such as OPS5 lack a clear logical semantics, often suffer from termination and confluence problems of their execution sequences, and typically do not support expressive non-monotonic features such as a kind of negation-as-failure, which can make it hard to express certain real life problems as production rules in a natural and simple way.

Exemplarily, we describe an abstract logical production rule system P in the following, which consists of a language L , a knowledge state S_i which is knowledge base KB consisting of premises¹ and conclusions², and a rule engine R :

{Gro2006}

1. load the KB premises into the KB store which is in state S_n ;

/* (first, clear the working memory and agenda;)

If it is the first time, the KB is in state S_0 and later it will transit from one state to the other*/

2. do loop {

1. pattern match;

/* with respect to (updated) working memory, i.e., find every rule instance whose LHS (left hand side) is satisfied, and that has not yet been fired. This creates an (updated) activation list. */

2. if activation list is empty, then go to step (3.) to finish up.

/* begin firing process: */

3. order the agenda (using the agenda control strategy), and select an agenda-maximal activated rule instance RI_n as next to fire;

/* begin firing that rule instance: */

4. add RI_n 's RHS's asserted facts as derived into the working memory;

/* A derived fact may have already been present in the working memory. If so, just keep one copy of the fact, overall. */

/* begin action launching */

5. add RI_n 's RHS's action calls to the launched-action-call store.

6. perform RI_n 's RHS's action-calls by performing the corresponding action procedural calls;

/* these have side effects */

/* end of action launching */

/* end of firing that rule instance */

7. remove RI_n from the activation list;

/* since it has now been fired */

/* end firing process */ }

3. finish up

1. return the conclusions set from the conclusions store;

/* The conclusion set includes both facts and launched-action-calls. The facts include both premise facts and derived facts. Also, the launched-action-calls when executed had side effects. */

2. stop;

¹ premise facts, production rules, agenda control strategy

² conclusion facts, launched action calls

Semantically, we define this abstract production rule system as follows:

L is used to describe the domain of interest. A state S_i (of the KB) is a snapshot of the world, which is represented as a typical Herbrand interpretation of L . We assume a set of typical update actions (assert and retract) denoted by A and the set of states is denoted by S . The effect (semantics) of actions is described by *effect*: $A \times S \rightarrow S$. A classical production rule is a rule of the form: *if* $l_1, \dots, l_n$ *then* a , where $l_1, \dots, l_n$ are ground literals of L and a is an action in A . A production rule *if* $l_1, \dots, l_n$ *then* a is applicable in state S' iff the conditions $l_1, \dots, l_n$ are true in S_i , i.e. $S_i \models l_1 \wedge \dots \wedge l_n$. A production rule system P is a set of production rules.

An execution E of a production rule system P is a sequence $S_0 \xrightarrow{r_1} S_1 \dots \xrightarrow{r_n} S_n$, where $n \geq 0$, S_i are states, r_i are production rules in P and for each $i \geq 1$, r_i is applicable in S_{i-1} and $S_i = \text{effect}(A(r_i), S_{i-1})$. S_0 is stated as *initial*(E) and S_n is stated *final*(E), where there is no further production rule in P which is applicable in S_n . The semantics I of a production rule system P is defined by: $I(P) = \{(S_0 = \text{initial}(E), S_n = \text{final}(E)) \mid E \text{ is a complete computation of } P\}$.

This core semantics can be further extended to consider non-ground production rule systems, where variables occurring in the rules' conditions are replaced by the ground terms of the matching patterns (unifying conditions) in each state. Formal definitions of the Rete algorithm have been given (Fages and Lissajoux, 1992) (Cirstea et al, 2004), implementation issues related to the algorithm have been widely studied (Albert, 1989) (Tambe and Rosenbloom, 1989) (Miranker, 1990) (Ishida, 1994) (Th'eret, 1994), and applications include, for instance, Petri Nets implementations based on the Rete algorithm (Burdescu and Brezovan, 2001). For a feature comparison of the several common production rule systems see table 1. (W3C-RIF-PRD, 2008)

Features					
	Clips 6.3	Jess 7.1	Drools 4.0.7	JRules	OPSJ
Conditional Elements					
and	x	x	x	x	x
or	x	x	x		x
exists	x	x	x	x	x
not	x	x	x	x	x
forall	x	x	x		x
collect		x(via accumulate)	x	x	?
accumulate		x	x		?
accumulate functions			x		
from			x	x(in)	?
set				x(from)	?
eval	x(test)	x(test)	x(eval)	x(evaluate)	x(?)
Pattern Matching					
slot specific modifies	x(default in COOL)	x			
pattern bindings	x	x	x	x	x
field bindings	x	x	x	x	x
direct accessor		x	x	x	x
nested accessor			x	x	x
lists	x	x	x	x	x
maps	x	x	x	x	x
list pattern matching		x(via test)	x	x	x
no-loop/no-repeat		x(rule)	x(rule+match)	x(Default, rule)	x
consequence keywords					
assert	x	x	x	x	x
retract	x	x	x	x	x
modify	x	x	x	x	x
logical assert (TMS)	x	x	x	x	x

Table 1: Feature comparison of Production Rule systems (W3C-RIF-PRD, 2008)

Most commercial production rule systems implement conflict resolution algorithms such as:

- *Refraction*. The essential idea of *refraction* is that a given instance of a rule must not be fired more than once as long as reasons that made it eligible for firing hold. In other terms, if an instance has been fired in a given state of the system, it is no longer eligible for firing as long as it satisfies all the subsequent states of facts;
- *Priority*. The rule instances are ordered by priority of the instantiated rules, and only the rule instances with the highest priority are eligible for firing;
- *Recency*. The rule instances are ordered by how long a rule instance has been continuously satisfied in the previous states of facts, and only the most recent ones are eligible for firing.

Several extensions to the core production systems paradigm have been made which introduce e.g. negations (classical and negation-as-finite failure) (e.g. Dung and Mancarella, 1996). They provide (operational or formal declarative) semantics for such expressive constructs, using e.g. priorities (Agrawal et al, 1991), denotational semantics (Widom, 1992), bottom-up magic transformation (Pang, 1994), stratified production rules (Raschid, 1992), argumentation (Dung and Mancarella, 2002), transformation of update programs into logic programs (Raschid, 1996), production logic programs (Grosz, 2005), transactional update logic (Paschke, 2006c). Closely related are also logical update languages such as transaction logics (Bonner and Kifer, 1995) and in particular serial Horn programs, where the serial Horn rule body is a sequential execution of actions (assert/retract) in combination with standard Horn pre-/post conditions. These serial rules can be processed top-down or bottom-up and hence are closely related to the production rules style of “*if condition do update action*”.

Active Databases and ECA Rule Systems

Event-driven applications based on reactive rules and in particular Event-Condition-Action rules (**ECA**), which trigger actions as a response to the detection of events, have been extensively studied during the 1990s in active databases. (Widom and Ceri, 1996) Many commercial databases systems have been extended to allow the user to express **active rules** whose execution can update the database and trigger the further execution of active rules leading to a cascading sequence of updates (often modelled in terms of execution programs). Triggers are supported by the SQL3 standard (Kulkarni et al, 1999). A trigger in SQL3 is an ECA rule that is activated by a database state transition and has an SQL3 predicate as a condition and a list of SQL3 statements as action. Triggers are supported by the major commercial databases such as Oracle, DB2, Sybase, Informix. Active databases are also an important research topic due to the fact that they provide foundational groundwork for many modern complex event processing technologies and find many applications in real world systems.

Active databases in their attempt to combine techniques from expert systems and databases to support automatic triggering of rules in response to events and to monitor state changes in database systems have intensively explored and developed the ECA paradigm and **event algebras** to compute complex events. In a nutshell, the ECA paradigm states that an ECA rule autonomously reacts according to actively or passively detected simple or complex events by evaluating a condition or a set of conditions and by executing a reaction whenever the event happens and the condition(s) is (are) true: “*On Event if Condition do Action*”. Variants of ECA rules are e.g. EA rules (triggers) or ECAP rules (reactive rules with post-conditions after action execution). Event algebras provide the operators to define complex event types. Typical event algebra operators are, e.g.:

- *Sequence operator* (;): the specified event instances have to occur in the order determined by this operator
- *Disjunction operator* ($\vee$): at least one of the specified instances has to occur
- *Conjunction operator* ($\wedge$): the specified event instances can occur in any order, where the detection time is the timestamp of the latest occurred event
- *Simultaneous operator* (=): the specified instances have to occur simultaneously
- *Negation operator* ($\neg$): the specified instance(s) are not allowed to occur in a given interval
- *Quantification* (Any): the complex event occurs when n events of the specified type have occurred
- *Aperiodic Event Operator* (Ap): The aperiodic operator A allows one to express the occurrence of an event $E2$ within the interval defined by two other events $E1$ and $E3$: $Ap(E2, E1, E3)$
- *Periodic Event Operator* (Per): The periodic operator Per is an event type which occurs every t time-steps in between $E1$ and $E2$

Often a distinction into the *event definition*, *event selection*, and *event consumption* is made (see section “Dimensions for Reaction Rule Languages”). Selection policies are, e.g., “select all events of a particular type”, “select the first”, “select the last”, which contribute to the detection of a complex event. Different consumption policies are, e.g., “remove all events which belong to a particular type” or “remove the first respectively the last event”, “do not consume”, etc. Similar to event algebras, action algebras provide operators to define complex actions, e.g.:

- *Succession*: Ordered Succession of Actions
- *Choice*: Non-Deterministic Choice
- *Flow*: Parallel Flow
- *Loop*: Loops

Several event algebras have been developed, e.g. Snoop (Chakravarthy, 1994), SAMOS (Gatzju and Dittrich, 1993), ODE (Gehani et al, 1992): The object database ODE implements event-detection mechanism using finite state automata. SAMOS combines active and object-oriented features in a single framework using colored Petri nets. Associated with primitive event types are a number of parameter-value pairs by which events of that kind are detected. SAMOS does not allow simultaneous atomic events. Snoop is an event specification language which defines different restriction policies that can be applied to the operators of the algebra. Complex events are strictly ordered and cannot occur simultaneously. The detection mechanism is based on trees corresponding to the event expressions, where primitive event occurrences are inserted at the leaves and propagated upwards in the trees, as they cause more complex events to occur. Several papers discuss formal aspects of active databases on a more general level – see e.g. (Paton, 1995) (Zimmer and Unland, 1999).

Based on event algebras several active database systems and prototypes have been developed, e.g. ACCOOD (Erikson, 1993), Chimera (Meo et al, 1996), ADL (Behrends, 1995), COMPOSE (Gehani et al, 1992), EXACT (Diaz and Jaime, 1997), REACH (Zimmermann and Alejandro, 1999), ROCK&ROLL (Dinn et al, 1996), REFLEX (Naqvi and Ibrahim 1994), NAOS (Collet and Coupaye, 1996), HiPac (Dayal et al, 1996) or JEDI (Cugola et al, 1998). Other prototypes based on interval-based event / action logics (e.g. ECA-LP (Paschke, 2006c)), functional programming (e.g. JEM (Guangtian et al, 1998)) and real time logic (RTL) (e.g. PFL (Reddi et al, 1999)), and ADL (Behrends, 1994) exist.

There has been a lot of research and development concerning knowledge updates in active rules (execution

models) in the area of active databases and several techniques based on syntactic (e.g. triggering graphs (Aiken et al, 1994) or activation graphs (Baralis and Widom, 1994)) and semantics analysis (e.g. (Baley, 1997) (Widom, 1992) (Paschke, 2006b)) of rules have been proposed to ensure termination of active rules (no cycles between rules) and confluence of update programs (always one unique minimal outcome). Several approaches in the active database domain also draw on transformations of active rules into LP derivation rules, in order to exploit the formal declarative semantics of LPs to overcome confluence and termination problems of active rule execution sequences. The combination of deductive and active rules has been investigated in different approaches, e.g. based on transformation or simulation of active rules by means of deductive rules (Dietrich et al, 2003) (Flesca and Greco, 1998) (Lausen et al, 1998) (Zaniolo, 1993). Moreover, there are approaches which directly combine reactive rules with deductive rules such as the Event-Condition-Action Logic Programming language (ECA-LP) which enables a homogeneous representation of ECA rules and derivation rules (Paschke, 2006c). In the following, we exemplarily introduce the ID-based transactional transition semantics of ECA-LP:

A positive (add) resp. negative (remove) ID-based update to a rule program P is a finite set

$U_{oid}^{pos/neg} := \{rule^N : H \leftarrow B, fact^M : A \leftarrow\}$, where $N=0,...,n$, $M=0,...,m$ and oid denotes the label of the update, i.e. the unique object id with which it is managed in the KB. Applying U_{oid}^{pos} resp. U_{oid}^{neg} to P leads to the extended knowledge state $P^+ = P \cup U_{oid}^{pos}$ resp. reduced state $P^- = P \setminus U_{oid}^{neg}$. Applying arbitrary sequences of positive and negative updates leads to a sequence of program states $P_0, ..., P_k$ where each state P_i is either $P_i = P_{i-1} \cup U_{oid_i}^{pos}$ or $P_i = P_{i-1} \setminus U_{oid_i}^{neg}$. In other words, a program P_i , i.e. the set of all clauses in the KB at a particular knowledge state i , is decomposable in the previous knowledge state plus/minus an update, whereas the previous state consists of the state $i-2$ plus/minus an update and so on. Hence, each particular knowledge state can be decomposed in the initial state P_0 and a sequence of updates. ECA-LP defines $P_0 = \emptyset \cup U_0^{pos}$ and $U_1^{pos} = \{P : \text{the set of rules and facts defined in the initial LP } P\}$, i.e. loading the initial rule program into the KB denotes the first update.

Based on this the semantics of updates is defined as a **sequence of transitions**

$\langle P, E, U \rangle \rightarrow \langle P', U, U' \rangle \rightarrow \langle P'', U'', U'' \rangle \rightarrow ... \rightarrow \langle P^{n+1}, U^n, A \rangle$, where E is an initiating event which triggers the update action U of an ECA rule (in case the ECA condition is satisfied) and transits the initial knowledge state P into P' . The update action U might be a further sub-event in another ECA rule(s) (active rule) which triggers another update, leading to a sequence of transitions which ends with a terminating action A . A might be e.g. an update action which is not an input event for further updates or an action without an internal effect on the knowledge state, e.g. an external notification action.

To overcome arising conflicts, preserve integrity of the KB in each state and ensure a unique declarative outcome of update sequences (active rules) or complex updates ECA-LP extends the semantics of updates **to transactional updates** which are safeguarded by **integrity constraints (ICs)**. ICs are used to verify and validate the actual or any hypothetically future knowledge state. A transactional update is an update, possibly consisting of several atomic updates, i.e. a sequence of transitions, which must be executed completely or not at all. In case a transactional update fails, i.e. it is only partially executed or violates integrity, it will be rolled back otherwise it will be committed. ECA-LP defines a transactional update as $U_{oid}^{trans} = U_{oid_1}^{pos/neg}, ..., U_{oid_n}^{pos/neg} \& C_1, ..., C_m$, where C_i is the set of ICs which are used to test integrity and verify and validate the intended models of the updated clauses/modules. In short, ICs are defined as constraints on the set of possible models and describe the model(s) which should be considered as strictly conflicting. The ICs are used as post-conditions to test the updated program state $P_{i+1}^{hypo} = P_i \cup U_{oid}^{trans, pos}$ resp. $P_{i+1}^{hypo} = P_i \setminus U_{oid}^{trans, neg}$ and rollback the (possibly partial executed) transactional update to the state $P_{i+1} = P_i$ or commit the update

$P_{i+1} = P_{i+1}^{hypo}$ in case the update preserves integrity wrt to the defined ICs. Hence, if there exists a satisfied IC in the hypothetical/pending state P_{i+1}^{hypo} the transactional update is rolled back:

$$P_{i+1} = P_i \cup U_{oid}^{trans^{pos}} \Rightarrow P_{i+1}^{hypo} \setminus U_{oid}^{trans^{pos}} \text{ iff exists } P_{i+1} \models IC_j, j=1, \dots, n$$

$$P_{i+1} = P_i \setminus U_{oid}^{trans^{neg}} \Rightarrow P_{i+1}^{hypo} \cup U_{oid}^{trans^{neg}} \text{ iff exists } P_{i+1} \models IC_j, j=1, \dots, n$$

A commit is defined as $P_{i+1}^{hypo} \Rightarrow P_{i+1}$.

In contrast to event notification systems and distributed rule-based complex event processing (see next section), the ECA rules in active databases are typically defined with a global scope (global state) and react on internal events of the reactive system such as changes (updates) in the active database. The ECA view and terminologies for events and actions also differ from those of KR event/action logics. The focus in the logical formalisms is on the development of axioms to formalize the notions of actions/events and causality, where events/actions are characterized in terms of necessary and sufficient conditions for their occurrences. In contrast, the main aim of ECA rules is immediate reactions on detected events as they occur in active databases. Events are typically handled in real-time and consumed afterwards (transient view).

Rule-based Complex Event Processing and Event Notification Systems

In distributed environments such as the Web and enterprise service networks, with independent service or agent nodes and open or closed domain boundaries, **complex event processing (CEP)** (Luckham 2002) is increasingly done using high-level **event notification and communication mechanisms** based on middleware such as **enterprise service bus (ESB)**, **object request brokers (ORBs)**, and **message oriented middleware (MoM)**. The idea of “middleware” is to place a layer of software between the application layer and the network layer to facilitate application-to-application programming and hide the underlying networking details. In the 1970's one of the first middleware was remote procedure call (RPC). By the end of the 1980's object request brokers (ORBs) were based upon RPC. Some years later, the message-oriented middleware (MOM) approach was introduced based upon decoupled communication by means of events using message queues (MQ) and protocols like e.g. publish-subscribe protocol for coordinating requests for service. Enterprise Service Bus (ESB) is a high level application-to-application communications bus, analogous to the hardware bus that carries signals between components in a computer. Modern commercial ESBs bundle together a lot of event driven messaging, often including Web services, e.g. Rest, SOAP and XML messaging. Some ESBs go as far as to say they offer CEP in the form of event interpretation, correlation and pattern matching, and together with expressive rule-based decision and reaction layers for **intelligent rule-based CEP**, e.g. Rule Responder (<http://responder.ruleml.org/>). Much of this is playing into the currently fashionable market for **event-driven BPM (EDBPM)**, **semantic BPM (SBPM)**, and service oriented architecture (SOA), with a trend towards combination with **event-driven architectures (EDA)**. Typically, the interest here is in the particular and complex event sequence or event processing workflow which possibly follows a certain communication or workflow-like coordination protocol (Paschke et al, 2006e), rather than in single global event occurrences which trigger immediate reactions as in the active database trigger or ECA rules. As a result, reactive rules which build on (complex) events are typically local to a particular context, e.g. within a particular conversation and coordination protocol (e.g. a workflow). That is, the communicated events contribute to the detection of a complex event situation which triggers a local reaction within a context (e.g., a conversation waiting for at

least three sequential answers or requests). Rule-based CEP systems are typically real-time reaction rule systems which need to react to events within stringent timing constraints¹. To fulfill these real-time constraints often parallel (distributed) inference mechanisms are used. Real-time reaction rule system and real-time rule-based Event Query Language (EQL) have been investigated e.g. by (Cheng et al, 1993).

In general, **Complex Event Processing (CEP)** (Luckham 2002) is an emerging enabling technology to achieve actionable, situational knowledge from distributed systems in real-time or quasi-real-time. Detecting complex event patterns (a.k.a. complex event types) and situations (complex events + conditional contexts), i.e. detecting transitions in the universe of interest (a so-called **event cloud**) that requires action either “reactive” or “proactive” is one of the basic ideas of the CEP technology. A complex event pattern describes the detection conditions of a complex event which is derived from low-level events that occur (real-time view) or have happened (retrospective view) in specific combinations. This process of event selection, aggregation, hierarchizing, and event abstraction for the generation of higher, business-level events is called “Complex Event Processing”. The evolving CEP discipline mostly comprises technologies from previous event processing research fields.

Stemming from the operational level of network management and IT infrastructure management (ITIM) event correlation systems / engines such as IBM Tivoli Event Console (IBM, 2001), HP OpenView Event Correlation Services (Sheers, 1996), VERITAS NerveCenter (Veritas, 1999), SMARTS InCharge (Yemini et al, 1996), and the ongoing work on realizing the Common Event Infrastructure (IBM, 2003), are designed to primarily handle network events and events on the IT infrastructure level.

The event languages from active database systems (see previous section) support event algebra operators to describe complex event types that can be applied over the event clouds and event instance histories in real time. The active mechanisms in these systems serve to detect and react to updates in the database that may jeopardize the data integrity, or to execute some business-related application logic.

A closely related area is sequence databases which extend the database query languages such as SQL with means to search for sequential patterns. Examples are the time series systems, such as the analytic functions in Oracle (Oracle, 1998) (Oracle, 1999) or the time-series data blades in Informix (IBM, 2001b). Sequences are supported as user-defined types (UDF) and stored in tuple fields. Users are provided with a library of functions that can be invoked from SQL queries. However, the functions provided are rather general and do not support a more complex and dedicated class of queries. Another approach seeks to extend SQL to support sequenced data. Such extensions were first proposed for the PREDATOR system (SEQUIN (Shadri et al, 1994)). SRQL (Ramakrishnan, 1998) extended the relational algebra with sequence operators for sorted relations, and added constructs to SQL for querying sequences. Then, more powerful extensions for pattern recognition over sorted relations were proposed for SQL-TS (Sadri, 2001a) (Sadri, 2001b). It also proposed techniques for sequence queries optimizations. A recent approach extending SQL is the Continuous Query Language (CQL) (Motwani et al, 2003) which is a declarative query language for data streams in Data Stream Management Systems. CQL is developed in the STREAM project at Stanford University. In summary, these works are aimed at finding more general solutions for sequence data. They do not take into account the specifics of the event data.

Figure 1 introduces a high-level reference architecture describing an abstract view of a CEP architectural

¹ as opposed to any-time reaction rule systems (see classification scheme for reaction rules)

reference model. (Paschke, 2008)

Figure 1: CEP Reference Architecture

The model describes four different roles:

Event Originator

The Event Originator or Event Producer is a possibly distributed set of event sources in which events – typically atomic but also higher-level events – occur. Event sources are for instance processes, workflows, services, applications, etc.

Event Modeler

The event modeler describes the event pattern definitions, i.e. the modeler implements the event types whose event instances (concrete events occurred/produced in the event source or those that should not occur) are needed to trigger the complex events which should be detected and processed in the CEP media. The modeler also specifies the interface between the CEP media and the CEP sources.

Event Processing Media

Event Processing Media are (distributed) platforms where events are processed. In particular, a CEP media is a device with electronic channels that process and transport event data. That is, the CEP media comprises the technical infrastructure supporting event selection, aggregation, hierarchizing, and event abstraction for generating higher-level complex events of interest.

Event Consumer

The event consumer consumes, uses, and possibly reacts to the relevant complex events and detected situations which are communicated to it by the event processing media. Through its reactions, an event consumer can create further events, hence also act as an event originator for further event processing flows in the event processing media.

Remark: CEP media systems either directly interoperate and process event data, e.g. event database management systems using e.g. CQL on top of Oracle databases with time series and data stream, or they communicate events and event queries (e.g. on event streams) in terms of messages using a particular transport protocol/language such as JMS, HTTP, SOAP, etc., sometimes according to a predefined or negotiated communication/coordination protocol, or they might subscribe to publishing event notification servers which actively distribute events (push) to the subscribed and listening rule-based agents (publish-subscribe). As the communication backbone for the CEP media, e.g. an enterprise service bus infrastructure may be used for asynchronous or synchronous interchange. The later message-oriented middleware approach is sometimes also referred to as **Event Processing Network (EPN)** (CEP, 2008): A set of event processing agents (EPAs) and a set of event channels connecting them, where

1. the set of EPAs can be dynamic, i.e., EPAs can be created and destroyed,
2. the set of channels can be dynamic, i.e., channels can be created and destroyed,
3. dynamic behavior may controlled by patterns of events occurring in the network.
4. An EPN need not be an acyclic directed graph, e.g., feedback loops (cycles) are possible.
5. The runtime deployment of an event processing network may be distributed across multiple physical networks, computers and software artifacts.
6. Various graphical representations of EPNs are possible.

On the basis of the presented CEP reference architecture, more concrete application-specific reference architectures for the different layers – from a technical view to a business view on CEP media (see figure 2) – can be derived, along different levels of abstraction, or “views,” thereby providing more flexibility in how a domain-specific reference architecture can be used.

Figure 2: Layers of CEP Media

The different phases of the life cycle of events processed by a CEP media are illustrated in figure 3.

Figure 3: Event Process Life Cycle in CEP Media

The life cycle spans from the event production (1) to the event definition (2), addressing how relevant complex event types can be specified unambiguously; further, to the event selection (3) and aggregation (4), which provides means to select the important (raw) events from huge event clouds in almost real-time and compute/detect complex event instances according to the specified event patterns from these (raw) event instance sequences; it continues with the event handling (5), which supports rating of the detected complex events in the context of situations triggered by them, including also their prediction; it finally ends with the complex event consumption (6), where the event consumer manages and operates situational, e.g. by automated decisions or reactions on the provided complex event information, possibly creating new events which start a new CEP life cycle in the CEP media. In this life cycle complex events may exist in reality, i.e. become persistent e.g. because they are asserted to a database or are interchanged as one complex event in a message, or they are only abstractions (non-materialized constructive views), where the complex event (instance of a particular event type) of an event definer is seen as a group of atomic events by the event consumer. (cf. Nominalism in philosophy ; complex events can be made persistent (treated as new events))

Various Event Processing Languages (EPLs) for CEP have been developed in the last years. There are SQL-like EPLs such as CQL (Motwani et al, 2003), rule-based EPLs such as Reaction RuleML (Paschke et al, 2007b) or the AMIT SMRL (Adi and Etzion, 2004), special, proprietary languages, GUI-based approaches with 3GL (e.g. Java) code respectively EPL code generation. There is ample evidence that there is no single EPL which is best suited for all domains, but that different representation approaches are needed such as rule-based EPLs for describing higher-level conditional business event patterns and pattern-matching SQL-like EPLs for defining patterns of low-level aggregation views according to event types defined as nested queries. In the following we will discuss the prominent rule-based approaches.

Reaction RuleML

Reaction RuleML (Paschke et al, 2006d) (Paschke et al, 2007b) is a general, practical, compact and user-friendly XML-serialized sublanguage of RuleML for the family of reaction rules (see chapter Rule Markup Languages and Semantic Web Rule Languages). It incorporates various kinds of production, action, reaction, and KR temporal/event/action logic rules as well as (complex) event/action messages into the native RuleML syntax using a system of step-wise extensions (see figure 4).

Figure 4: Scope of Reaction RuleML

The general syntax of a reaction rule consists of six partially optional parts:

```

<Rule style="active" evaluation="strong">
  <label> <!-- metadata --> </label>
  <scope> <!-- scope --> </scope>
  <qualification> <!-- qualifications --> </qualification>
  <oid> <!-- object identifier --> </oid>
  <on> <!-- event --> </on>
  <if> <!-- condition --> </if>
  <then> <!-- conclusion --> </then>
  <do> <!-- action --> </do>
  <after> <!-- postcondition --> </after>
  <else> <!-- else conclusion --> </else>
  <elseDo> <!-- else/alternative action --> </elseDo>
  <elseAfter> <!-- else postcondition --> </elseAfter>
</Rule>
  
```

Inbound and outbound messages < Message > can be used to interchange events (e.g. queries and answers) and rule bases (modules) between system/services or agent nodes:

```

<Message mode="outbound" directive="pragmatic performative">
  <oid> <!-- conversation ID--> </oid>
  
```

```

<protocol> <!-- transport protocol --> </protocol>
<sender> <!-- sender agent/service --> </sender>
<content> <!-- message payload --> </content>
</Message>

```

@mode = inbound/outbound - attribute defining the type of a message

@directive - attribute defining the pragmatic context of the message, e.g. a FIPA ACL performative

< oid > - the conversation id used to distinguish multiple conversations and conversation states

< protocol > - a transport protocol such as HTTP, JMS, SOAP, Jade, Enterprise Service Bus (ESB) ...

< sender >< receiver > - the sender/receiver agent/ service of the message

< content > - message payload transporting a RuleML / Reaction RuleML query, answer or rule base

The directive attribute corresponds to the pragmatic instruction, i.e. the pragmatic characterization of the message context. External vocabularies defining pragmatic performatives might be used by pointing to their conceptual descriptions. The typed logic approach of RuleML enables the integration of external type systems such as Semantic Web ontologies or XML vocabularies. A standard nomenclature of pragmatic performatives is defined by the Knowledge Query Manipulation Language (KQML) and the FIPA Agent Communication Language (ACL) which defines several speech act theory-based communicative acts.

R2ML

Like RuleML, REVERSE II Rule Markup Language (R2ML) (R2ML, 2006) (see also chapter Rule Markup Languages and Semantic Web Rule Language) is a general Web rule markup language which provides support for reaction rules. The model for complex event expressions uses the standard SNOOP-like event algebra expressions such as sequence, parallel, choice, not – as can be seen in figure 5.

Figure 5: R2ML Event Expression Meta Model

Complex selection and consumption policies as in Reaction RuleML cannot be defined.

IBM Situation Manager Rule Language (SMRL) (Adi and Etzion, 2002) is a markup language for describing situations, which are semantic concepts in the customers' domain of discourse and syntactically equivalent to (complex) event patterns. The Situation Manager is part of the Amit (Active Middleware Technology) framework (Adi and Etzion, 2004).

Figure 6: Amit Architecture

Events in SMRL have a flat structure, and have a unique name and attributes that can be standard or user defined. The conceptual model defines an event type generalization hierarchy. The root event type defines a set of standard attributes which reflect, e.g., the event source, temporal dimensions like occurrence time, detection time and /or transaction time of an event, and quantifications such as status, priority, severity, count. All application specific event types are children which inherit the standard attributes and contain further specific user attributes. A similar approach is also used in IBM's Tivoli Event Console (IBM, 2001), where the supported event model forms a nested multi-level generalization hierarchy.

The markup language provides an event algebra with joining operators (*all*, *sequence*), counting operators (*atleast*, *atmost*, *nth*), absence operators (*not*, *unless*) and temporal operators (*every*, *after*, *any*).

The algebra is combined with semantic notations such as context (*lifespan intervals*), and semantic association (*event group*). Events can happen during or outside of time intervals which have duration and may have one or more initiators, zero or more terminators, and may have related keys for grouping of event instances according to application specific criteria. In order to reason about the different time intervals SMRL applies Allen's interval algebra plus additionally all converse interval relation operators.

Situation definitions are the main constructs in SMRL. A situation instance has a unique name and a

validity interval. Situations are reported in AMIT as new event. This allows the chained execution of situation operators. Predefined external actions may fire due to detected situations according to their detection type policies (immediate, delayed, deferred). Additionally the event instance selection (first, last, all), and event instance consumption policies (consume, keep) can be configured in Amit.

The main focus of SMRL is on the definition of situations and their detection conditions and not on the descriptions of reaction rules with complex chained conditions and actions in general.

IBM's Common Base Event (CBE) (CBE, 2003) is intended as an implementation of the Web Services Distributed Management (WSDM) Web Event Format standard (WSDM, 2006). The data model of CBE assumes that events can have a hierarchical nested structure of extended user-attributes including also aggregation hierarchies for event types. That is, in contrast to SMRL and Tivoli TEC where events have a flat data structure, CBE supports nested aggregations of event types. CBE is mainly a mechanism for managing and interchanging events in business enterprise applications and distributed services.

ruleCore Markup Language (rCML) is used for specification of events and ECA rules in ruleCore (Seiriö and Berndtsoon, 2005). Like in SMRL, in ruleCore terminology a composite event is called a *situation* and the main focus ruleCore is situation detection. Two different event types are supported in rCML: basic events and composite events. Each composite event defined in rCML has four sub elements:

- **<detect-event>**. The **<detect-event>** element specifies the event that is generated when the composite event is detected.
- **<no-detect-event>**. The **<no-detect-event>** element specifies the event that is generated when the composite event can not be detected.
- **<event-selector>** The **<event-selector>** element specifies a logical condition (or filter) for the composite event.
- **<detector>**. The composite event detector consists of a number of event algebra operators (the SNOOP algebra operators) that describe the composite event.

Conditions in rCML are possible in a number of places, e.g., as logical conditions for events (event selector), or as rule conditions. Four types of rule actions are supported in rCML:

- **script** - Script actions execute external scripts or applications
- **event** - Event actions send event occurrences to the ruleCore rule engine
- **create - state** - Rule actions that create new state items
- **delete - state** - Rule actions that delete state items

ECA rules are specified inside a **<rules>** element and each individual ECA rule is described with a **<rule>** element. The **<rule>** element has the following attributes:

- **name** - A unique value for the name attribute that identifies the rule.
- **create** - The create attribute controls rule instance creation.
- **parameter** - Controls how the event parameters of each event are stored during the event detection process.

The **<rule>** element has the following structure:

- The **<description>** element contains a description of the rule.
- The **<event-ref>** element contains a reference to the composite event that trigger the rule.

- The <condition-ref> contains a reference to a condition definition element <condition-def> that specifies a condition that is evaluated when the rule is triggered by its event.
- The <action-ref> contains a reference to a <action-def> element that is executed if the rule condition is evaluated to true.
- The <minus-action-ref> contains a reference to a <action-def> element that is executed if the triggering event can never be detected.
- The <instance-limit> element is used to limit the number of the rule instance for each type of rule.

XChange (Pătrânjan, 2005) is another Web rule language developed in REVERSE for Event-Condition-Action rules (ECA rules) of the form "*Event Query — Web Query — Action. Event queries*". Web queries and actions follow the same approach of specifying patterns for data queried, updated, or constructed. Every incoming event is queried using the event query (introduced by keyword ON). If an answer is found and the Web query (introduced by keyword FROM) has also an answer, then the specified action (introduced by keyword DO) is executed. Event queries are evaluated against the stream of incoming events. The condition part of XChange rules queries data from Web resources such as XML documents or RDF documents. Such Web queries are expressed in Xcerpt, a Web and Semantic Web query language.

The rule parts communicate through variable substitutions. Substitutions obtained by evaluating the event query can be used in the Web query and the action part, those obtained by evaluating the Web query can be used in the action part. This behavior is adopted from earlier logic-based ECA rule languages such as Prova's ECA-LP (Paschke, 2006b, <http://prova.ws/>) which combines ECA rules and derivation rules and supports queries on derivation rules sets in the condition part of an ECA rule with variable binding and backtracking.

MARS (Modular Active Rules for the Semantic Web) (Behrends et al, 2008) is another modular framework developed in REVERSE for specifying and implementing active rules (ECA rules) by *composing* languages for events, queries, conditions, and actions by separating the ECA semantics from the underlying semantics of events, queries, and actions (see figure 7).

Figure 7: MARS Architecture

ECA rules in MARS have the following structure:

every rule uses an event language (E), one or more query languages (Q), a test language (T), and an action language (A) for the respective components, each of these languages and their constructs are described by metadata and an ontology of its semantics, and their nature as a language, e.g., associating them with a processor, there is a well-defined *interface* for communication between the E, Q&T, and A components by variables. The four types of rule components use specialized types of languages that, although dealing with different notions, share the same algebraic language structure:

- event languages: every expression gives a description of a (possibly composite) event. Expressions are built by composers of an event algebra, where the leaves are atomic events of the underlying application;
- query languages: expressions of an algebraic query-language;
- test languages: they are in fact formulas of some logic over literals (of that logic) and an underlying domain (that determines the predicate and function symbols, or class symbols etc., depending on the logic);
- action languages: every expression describes an (possible composite) activity. Here, algebraic languages (like process algebras) or “classical” programming languages (that nevertheless consist of expressions) can be used. The atomic items are actions of the application domain.

Rule Responder (<http://responder.ruleml.org>) is a middleware for distributed intelligent rule-based CEP and Pragmatic Web inference agents / services on the Web. It adopts the OMG model driven architecture (MDA) approach:

1. On the computational independent level rules are engineered computational independent e.g. in a natural controlled English language using blueprint templates and user-defined controlled vocabularies.
2. The rules are mapped and serialized in a interchange format such as Reaction RuleML (or RIF XML) which is used as platform independent rule interchange format to interchange rules between the rule inference services and other arbitrary execution environments.
3. The Reaction RuleML rules are translated into the platform specific rule language for execution, e.g. the Prova rule language.

Fig. 8. Prova Rule Engines deployed as Web Rule Inference Services on Rule Responder Enterprise Service Bus using Reaction RuleML as interchange format

Fig. 9 Prova Rule Engines communicate using Reaction RuleML via ESB transport (>30 Protocols)

Figure 8 illustrates the Rule Responder enterprise service middleware. Translator services translate from Prova execution syntax into Reaction RuleML and vice versa (see Fig. 9). Arbitrary transport protocols such as MS, SMTP, JDBC, TCP, HTTP, and XMPP (the ESB supports all common transport protocols) can be used to transport rule sets, queries and answers between distributed Prova endpoints on the Web using Reaction RuleML as common rule interchange format. For workflow-like intelligent message processing Rule Responder employs the Web rule engine Prova (<http://prova.ws>) and messaging reaction rules to describe (abstract) processes / situations and complex event processing logic in terms of message-driven conversations between inference services / agents, i.e. it represents their associated interactions via constructs for asynchronously sending and receiving event messages. The control flow of a process is defined by the order of sending and receiving message constructs in messaging reaction rules. In contrast to standard Event-Condition-Action (ECA) rules, which typically only have one global state, messaging reaction rules maintain a local conversation state that reflects the process execution state and supports performing different activities within process instances managed in simultaneous conversation branches.

FUTURE TRENDS

The CEP technology, which provides enhanced situation awareness, is considered as one of the main prerequisites for many emerging technologies such as operational intelligence systems, predictive business enterprise networks, real-time adaptive enterprises and autonomic IT systems.

They typically achieve higher levels of scalability, but provide only inadequate expressiveness to describe, in a declarative way, more complex behavioral logic with expressive conditions, complex actions, and workflow-like executions as needed to declaratively program domain problems on a very high level of abstraction and to provide qualitatively new solutions.

Modern declarative rule languages based especially on AI and logic-based (formal) representations enable the evolution from pure event processing systems into "intelligent" reactive systems. The work done in temporal logic, spatial reasoning, knowledge representation, ontologies, etc. enables a more declarative representation of events and actions as well as their semantic processing, including for correctness proofs.

As a result, **Intelligent CEP** based on reaction rules offers the prospect of efficiently detecting situations of interest from vast amounts of events (from the so-called event cloud), and at the same time provide the required expressiveness to describe higher-level logic such as business rule policies, reactive workflows and business processes.

However, the heterogeneous and highly distributed nature of event-processing systems, especially on the Web, provides new challenges for the declarative rule technologies, such as contextualized reasoning over large data streams, scalable mapping of complex structures, and distributed approximative reasoning, to name just a few.

At present, the areas of event processing and rules (in particular logic-based rules) are addressed by different research and development communities. These communities are just beginning to cooperate, realizing that their areas will benefit from each other. For instance, the RuleML initiative (<http://ruleml.org>), which advances Web rule standards (e.g. RuleML, SWRL, W3C RIF, OMG PRR), technologies and applications has recently co-founded the Event Processig Technology Society (<http://www.ep-ts.com/>). Several international research projects already work on initial prototypes of rule-based intelligent CEP systems, e.g. Rule Responder (see subsection on Rule Responder), and on rule-based Event Processing Languages, e.g. Reaction RuleML and an envisioned RIF Reaction Rules dialect (www.w3.org/2005/rules/).

However, practical experience in using rule and CEP technology and setting up event-driven rule-based applications has shown that potential adopters have major problems in understanding and adequately designing and implementing successful solutions. CEP engineering remains a laborious trial-and-error process with slow development and change cycles. A design pattern approach for describing and collecting successful rule patterns and CEP patterns should support engineers in their design decisions to build robust and efficient solutions with well-understood tradeoffs, and should enable an interdisciplinary and efficient

communication process about successful solutions in different application domains. (Paschke, 2008a)

CONCLUSION

Event-based systems are gaining increasing momentum and become ever more important in various application domains, ranging from traditional business applications, like supply-chain management, to the entertainment industry, like on-line gaming applications. The dominating market research companies, like Gartner or Forrester, predicted a key role of event-driven processing for making businesses more agile, postulating an era of events which will follow the current service-orientation of Web based applications. The combination of the emerging CEP technology for processing events and creating enhanced situation-awareness with the expressiveness of reaction rules and declarative rules in general, promises higher levels of expressiveness and declarative programming flexibility, while still maintaining the required efficiency and scalability for real-time applications on vast amounts of event data.

In this chapter we have surveyed the major streams of reaction rule languages and technologies, and have discussed the different views and approaches in the different domains. We have implemented a multi-dimensional classification scheme to capture the broad variety of features supported by them. The scheme provides a common vocabulary and lays the theoretical foundations for a thorough analysis and discussion of modern reaction rules technologies. It can contribute to a more efficient communication process between different communities about design solutions for reaction rule / rule-based event processing languages. And, it can guide engineers in their decisions about adequate rule languages and tools for certain implementations.

REFERENCES

- (Abiteboul and Vianu, 1991) Abiteboul, S. and V. Vianu, Datalog extensions for database queries and updates. *Journal of Computer and System Science*, 1991. 43: p. 62-124.
- (Adi and Etzion, 2002) Asaf Adi and Opher Etzion, The situation manager rule language, In proceedings RuleML 2002, 2002.
- (Adi and Etzion, 2004) Adi, A. and Etzion, O. 2004. Amit - the situation manager. *The VLDB Journal* 13, 2 (May. 2004), 177-203.
- (Agrawal et al, 1991) R. Agrawal, R. Cochrane, and B. Lindsay. On maintaining priorities in a production rule system. In Proceedings International Conference on Very Large Data Bases, pages 479--487, Barcelona, Spain, 1991.
- (Aiken et al, 1994) Aiken, A., J. Widom, and J.M. Hellerstein. Behaviour of database production rules: termination, confluence and observable determinism. in Int. Conf. on Management of Data. 1994: ACM.

- (Albert, 1989) Luc Albert. Average case complexity analysis of rete pattern-match algorithm and average size of join in databases. Rapport de recherche no. 1010, INRIAROCQUENCOURT, 1989.
- (Alferes, 2002) Alferes, J.J., et al. Evolving logic programs. in JELIA'02. 2002.
- (Allen, 1983) James F. Allen: Maintaining knowledge about temporal intervals. In: Communications of the ACM. 26/11/1983. ACM Press. S. 832-843, ISSN 0001-0782
- (Baeten and Weijland, 1990) J.C.M. Baeten and W.P. Weijland. Process Algebra. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.
- (Baeten, 2005) Baeten, J. C. 2005. A brief history of process algebra. *Theor. Comput. Sci.* 335, 2-3 (May. 2005), 131-146.
- (Baldoni et al, 2003) Baldoni, R., M. Contenti, and A. Virgillito. The Evolution of Publish/Subscribe Communication Systems. in Future Directions in Distributed Computing. 2003.
- (Baley, 1997) Baley, J., et al. Abstract interpretation of active rules and its use in termination analysis. in Int. Conf. on Database Theory. 1997.
- (Baralis and Widom, 1994) Baralis, E. and J. Widom. An algebraic approach to rule analysis by means of triggering and activation graphs. in VLDB 94. 1994.
- (Bennett and Galton, 2004) Bennett, B. and A.P. Galton, A unifying semantics for time and events. *Artificial Intelligence*, 2004. 153(1-2): p. 13-48.
- (Behrends, 1994) H. Behrends: "Simulation-based Debugging of Active Databases", Proc. IEEE Workshop "Research Issues in Database Engineering, RIDE 94", Houston, TX, 1994.
- (Behrends, 1995) Behrends, H., A description of Event Based Activities in Database Related Information Systems, Report 3/1995. 1995, Univ. of Oldenburg.
- (Behrends et al, 2008) Erik Behrends, Oliver Fritzen, Wolfgang May, Franz Schenk; Event Algebras and Process Algebras in ECA Rules. *Fundamenta Informaticae* 82, pp. 237-263, 2008.
- (Björn and Parrow, 1998) Björn Victor and Joachim Parrow, Concurrent Constraints in the Fusion Calculus, Proceedings of ICALP', 1998.
- (Bonner and Kifer, 1995) Bonner, A.J. and M. Kifer, Transaction logic programming (or a logic of declarative and procedural knowledge). 1995, University of Toronto.
- (Burdescu and Brezovan, 2001) Dumitru Dan Burdescu and Marius Brezovan. Algorithms for high level petri nets simulation and rule-based systems. *Acta Universitatis Cibiniensis*, XLIII:33 – 39, 2001.
- (Cardelli and Gordon, 1998) Cardelli, L.; A.D. Gordon. "Mobile Ambients". Proceedings of the First international Conference on Foundations of Software Science and Computation Structure (March 28 - April 4, 1998). M. Nivat, Ed. Lecture Notes In Computer Science 1378: 140–155. Springer-Verlag.
- (CBE, 2003) Common Situation Data Format: The Common Base Event– submitted to OASIS, 2003.
- (Chandy, 2006) K.M. Chandy, Event-driven applications: costs, benefits and design approaches, Gartner Application Integration and Web Services Summit 2006
- (Chakravarthy, 1994) Chakravarthy, S., et al. Composite Events for Active Databases: Semantics Contexts and Detection. in VLDB 94. 1994.
- (Cheng et al, 1993) Cheng, A. M., Browne, J. C., Mok, A. K., and Wang, R. 1993. Analysis of Real-Time Rule-Based Systems with Behavioral Constraint Assertions Specified in Estella.

IEEE Trans. Softw. Eng. 19, 9 (Sep. 1993).

(Cervesato et al, 1996) I. Cervesato, L. Chittaro, and A. Montanari. A general modal framework for the event calculus and its skeptical and credulous variants. In ECAI'96, 1996.

(CEP, 2008) Event Processing Glossary, Event Processing Technology Society, <http://complexevents.com/?p=361>, 2008.

(Cirstea et al, 2004) Cirstea H., Kirchner C., Moossen M., Moreau P.-E. Production Systems and Rete Algorithm Formalisation, Rapport de recherche n° inria-00280938 – version 1 (2004).

(Collet and Coupaye, 1996) Collet, C. and T. Coupaye. Composite Events in NAOS. in Dexa. 1996. Zürich, Switzerland.

(Cugola et al, 1998) Cugola, G., E.D. Nitto, and A. Fuggeta. Exploiting an event-based infrastructure to develop complex distributed systems. in Int. Conf. on Software Engineering. 1998.

(Dayal et al, 1996) Dayal, U., A. Buchmann, and S. Chakravarty, The HiPAC Project, in Active Database Systems, J. Widom and S. Ceri, Editors. 1996, Morgan Kaufmann.

(Declambre and Etheredge, 1988) Declambre, L.M.L. and J.N. Etheredge. A self-controlling interpreter for the relational production language. in ACM Sigmod Int. Conf. on the Management of Data. 1988.

(Denecker et al, 1992) M. Denecker, L. Missiaen, and M. Bruynooghe. Temporal reasoning with abductive event calculus. In ECAI'92, Vienna, Austria, 1992.

(Diaz and Jaime, 1997) Díaz, O. and Jaime, A. 1997. EXACT: an extensible approach to active object-oriented databases. *The VLDB Journal* 6, 4 (Nov. 1997), 282-295.

(Dinn et al, 1996) A. Dinn, N.W. Paton, M.H. Williams, and A.A.A. Fernandes. An Active Rule Language for ROCK & ROLL. In Proc. 14th British National Conference on Databases. Springer-Verlag, 1996.

(Dietrich et al, 2003) Dietrich, J., et al., Rule-Based Agents for the Semantic Web. Journal on Electronic Commerce Research Applications, 2003.

(Doherty, 1998) Doherty, P., et al., TAL: Temporal Action Logics language specification and tutorial. Linköping Electronic Articles in Computer and Information Science, 1998. 3(015).

(Dung and Mancarella, 1996) Dung, P.M., Mancarella, P., 1996. Production systems need negation as failure. In Proceedings of 13th National Conference on AI, AAAI press.

(Dung and Mancaralle, 2002) Dung, P.M. and P. Mancaralle, Production Systems with Negation as Failure. *IEEE Transactions on Knowledge and Data Engineering*, 2002. 14(2).

(Eiter, 2001) Eiter, T., et al. A framework for declarative update specification in logic programs. in IJCAI. 2001.

(Eiter et al, 2004) Eiter, T., Faber, W., Leone, N., Pfeifer, G., and Polleres, A. 2004. A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM Trans. Comput. Logic* 5, 2 (Apr. 2004), 206-263.

(Erikson, 1993) Erikson, J., CEDE: Composite Event Detector in An Active Database. 1993, University of Skövde.

(Fages and Lissajoux, 1992) Francois Fages and R'emi Lissajoux. S'émantique opérationnelle et compilation des systèmes de production. *Revue d'intelligence artificielle*, 6 - 4:431–456, 1992.

(Fikes and Nilsson, 1971) Fikes, R.E. and N.J. Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 1971(2): p. 189-208.

(Fournet and Gonthier, 2000) Cedric Fournet, Georges Gonthier. "The Join Calculus: A

- Language for Distributed Mobile Programming, In Proceedings of the Applied Semantics Summer School (APPSEM), Caminha, September 2000.
- (Flesca and Greco, 1998) Flesca, S. and Greco, S. 1998. Declarative Semantics for Active Rules. In *Proceedings of the 9th international Conference on Database and Expert Systems Applications* (August 24 - 28, 1998). G. Quirchmayr, E. Schweighofer, and T. J. Bench-Capon, Eds. Lecture Notes In Computer Science, vol. 1460. Springer-Verlag, London, 871-880.
- (Forgy, 1982) Charles Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", *Artificial Intelligence*, 19, pp 17-37, 1982
- (Gatzui and Dittrich, 1993) Gatzui, S. and K. Dittrich. Event in an active object-oriented database system. in *Int. Conf. on Rules in Database Systems*. 1993. Edinburgh.
- (Gehani et al, 1992) Gehani, N., H.V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. in *Int. Conf. on Management of Data*. 1992. San Diego.
- (Gelfond and Lifschitz, 1993) Gelfond, M. and V. Lifschitz, Representing action and change by logic programs. *Journal of Logic Programming*, 1993. 17(2-4): p. 301-321.
- (Giunchiglia and Lifschitz, 1997) E. Giunchiglia, N. Kartha, and V. Lifschitz. Representing action: indeterminacy and ramifications. *Artificial Intelligence*, 95:409--443, 1997.
- (Giunchiglia and Lifschitz, 1998) Giunchiglia, E. and V. Lifschitz. An action language based on causal explanation: Preliminary report. in *Conf. on Innovative Applications of Artificial Intelligence*. 1998. Menlo Park, CA: AAAI Press.
- (Giunchiglia and Lifschitz, 1999) Giunchiglia, E. and V. Lifschitz, Action languages, temporal action logics and the situation calculus. *Linköping Electronic Articles in Computer and Information Science*, 1999. 4(040).
- (Giunchiglia et al, 2004) Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories, *Artificial Intelligence*, 153, (2004), 49-104.
- (Grosz, 2005) Grosz, B. "The Production Logic Programs Approach, in a Nutshell: Foundations for Semantically Interoperable Web Rules". MIT Sloan School of Management, Working Paper of Dec. 19, 2005.
- (Guangtian et al, 1998) Guangtian Liu, Aloysius K. Mok, and P. Konana. A unified approach for specifying timing constraints and composite events in active real-time database systems. In *Real-Time Technology and Applications Symposium*, June 1998.
- (McCarthy and Hayes, 1969) McCarthy, J. & Hayes, P.J. , "Some Philosophical Problems from the Standpoint of Artificial Intelligence", in *Machine Intelligence 4*, ed. D.Michie and B.Meltzer, Edinburgh: Edinburgh University Press, pp. 463-502, 1969.
- (Hanson and Widom, 1993) Hanson, E.N., Widom, J., An Overview of Productions Rules in Database Systems, *Knowledge Engineering Review*, vol. 8, no. 2, pp. 21-142, 1993.
- (Hayes-Roth, 1985) Hayes-Roth, F., Rule based systems. *ACM Computing Surveys*, 1985. 28(9).
- (Hillston, 1996) J. Hillston. A Compositional Approach to Performance Modelling. Cambridge University Press, 1996.
- (Hoare, 1985) Hoare, C.A.R., Communication and Concurrency. 1985: Prentice Hall.
- (Hölldobler and Schneeberger, 1990) Hölldobler, S. and J. Schneeberger, A new deductive approach to planning. *New Generation Computing*, 1990. 8(3): p. 225-244.
- (IBM, 2001) IBM Tivoli Enterprise Console - documentation, 2001.
- (IBM, 2001b) IBM Informix Time Series Data Blade Module, User Guide, V.4.0, 2001.
- (IBM, 2003) Common Event Infrastructure, IBM, 2003.

- (Ishida, 1994) Toru Ishida. An optimization algorithm for production systems. IEEE Transactions on Knowledge and Data Engineering, 6 - 4:549–558, 1994.
- (Kakas and Miller, 1997) A. Kakas and R. Miller. A simple declarative language for describing narratives with actions. JLP (special issue), 31(1-3):157200, 1997.
- (Kakas and Miller, 1999) A. Kakas, R. Miller, and F. Toni. An argumentation framework for reasoning about actions and change. In LPNMR'99, 1999.
- (Kowalski, 1992) R. Kowalski. Database updates in the event calculus. Journal of Logic Programming, 12:121-146, 1992.
- (Kowalski and Sergot, 1986) Kowalski, R.A. and M.J. Sergot, A logic-based calculus of events. New Generation Computing, 1986. 4: p. 67-95.
- (Kulkarni et al, 1999) Kulkarni, K., Mattos, N.M., Cochrane, R. Active Database Features in SQL3. Active Rules in Database-Systems. Springer Verlag, 1999. 197-219.
- (Lausen et al, 1998) Lausen, G., B. Ludascher, and W. May, On Logical Foundations of Active Databases. Logics for Databases and Information Systems, 1998: p. 389-422.
- (Leite et al, 2001) Leite, J.A., J.J. Alferes, and L.M. Pereira. On the use of multi-dimensional dynamic logic programming to represent societal agents' viewpoints. in EPIA01. 2001.
- (Leite, 2003) Leite, J.A., Evolving Knowledge Bases. Frontiers in Artificial Intelligence and Applications, 2003. 81.
- (Luckham 2002) Luckham, D. The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems, Addison Wesley Professional, May 2002, ISBN: 0201727897
- (Meo et al, 1996) Meo, R., G. Psaila, and S. Ceri. Composite Events in Chimera. in EDBT. 1996. Avignon, France.
- (Meolic, 2000) Meolic, R., T. Kapus, and Z. Brezonic. Verification of concurrent systems using ACTL. in IASTED Int. Conf. on Applied Informatics (AI'00). 2000. Anaheim, Calgary: IASTED/ACTA Press.
- (Meolic, 2003) Meolic, R., T. Kapus, and Z. Brezonic. An Action Computation Tree Logic With Unless Operator. in Proc. of South-East European Workshop on Formal Methods (SEEFM'03). 2003. Thessaloniki, Greece.
- (Milner, 1989) Milner, R., Communication and Concurrency. 1989: Prentice-Hall.
- (Miranker, 1990) Daniel Miranker. Treat: a new and efficient match algorithm for AI production systems. Morgan Kaufmann, 1990.
- (Motwani et al, 2003) Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein und Rohit Varma: Query Processing, Resource Management, and Approximation in a Data Stream Management System. Stanford, 2002 (CIDR 2003)
- (Naqvi and Krishnamurthy, 1988) Naqvi, S. and R. Krishnamurthy. Database updates in logic programming. in ACM Symposium on Principles of Database Systems. 1988. New York: ACM.
- (Naqvi and Ibrahim 1994) Naqvi, W. and Ibrahim, M. T. 1994. EECA: An Active Knowledge Model. In *Proceedings of the 5th international Conference on Database and Expert Systems Applications* (September 07 - 09, 1994). D. Karagiannis, Ed. Lecture Notes In Computer Science, vol. 856. Springer-Verlag, London, 380-389.
- (Oracle, 1998) Oracle8 Time Series Data Cartridge, White Paper, February 1998
- (Oracle, 1999) Migrating from Oracle Time Series to Oracle Analytic Functions: Time Series Functions, White Paper, November 1999

- (Pang, 1994) Pang, P. and Raschid, L., Bottom-Up magic evaluation techniques for stratified production rules in a DBMS. Technical Report, University of Maryland Institute for Advanced Computer Studies, 1994
- (Paschke, 2006a) Adrian Paschke: ECA-RuleML: An Approach combining ECA Rules with temporal interval-based KR Event/Action Logics and Transactional Update Logics [CoRR abs/cs/0610167](#): (2006)
- (Paschke, 2006b) Adrian Paschke: ECA-LP / ECA-RuleML: A Homogeneous Event-Condition-Action Logic Programming Language [CoRR abs/cs/0609143](#): (2006)
- (Paschke, 2006c) Paschke, A.: ECA-RuleML/ECA-LP, Int. Conf. of Rule Markup Languages (RuleML'06), Athens, Georgia, USA, 2006, http://2006.ruleml.org/online-proceedings/poster_submission_54.pdf.
- (Paschke et al, 2006d) A. Paschke, A. Kozlenkov, H. Boley, M. Kifer, S. Tabet, M. Dean, and K. Barrett. Reaction ruleml, <http://ibis.in.tum.de/research/reactionruleml/>, 2006.
- (Paschke et al, 2006e) Paschke, A., Kiss, C., Al-Hunaty, S.: NPL: Negotiation Pattern Language - A Design Pattern Language for Decentralized (Agent) Coordination and Negotiation Protocols, In: Banda, R. (Ed.), In: E-Negotiation - An Introduction, ICFAI University Press, ISBN 81-314-0448-X, September 2006.
- (Paschke, 2007a) Paschke, A.: Rule-Based Service Level Agreements - Knowledge Representation for Automated e-Contract, SLA and Policy Management, ISBN 978-3-88793-221-3, IDEA Verlag GmbH, München. 2007.
- (Paschke et al, 2007b) Paschke, A. Kozlenkov, A., Boley, H.: A Homogenous Reaction Rules Language for Complex Event Processing, International Workshop on Event Drive Architecture for Complex Event Process (**EDA-PS 2007**) at VLDB'07, Vienna, Austria, 2007.
- (Paschke, 2008) Paschke, A.: DoReMoPat - Domain Specific Reference Models for Complex Event Processing, EU Consultation workshop on priorities for the FP7 WP2009-10, ftp://ftp.cordis.europa.eu/pub/fp7/ict/docs/ssai/events-20080304-05-paschke_en.pdf, Brussels, March, 2008.
- (Paschke, 2008a) Paschke, A.: Design Patterns for Complex Event Processing, 2nd International Conference on Distributed Event-Based Systems (DEBS'08), Rome, Italy, 2008.
- (Paton, 1995) Paton, N., et al. Formal Specification Of Active Database Functionality: A Survey. in 2nd Int. Workshop RIDS-95. 1995. Athens, Greece.
- (Paton, 1999) Paton, N.W., Active Rules in Database Systems. Monographs in Computer Science. 1999: Springer.
- (Pătrânjan, 2005) Paula-Lavinia Pătrânjan: The Language XChange: A Declarative Approach to Reactivity on the Web. PhD Thesis (Dissertation). University of Munich, Germany. July 2005.
- (Pednault, 1994) Edwin P. D. Pednault: ADL and the State-Transition Model of Action. J. Log. Comput. 4(5): 467-512 (1994)
- (Ramakrishnan, 1998) Ramakrishnan, R. et al.: "SQRL: Sorted Relational Query Language", SSDBM 1998, pp. 84-95
- (Raschid, 1992) Raschid, L., A semantics for a class of stratified production system programs. 1992, Univ. of Maryland Institute for Advanced Computer Studies-UMIACS-TR-91-114.1: College Park, MD, USA.
- (Raschid, 1996) Raschid, L. and J. Lobo, Semantics for Update Rule Programs and Implementation in a Relational Database Management System. ACM Transactions on Database Systems, 1996. 22(4): p. 526-571.

- (Reddi et al, 1999) Swarup Reddi, Alexandra Poulovassilis, Carol Small: PFL: An Active Functional DBPL. *Active Rules in Database Systems 1999*: 297-308
- (Reiter, 1995) Reiter, R., On specifying database updates. *Journal of Logic Programming*, 1995. 25(1): p. 53-91.
- (R2ML, 2006) REVERSE I1 Rule Markup Language (R2ML), <http://oxygen.informatik.tu-cottbus.de/reverse-i1/?q=R2ML>, 2006.
- (Sadri and Kowalski, 1995) F. Sadri and R. Kowalski. Variants of the event calculus. In *Int. Conf. on Logic Programming*, Kanagawa, Japan, 1995.
- (Sadri, 2001a) Sadri, R., Zaniolo, C., Zarkesh, A., Adibi, J.: “Optimization of Sequence Queries in Database Systems”, 2001
- (Sadri, 2001b) Sadri, R.: “Optimization of Sequence Queries in Database Systems”, PhD Thesis, UCLA, 2001
- (Sandewall, 1989) Sandewall, E. Combining Logic and Differential Equations for Describing Real World Systems. in *KR89*. 1989: Morgan Kaufman.
- (Sangiorgi and Walker, 2001) Davide Sangiorgi and David Walker: *The Pi-calculus: A Theory of Mobile Processes*, Cambridge University Press, 2001, ISBN 0-521-78177-9
- (Saraswat and Gupta, 1996) Saraswat, V.A., J. R., and V. Gupta, Timed default concurrency constraint programming. *Journal of Symbolic Computation*, 1996. 22(5/6).
- (Seiriö and Berndtsson, 2005) Marco Seiriö, Mikael Berndtsson: Design and Implementation of an ECA Rule Markup Language. In *proceedings RuleML 2005*: 98-112.
- (Shadri et al, 1994) Shadri, P., Livny, M., Ramakrishnan, R.: “Sequence Query Processing” , *SIGMOD*, 1994, pp. 430-441
- (Shadri et al, 1995) Shadri, P., Livny, M., Ramakrishnan, R.: “SEQ: A Model for Sequence Databases”, *ICDE*, 1995, pp. 232-239
- (Shanahan, 1990) M. Shanahan. Representing continous change in the event calculus. In *ECAI'90*, 1990.
- (Shanahan, 1997a) M. Shanahan. *Solving the Frame Problem*. MIT Press, London, 1997.
- (Shanahan, 1997b) M. Shanahan. Event calculus planning revisited. In *4th European Conference on Planning (ECP97)*, volume 1348. Springer LNAI, 1997.
- (Sheers, 1996) Sheers, KR. "HP OpenView event correlation services." *Hewlett Packard Journal*. 47.5 (1996): 31-42.
- (Sellis et al, 1993) Sellis, T., C.C. Lin, and L. Raschid. Coupling production systems and database systems. in *ACM Sigmond Int. Conf. on the Management of Data*. 1993.
- (Son and Baral, 2001) Son, T. C. and Baral, C. 2001. Formalizing sensing actions—a transition function based approach. *Artif. Intell.* 125, 1-2 (Jan. 2001), 19-91.
- (Tambe and Rosenbloom, 1989) Milind Tambe and Paul Rosenbloom. Eliminating expensive chunks by restricting expressiveness. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 431–456, 1989.
- (Thielscher, 1999) Thielscher, M., From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. *Artificial Intelligence*, 1999. 111: p. 277-299.
- (Th´eret, 1994) Philippe Th´eret. *De l'ecacit´e des interpr´eteurs de syst ems de r egles de production dans les syst emes a base de connaissances*. PhD thesis, Universit´e Paris XIII, February 1994.
- (Veritas, 1999) “VERITAS NerveCenter™” VERITAS Software, <http://eval.veritas.com/Webfiles/docs/NCOverview.pdf>, 1999.

- (W3C-RIF-PRD, 2008) Proctor, M.: W3C RIF PRD Feature Comparison of Production Rules, http://www.w3.org/2005/rules/wiki/PR_Feature_comparison, accessed, October, 2008.
- (Widom, 1992) Widom, J., A denotational semantics for starbust production rule language. SIGMOD record, 1992. 21(3): p. 4-9.
- (Widom and Ceri, 1996) Widom, J. and S. Ceri, Active Database Systems: Triggers and Rules For Advanced Database Processing. 1996: Morgan Kaufmann.
- (Widom and Finkelstein, 1990) Widom, J. and S.J. Finkelstein. Set-oriented production rules in relational database systems. in ACM Sigmond Int. Conf. on the Management of Data. 1990.
- (WSDM, 2006) OASIS Web Services Distributed Management WSDM V1.1 as an OASIS Standard on August 01, 2006 www.oasis-open.org/committees/wsdm/, 2006.
- (Yemini et al, 1996) Yemini, S.A., Kliger, S., Mozes E., Yemini, Y., and Ohsie, D. "High speed and robust event correlation." IEEE Communications Magazine. 34.5 (1996): 82-90.
- (Zimmer and Unland, 1999) D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In ICDE'99, pages 392--399, 1999.
- (Zimmermann and Alejandro, 1999) Jürgen Zimmermann, Alejandro P. Buchmann: REACH. Active Rules in Database Systems 1999: 263-277
- (Zaniolo, 1993) Zaniolo, C. A unified semantics for Active Databases. in Int. Workshop on Rules in Database Systems. 1993. Edinburgh, U.K.

KEY TERMS AND THEIR DEFINITIONS

See section “Foundational Definitions”.