Evolution and Reactivity for the Web

José Júlio Alferes[1] and Wolfgang May[2]

[1] Centro de Inteligência Artificial - CENTRIA, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal; jja@di.fct.unl.pt
[2] Institut für Informatik, Universität Göttingen, 37083 Göttingen, Germany; may@informatik.uni-goettingen.de

Abstract. The Web and the Semantic Web, as we see it, can be understood as a "living organism" combining autonomously evolving data sources, each of them possibly reacting to events it perceives. Rather than a Web of data sources, we envisage a Web of Information Systems, where each such system, besides being capable of gathering information (querying persistent data, as well as "listening" to volatile data such as occurring events), is capable of updating persistent data, communicating the changes, requesting changes of persistent data in other systems, and being able to react to requests from other systems. The dynamic character of such a Web requires declarative languages and mechanisms for specifying the evolution of the data.

In this course we will talk about foundations of evolution and reactive languages in general, and will then concentrate on some specific issues posed by evolution and reactivity in the Web and in the Semantic Web.

1 Introduction

Use of the *Web* today –commonly known as the "World Wide Web"– mostly focuses on the page-oriented perspective: most of the Web consists of browsable HTML pages only. From this point of view, the Web can be seen as a graph that consists of the resources as nodes, and the *hyperlinks* form the edges. Here, queries are stated against individual nodes, or against several nodes. As such, the Web is mainly seen from its static perspective of autonomous *sources*, whereas the *behavior* of the sources, including active interaction of resources does not play any important role here.

But there is more on the Web of today than HTML pages. Leaving the superficial point of view of HTML pages, the Web can be seen as a set of *data sources*, some of which are still browsing-oriented, but there are also database-like resources that can actually be queried. Moreover, there are specialized information sources like Web Services and Portals.

With these representations, the perspective may shift more to the idea of a Web consisting of (a graph of) *information systems*. In these information systems, data extraction may be thought not only in terms of local queries, but also in terms of global queries that are stated against the Web, or against a group (or community) of nodes on the Web. Given the highly heterogeneous and autonomous characteristics of the Web, this requires appropriate query languages, and a way to deal with the integration of data from the various sources.

But such an infrastructure of autonomous sources should allow for more than querying. Consider a set of sources of travel agencies and airline companies. It is important to be capable of querying such a set for, e.g. timetables of flights, availability of flight tickets, etc. But a Web consisting of information systems should allow for more. For example: it should allow for drawing conclusions based on knowledge (e.g. in the form of derivation rules) available on each node; it should allow for making reservations via a travel agency, and automatically make the corresponding airline company (and also other travel agencies) aware of that; it should allow airline companies to change their selling policies, and have travel agencies automatically aware of those changes; etc. The Web, as we see it, with such capabilities can be seen as forming an active, "living" infrastructure of autonomous systems, where reactivity, evolution and its propagation plays a central role.

In this course, we discuss issues related to this question. The focus and the message of the course is on the *concepts* – specific formalisms are given as examples and illustrations. Section 2 classifies different kinds of evolution and reactivity in the (Semantic) Web and motivates the use of rules, especially ECA rules as the target formalism. Section 3 discusses foundations for modeling and reasoning about temporal issues, i.e., formalizing temporal structures, specification of actions and processes, and events. These concepts are then combined in Section 4 where existing rule-based approaches to evolution and reactivity are discussed. The application of these concepts to the Semantic Web is then discussed in Section 5. A proposal how such a framework could look like is sketched in Section 6.

2 Concepts in Evolution and Reactivity in the Web

2.1 Local and Global Reactivity and Evolution

In contrast to the conventional (Hypertext) Web, the Semantic Web consists of *active* nodes that are able to answer queries, to evolve and to communicate. Evolution of the Web is a twofold aspect: on today's Web, evolution means mainly evolution of individual Web sites that are updated locally. In contrast, considering the Web as a "living organism" that *consists* of autonomous data sources, but that will *show* a global "behavior" leads to a notion of evolution of the Web as *cooperative evolution* of (the state of) its individual resources.[1]

Below, we incrementally introduce such behavior and sketch the concepts that arise: simple reactive behavior (non-state-changing), local evolution of nodes (i.e. state-changing), and collaboration of nodes, both non-state-changing and state-changing.

Pure Reactivity Without Evolution. The most basic and primitive activity on the Web is query answering. From the point of view of the user, querying is

[1] For further details on the global behavior, and its relation to querying and evolution aspects see [42].

a static issue: there is no actual dynamic aspect in it (except possibly a delay). Nevertheless, from the point of view of the resources, there comes reactivity into play: when answering queries, the resources must answer in reaction to a query message, and in case that the query is answered with cooperation of several resources, the resources do also exchange messages.

Local Updates and Evolution. The state of such a Web node can be classified into three conceptual levels: facts (e.g. in XML, RDF or OWL; divided into data and metadata); a knowledge base given by derivation rules; and behavior (e.g., reaction patterns), e.g. of a Web Service.

When updating the Conventional Web, the update is expressed as a specific update operation on a specific Web site, using e.g. an XML update language. For the Semantic Web data formats, RDF and OWL, also update languages are available. Here, simple data updates and ontology evolution has to be distinguished.

Global Evolution. The power of the Semantic Web raises from the combination of the knowledge and behavior of sets of data sources. Especially, as an *intelligent Web*, a *cooperative evolution* of (the state of) its individual nodes is required.

Thus, having data flow and dependencies between Web sites, besides a plain communication mechanism, mechanisms to maintain consistency between Web sites by update propagation are required. Update propagation consists of (i) propagating an update, and (ii) processing/materializing the update at another Web resource. The latter, as we have just seen, is solved by local update languages. So, the remaining problem turns out to be how to communicate changes on the (Semantic) Web. Often a change is not propagated as an explicit update, but there must be "evolution" of "the Web" as a consequence of a change to some information.

For this, it is clear that we need a "global" language for communicating changes, and communication strategies for how to propagate pieces of information through the Semantic Web, seeing it globally as the union of *all* information throughout the Web. In it, the *Semantic*-property of the Web is crucial for automatically mediating between several actual schemata.

In this setting, evolution on the Web takes place by: either local changes in data sources via updates; or local evolution of Web Services by reaction to events on the Web due to their own, specified, behavior.

2.2 Event-Condition-Action (Reactive) Rules for Evolution

Following a well-known and successful paradigm, we propose to use rules, more specifically, *reactive rules* according to the *Event-Condition-Action* (ECA) paradigm for the specification of reactivity. An important advantage of them is that the *content* of the communication can be separated from the *generic semantics* of the rules themselves. Cooperative and reactive behavior is then based on

events (e.g., an update at a data source where possibly others depend on). The depending resources detect events (either they are delivered explicitly to them, or they poll them via the communication means of the Web; see next section) Then, conditions are checked (either simple data conditions, or e.g. tests if the event is relevant, trustable etc.), which are queries to one or several nodes and are to be expressed in the proposed query framework. Finally, an appropriate action is taken (e.g., updating own information accordingly). The action part can also be formulated as a transaction whose ACID properties ensure that either all actions in a transaction are performed, or nothing of is done; this also allows to check postconditions. In the literature, ECA rules are also referred to as triggers, active rules, or reactive rules.

The language for the ECA rules must comprise a language for describing events (in the "Event" part; there are atomic events, such as simple data updates or incoming messages, and *composite events* such as "if first A happens and then B"), the language for queries (in the "Condition" part), and a language for actions (including updates) and transactions (in the "Action" part). An important requirement here is that event specification and detection be as much declarative and application-level as possible. This point calls for modular design of the sublanguages and the ECA language, and the respective processors.

Usually, ECA rules are *patterns* that contain variables to be bound in the event part that are communicated to the condition and action parts.
A well-known example for simple ECA rules are e.g. the SQL triggers:

```
ON database-update WHEN condition BEGIN pl/sql-fragment END
```

where the values of the updated tuple are accessible as `old` and `new`.

These triggers react on local events in the database. For cooperative evolution and reactivity on the Web, events and other information must be *communicated.*

2.3 Communication Structure and Propagation of Knowledge

Communication on the Web as a living organism consisting of autonomous sources takes place as *peer-to-peer communication.* In this setting, evolution takes place if a resource (or its knowledge) evolves locally, and another resource that depends upon it also evolves (as a reaction). The communication can be classified by *communication strategies.*

- Push: an information source informs a client of the updates. A directed, targeted propagation of changes by the *push* strategy is only possible along registered communication paths. It takes place by explicit *messages*, that can be update messages, or just information about what happened. In this case, control flow and data flow are in parallel and synchronous.
- Pull: resources that obtain information from a source can *pull* updates by either explicitly asking whether it executed some updates recently, or can regularly update themselves based on queries against the source. Communication is based on queries and answers (that are in fact again sent as messages).

In this case, control flow and data flow are antiparallel, which has obvious drawbacks.

The above basic forms describe direct communication. Advanced communication strategies are then based on these, providing more efficient data and control flow:

- *broadcast:* general *push* to all peers.
- *blackboard:* separates the data source from answering of *pull*-queries and allows for a pre-filtering by a short *push* (to the blackboard) from where clients *pull.*
- *publish-and-subscribe* services (see e.g. [60]) receive messages from publishers and notify subscribers if the messages match the subscriptions. Here, the communication follows a pure *push* pattern, i.e., information (published items) are pushed from their originators to the pub/sub service, and derived information (notification about changes) is pushed from the pub/sub service to its subscribers.
- *continuous query systems* (see e.g., NiagaraCQ [14]) allow users to "register" queries at the service that then continuously evaluates the query (together with other queries) against the source, and informs the user about the answer (or when the answer changes). Here, communication combines *pull* and *push*: the CQ system *pulls* information from sources, and *pushes* derived information to the end user.

Note that both in the case of push and pull strategies, the actual reactivity, i.e., how the instance that is informed reacts on an *event*, can be expressed by ECA rules as described in the previous section:

- push: on an event (update), send a message (control flow + data flow).
- pull: regularly send a query (control flow) and, on a query, send an answer (data flow).
- The behavior of pub/sub and continuous query systems can also be expressed by simple ECA rules.

3 Foundations of Evolution and Reactivity

In the previous section we described the concepts that enable evolution and reactivity on the Semantic Web. In this section, we describe the theoretical background and formal means for analyzing and describing these concepts[2].

For dealing with evolution, be it in the Web or in any other context, a formal understanding of how the knowledge evolves and how to represent such evolution is needed. We start this section by describing foundational work on *models* for (temporal) knowledge evolution, that considers sequences of *states.* We then proceed by presenting temporal logics, that allow to *reason* about evolution in these models. When considering evolving sequences of states, the *actions* that cause transitions of states are also relevant. For this, in Sections 3.3 and 3.4,

[2] A more complete survey on these foundational issues can be found at [1].

we show logics for dealing with transition systems and formalisms for defining (complex) actions, respectively. Finally, we focus our attention on events, which in general are manifestations (i.e. visible consequences) of action execution, that may trigger evolution.

All these concepts are combined in Section 4, where existing rule-based approaches for evolution and reactivity are exposed.

3.1 Models of Dynamics and Temporal Structures

Kripke structures serve as a *generic* model-theoretic framework for multi-state structures: the semantics of the individual states is given by some single-state interpretations, and the Kripke structure provides the "infrastructure" that connects the states. Some (arbitrary) logic is used for the single-state interpretations, and this logic is extended, in a modular way, with additional concepts for handling the multi-state aspects. This can be done by modalities (in our situation, temporal modalities, but modalities of knowledge and belief are also often used).

Many approaches to multi-state reasoning use Kripke structures explicitly; here, temporal logics will be described. Other, often specialized, formalisms extend single-state formalisms with a notion of state (in which Kripke structures are –more or less explicitly– the model of choice).

Yet other formalisms –although basically mappable to Kripke semantics– put emphasis on the dynamic aspects, whereas the individual states and their properties become less important (Transaction Logic, and, even much stronger, process calculi).

Kripke Structures. Assume some logic (e.g., first-order logic) to describe individual states. A (first-order) *Kripke structure* is a triple $\mathcal{K} = (\mathcal{G}, \mathcal{R}, \mathcal{M})$ where $\mathcal{G}$ is a set of states (to be interpreted as states or possible worlds), $\mathcal{R} \subseteq \mathcal{G} \times \mathcal{G}$ is an *accessibility relation*, and M is a function which maps every state $g \in \mathcal{G}$ to a (first-order) structure $\mathcal{M}(g) = (M(g), \mathcal{U}(g))$ over Σ with universe $\mathcal{U}(g)$. $\mathcal{G}$ and $\mathcal{R}$ are called the *frame* of $\mathcal{K}$. A *path* p in a Kripke structure $\mathcal{K} = (\mathcal{G}, \mathcal{R}, \mathcal{M})$ is a sequence $p = (g_0, g_1, g_2, \ldots)$, $g_i \in \mathcal{G}$ with $\mathcal{R}(g_i, g_{i+1})$ holding for all i.

As mentioned above, Kripke structures provide just a multi-state "infrastructure": a suitable single-state-logic must then be chosen for an application, which is then extended to Kripke structures. *Temporal extensions*, where the Kripke structure is interpreted as a temporal structure, are suitable for our project. Even in the area of temporal applications, there are different interpretations of Kripke structures.

Labeled Transition Systems/Path Structures. Labeled transition systems are one of the fundamental concepts for modeling processes (cf. [51], [61]). We present LTSs here (semantically equivalent to the original literature) as an extension of the above Kripke Structures. A *labeled transition system (LTS)* consists of a set $\mathcal{G}$ of states/configurations, a set $\mathcal{A}$ of actions/labels (elementary actions,

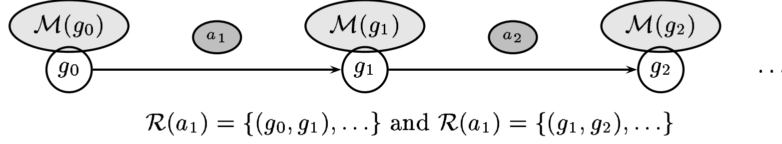

Fig. 1. Excerpt of a Kripke Structure as an LTS

or programs), and, for every $a \in \mathcal{A}$, $\mathcal{R}(a) \subseteq \mathcal{G} \times \mathcal{G}$ is a binary relation that provides the interpretation of actions (i.e., the *labeled* accessibility relation).

Another view of the same thing are *path structures*. The idea goes back to propositional *Dynamic Logic* [32], the term *path model* came up with *Process Logic* [34], where especially extended, derived accessibility relations for composite actions/programs/processes are defined (see Section 3.3).

3.2 Temporal Logics

Temporal –and other– model logics provide modal operators for modalizing the semantics of formulas of an underlying single-state logic. Due to the historical development of modal logics, the modal operators $\Box$ and $\Diamond$ were introduced. $\Box F$ stands for "F is necessary true", resp. "F holds in all possible worlds", and $\Diamond F$ for "F is possibly true", resp. "there is some possible world where F holds". Translated to modal logic of time (temporal logic), the operators are interpreted as: $\Box F$ – "always" (F holds in all subsequent states), and $\Diamond F$ – "sometimes" (F eventually holds). For reasoning in temporal Kripke structures, there are two alternatives: *linear time* considers a single path, whereas *branching time* considers a whole tree-like structure.

Linear Time Temporal Logics. The most intuitive idea for interpreting temporal logic is a sequence of states. Here, the basic operators of temporal modal logic are others, having a pure temporal semantics: $\circ$ ("nexttime") and until:

- $\circ F$: in the next state, F holds.
- F until G: there is a subsequent state where G holds, and in all states between now and this state, F holds.

The semantics of the temporal modal operators $\Diamond$ and $\Box$, is equivalently defined via until (note that there is a also an inductive definition based on $\circ$ which is typically used for model checking-like approaches):

- $\Diamond P :=$ true until P and $\Box P := \neg\Diamond\neg P$.

The Logics PLTL and FOTL. Linear Temporal Logic LTL (as propositional PLTL or as first-order FOTL) extends propositional logic with the above temporal operators: each state is a propositional or first-order interpretation, and

the states are connected as a *linear* Kripke structure (or, a single path in a branching Kripke structure is considered).
The language of LTL formulas is defined as follows:

- Every (propositional or first-order) formula is an LTL formula.
- With F and G LTL formulas, $\circ F$, $\Box F$, $\Diamond F$ and (F until G) are LTL formulas.

The satisfaction relation $\models_{\text{LTL}}$ (for short also denoted by $\models$) is defined according to the inductive definition of the syntax with respect to a propositional or first-order (infinite) linear Kripke structure $\mathcal{K} = (\mathcal{G} = \{g_1, g_2, \ldots\}, \{(n, n+1) \mid n \in \mathbb{N}\}, \mathcal{M})$, based on the propositional satisfaction relation $\models_{PL}$ or $\models_{FOL}$:
Let $g = g_i$ a state in $\mathcal{K}$, A an atomic formula, F and G LTL formulas and, in the first-order case, χ a variable assignment. Then,

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

Branching Time Temporal Logics. Applying the classical temporal modalities $\Diamond$ and $\Box$ (without $\circ$ and until) in a *branching* structure leads to surprising interpretations: Whereas in linear time logic, $g \models \Diamond F$ means that F will eventually hold in the possible future ("sometimes"), the same formula for branching time means that there *is a future*, where F will eventually hold ("not never"). For this aspect and the (dis)advantages of linear vs. branching time logic, see [37] (L. Lamport: "'Sometimes' is Sometimes 'Not Never'") and [20] (E. A. Emerson and C.-L. Lei: "Modalities for Model Checking: Branching Time Strikes Back") and several other papers.

The Logic CTL. For combining the expressiveness of both linear and branching time logic, the logics UB (*unified branching time*) [5] and CTL (*Computation Tree Logic*) [16] have been introduced:

- temporal modal operators $\circ$, $\Diamond$, $\Box$ and until (although $\Diamond$ and $\Box$ can be expressed by until, they are used here as "basic" operators to obtain the syntax definition described below),
- an existential path quantifier E ("there exists a path such that ...") and a universal path quantifier A ("on all paths").

CTL distinguishes between two different types of formulas: state formulas that hold *in* a state (all first-order formulas are state-formulas), and path formulas, that hold on paths, i.e., on *sequences* of states.

The language of CTL-formulas does not allow arbitrary combinations, but is defined as follows:

- Every first-order formula is a CTL-state formula.

- With F and G CTL-state formulas, $\neg F$, $F \wedge G$ and $F \vee G$ are CTL-state formulas.
- With F a CTL-state formula and x a variable, $\forall x : F$ and $\exists x : F$ are CTL-state formulas.
- With F and G CTL-state formulas, $\circ F$, $\Box F$, $\Diamond F$ and $(F \ \mathsf{until}\ G)$ are CTL-path formulas.
- With P a CTL-path formula, $\neg P$ is a CTL-path formula.
- With P a CTL-path formula, $\mathsf{A}P$ and $\mathsf{E}P$ are CTL-state formulas.
- Every CTL-state formula is a CTL-formula.

With the above definition, in CTL every (possibly negated) modal operator is immediately preceded by a path quantifier.

The satisfaction relation $\models_{\mathrm{CTL}}$ (for short also denoted by $\models$) is defined according to the inductive definition of the syntax with respect to a first-order Kripke structure $\mathcal{K} = (\mathcal{G}, \mathcal{R}, \mathcal{M})$, based on the first-order satisfaction relation:

Let $g \in \mathcal{G}$ be a state, $p = (g_0, g_1, \ldots)$ a path in $\mathcal{K}$, A an atomic first-order formula, F and G CTL-state formulas, P a CTL-path formula and χ a variable assignment. Then,

$$
\begin{array}{lll}
(g,\chi) \models A & :\Leftrightarrow & (M(g),\chi) \models_{PL/FOL} A\ , \\
(g,\chi) \models \neg F & :\Leftrightarrow & \text{not } (g,\chi) \models F\ , \\
(g,\chi) \models F \wedge G & :\Leftrightarrow & (g,\chi) \models F \text{ and } (g,\chi) \models G\ , \\
(p,\chi) \models \circ F & :\Leftrightarrow & (g_1,\chi) \models F\ , \\
(p,\chi) \models F \ \mathsf{until}\ G & :\Leftrightarrow & \text{there is an } i \geq 0 \text{ s.t. } (g_i,\chi) \models G \text{ and} \\
 & & \text{for all } j : 0 \leq j < i,\ (g_i,\chi) \models F\ , \\
(p,\chi) \models \neg P & :\Leftrightarrow & \text{not } (p,\chi) \models P\ , \\
(g,\chi) \models \mathsf{E}P & :\Leftrightarrow & \text{there is a path } p = (g = g_0, g_1, \ldots) \text{ in } \mathcal{K} \text{ s.t. } (p,\chi) \models P\ .
\end{array}
$$

The semantics of the modal operators $\Diamond$, $\Box$, and of the path quantifier A is defined via until and E:

$$\Diamond P := \mathsf{true\ until}\ P \quad , \quad \Box P := \neg\Diamond\neg P \quad , \quad \mathsf{A}P := \neg\mathsf{E}\neg P \quad .$$

Extensions. There exist multiple extensions of CTL in different directions. The CTL family itself provides even more expressiveness,

- CTL^{+} and $\mathrm{CTL}^{\star}$ extend to arbitrary combinations of temporal operators to path formulas; especially, "fairness" requirements cannot be expressed in CTL, but need more complex path formulas:
  - "Justice" requires that an action that is executable continuously ("waiting") from a certain state on, is eventually executed:
    $\mathrm{CTL}^{\star}$: $\mathsf{A}((\Diamond\Box(\text{Action waiting})) \rightarrow \Diamond(\text{Action is executed}))$
    (carefully note the implication semantics $A \rightarrow B \Leftrightarrow \neg A \vee B$ of this formula)
  - "Compassion (strong Fairness)" is the (stronger) requirement that every action that is executable/asked for infinitely often, is also eventually executed:
    $\mathrm{CTL}^{\star}$: $\mathsf{A}((\Box\Diamond(\text{Action waiting})) \rightarrow \Diamond(\text{Action is executed}))$

- The semantics of the accessibility relation wrt. execution of actions is considered e.g., in *Dynamic Logic* and *Hennessy-Milner-Logic* (see subsequent sections).
- Past Tense Logics add past-time temporal operators: $\bullet$ (previous state), $\blacklozenge$ (sometimes in the past), $\blacksquare$ (always in the past), and since (e.g., A since B), symmetrical to the future tense operators.

  In [22], it is shown that in the propositional case, past-tense connectives do not increase the expressiveness of temporal logic. The use of modal temporal logic for executable process specifications is described in [23], quite similar to Transaction Logic (see Section 4.3).

3.3 Logics for Labeled Transition Systems and Path Structures

The above approaches formalize a sequence of states without any special semantics for the transition. When considering evolving sequences of states, the transition is also relevant. In approaches taking this into account, in general there is a set $\mathcal{A}$ of (atomic) actions with which the transitions are labeled. The labeled transition relation leads straightforwardly to *polymodal* logics (i.e., each modality is also labeled with actions).

The following logics use not only *atomic* actions in their formulas, but define also restricted languages for *composite actions* or *programs* based on these actions.

Dynamic Logic. Dynamic Logic [32, 33, 52] provides a logic for labeled transition systems. The main difference between CTL and Dynamic Logic lies in the scope of the modalities: There, the modal operators $\Diamond$ and $\Box$ are interpreted in their historical sense as "possibly" and "necessarily", i.e., they do not apply to paths here, but only to single transitions. The modal operators are labeled with *programs* given by the algebra $\langle \mathcal{A}, \{;, \cup, ^\star\} \rangle$, (";" denotes sequential composition, $\cup$ denotes alternative composition ("choice"), and * denotes iteration). With every program a, a binary transition relation $\mathcal{R}(a) \subset \mathcal{G} \times \mathcal{G}$ is assigned.

- any (propositional or) first order formula is a DL formula, and
- for any DL formula F and any action or program a, $\langle a \rangle_{DL} F$ is a DL formula with the semantics

$$g \models \langle a \rangle_{DL} F \;\Leftrightarrow\; \text{there is a state } h \text{ such that } (g,h) \in \mathcal{R}(a) \text{ and } h \models F \;.$$

In agreement with the tradition, $\Box_{DL} F := \neg\Diamond_{DL}\neg F$ is defined to be the dual of $\Diamond_{DL}$.

Here, the difference between the interpretation of the modal operators between CTL and Dynamic Logic becomes visible: In CTL, the modal operators reach into the future along a *single* path and the path quantifiers range orthogonally over all possible futures, speaking about paths not about states. In Dynamic Logic, the modal operators look ahead one step on every path (i.e., they correspond to CTL's path quantifiers, not to CTL's modal operators).

Thus, the *eventually*, *always*, and *until*-operators can not be expressed in DL without resorting to a fixpoint logic. Instead, "if now, a is executed, F will definitely/probably hold" can be expressed.

Hennessy-Milner Logic. In [58] and [48], Hennessy-Milner-Logic, $\mathcal{HML}$, a modal logic interpretation of the CCS calculus (see Section 3.4) is given whose modalities are very similar to those of Dynamic Logic. The set of formulas of Hennessy-Milner-Logic, $\mathsf{Fml}_{\mathcal{HML}}$, is defined inductively as

- $\mathsf{T} \in \mathsf{Fml}_{\mathcal{HML}}$,
- $F \in \mathsf{Fml}_{\mathcal{HML}} \Rightarrow \neg F \in \mathsf{Fml}_{\mathcal{HML}}$,
- $F, G \in \mathsf{Fml}_{\mathcal{HML}} \Rightarrow F \wedge G \in \mathsf{Fml}_{\mathcal{HML}}$,
- $F \in \mathsf{Fml}_{\mathcal{HML}}$ and $a \in \mathcal{A} \Rightarrow \langle a \rangle_{\mathcal{HML}} F \in \mathsf{Fml}_{\mathcal{HML}}$.

(instead of $\langle a \rangle_{\mathcal{HML}}$, also $\langle a \rangle_{\mathcal{HML}}$ can be written).
CCS (see Section 3.4) does not use a notion of propositional or first-order states, but is based on the notion of ***processes*** as nodes of its LTS structures. The satisfaction relation $\models_{\mathcal{HML}}$ (for short also denoted by $\models$) between processes and $\mathcal{HML}$-formulas is defined similar to Dynamic Logic by

1. $P \models \mathsf{T}$ for all processes P ,
2. $P \models \neg F$ $:\Leftrightarrow$ not $P \models F$,
3. $P \models F \wedge G$ $:\Leftrightarrow$ $P \models F$ and $P \models G$,
4. $P \models \langle a \rangle_{\mathcal{HML}} F$ $:\Leftrightarrow$ there is a process P' s.t. $P \xrightarrow{a} P'$ and $P' \models F$.

Additionally, derived expressions in $\mathcal{HML}$ are defined:

- $\mathsf{F} \equiv \neg \mathsf{T}$,
- $F \vee G \equiv \neg(\neg F \wedge \neg G)$,
- $\langle a \rangle_{\mathcal{HML}} F \equiv \langle a_1 \rangle_{\mathcal{HML}} \ldots \langle a_n \rangle_{\mathcal{HML}} F$ for $a = a_1. \cdots .a_n$,
- $[a]_{\mathcal{HML}} F \equiv \neg \langle a \rangle_{\mathcal{HML}} \neg F$.

Similar to the discussion about CTL and Dynamic Logic above, it is not possible in $\mathcal{HML}$ to express properties like "P will eventually execute a" or "in the next step, P will execute a". Instead, "if now, a is executed, F will definitely/probably hold" can be expressed.

Process Logic. In contrast to Dynamic Logic and Hennessy-Milner Logic, where all formulas apply to states, the syntax of logics for path structures focusses on the notion of path formulas: their $\models$-relation relates paths to formulas. Nevertheless, in those logics, paths consisting of exactly one state actually take the role of states.

Process Logic [34] is a (propositional) logic for describing activities, based on path structures. It uses **path structures** with a slightly different focus of the semantics of formulas: $\mathcal{P}$ is a relation assigning sets of paths to ***programs***, i.e., it extends $\mathcal{R}$ to compound programs (thereby defining a restricted language of compound programs or transactions):

– if α and β are programs, then so are $\alpha\beta$, $\alpha \cup \beta$, and α^*.

Note that Process Logic does not have a notion of "parallel" actions. The accessibility relation is in the same way extended to these programs:

$\mathcal{P}_\alpha \quad = \mathcal{R}_\alpha$ for actions/primitive programs α,
$\mathcal{P}_{\alpha\beta} = \mathcal{P}_\alpha \mathcal{P}_\beta = \{pq \mid p \in \mathcal{P}_\alpha \text{ and } q \in \mathcal{P}_\beta\}$,
$\mathcal{P}_{\alpha\cup\beta} = \mathcal{P}_\alpha \cup \mathcal{P}_\beta$,
$\mathcal{P}_{\alpha^*} \quad = \bigcup_{i<\omega} \mathcal{P}_{\alpha^i}$.

In Process Logic, all formulas are *path formulas*, i.e., evaluated against paths. Its syntax extends Dynamic Logic with the following connectives:

– if X and Y are formulas, then so are $\mathbf{f}X$, and X **suf** Y.
– if α is a program and X is a formula, then $\langle\alpha\rangle_{PR}X$ is a formula.

The satisfaction relation $\models$ is extended to a relation between paths and state formulas. Let $p = (s_o, s_1, \ldots)$ be a path.

$p \models X \quad \Leftrightarrow \quad s_0 \models X$ for primitive (propositional or first-order) formulas X. (not explicit, but see [34, Def.4.1, p.155])
$p \models \mathbf{f}X \quad \Leftrightarrow \quad s_0 \models X$.
$p \models X$ **suf** $Y \quad \Leftrightarrow$ there is a $q \in \mathcal{P}_\alpha$ s.t.
(i) q is a proper suffix of p and $q \models Y$, and
(ii) for all r, if r is a proper suffix of p and q is a proper suffix of r, then $r \models X$.
$p \models \langle\alpha\rangle_{PR}X \quad \Leftrightarrow$ there is a $q \in \mathcal{P}_\alpha$ s.t. $pq \models X$.

Note that the semantics of $\langle\alpha\rangle_{PR}$ is different from the usual semantics of $\diamond$: $p \models \langle\alpha\rangle_{PR}X$ states that from the *endpoint* of path p, there is a path q executing α s.t. then the whole path pq satisfies X.
Transaction Logic (cf. Section 4.3) is another logic which is based on path structures; using *temporal connectives* instead of temporal *modal operators.*

Summary and Examples. While the static aspects (i.e., the states) use the same formalism (except in Hennessy-Milner Logic), i.e., propositional or first-order logic (where special approaches exist that use predicates only (Datalog) or functions only (Evolving Algebras) [30, 31]), the dynamic aspects are described differently in the above approaches as shown below:

Example 1 *Consider two persons, Alice and Bob who have bank accounts with a given balance given by a function, balance(name). Actions are debit(name, amount) and deposit(name, amount).*
Considering the excerpt of a Kripke Structure given in Figure 1, e.g.,

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

Obviously,

$$\mathcal{M}(g_1) = \{balance(Alice) = 180, balance(Bob) = 100\} \text{ and}$$
$$\mathcal{M}(g_2) = \{balance(Alice) = 180, balance(Bob) = 120\} .$$

The semantics of debit can be specified in Hennessy-Milner Logic and Dynamic Logic by

$$\forall Acc, Am_1, Am_2 :$$
$$balance(Acc_1) = Am \rightarrow [debit(Acc_1, Am_2)]balance(Acc_1) = Am_1 - Am_2$$
(where $[a]$ denotes the "always" modality ⓐ).

Both logics allow now for reasoning about sequences, e.g., expressing a simple integrity constraint that any sequence of actions $debit(Acc_1, Am)$ and $deposit(Acc_2, Am)$ keeps the sum of the overall balances unchanged.
Hennessy-Milner Logic can express this by

$$\forall Acc_1, Acc_2, Am, B_1, B_2, Sum :$$
$$Sum = balance(Acc_1) + balance(Acc_2) \rightarrow$$
$$[debit(Acc_1, Am) \cdot deposit(Acc_2, Am)]Sum = balance(Acc_1) + balance(Acc_2) \ .$$

Analogous for Dynamic Logic (with ";" as sequential concatenation).
Process Logic's transition relation $\mathcal{P}$ for programs *will e.g. contain*

$$(g_0, g_2) \in \mathcal{P}_{debit(Alice,20)\ deposit(Bob,20)} \ ,$$

making the above "transition by a composite action" explicit in the model. We also have

$$(g_0, g_1) \models \mathbf{f}(balance(Alice) = 200 \wedge balance(Bob) = 100) \wedge$$
$$\langle deposit(Bob, 20)\rangle \mathbf{last}(balance(Alice) = 180 \wedge balance(Bob) = 120)$$
with **last** *as formally defined in [34] .*

The above example shows that modal logics are useful for *reasoning about* temporal structures, especially proving correctness. Up to here, "actions" occurred only for the definition of paths that then satisfy formulas – i.e., to check if something "is true" after executing some actions. Also, the notions of actions and events are not really distinguishable, because both notions are identified with the labels of the transitions. So far, one can see an action as an action from the point of view of generating the next state, and as an event from the point of view of looking at the transition afterwards.

For *specifying* pure *evolution*, the notions of *actions*, *transactions* (that have to satisfy several requirements), and *processes* are used to describe what sequences of transitions are actually executed. When coming to *reactivity*, we want to express implications (rules) that under certain circumstances, something must be done. These circumstances can not only be static conditions, but also dynamic occurrences of *events*. The goal of these rules is to check *if something happened*, and then *to make something happen.*

3.4 Actions, Transactions, and Processes

Some of the above languages already define restricted mechanisms for (reasoning about the effects of) actions and composite actions. Other, rule-based formalisms

will be discussed in Section 4.1. There are also several frameworks and formalisms for the definition of composite actions as *programs*, *processes*, or *transactions* (which is essentially the same from different points of view and with different consequences regarding parallelism and interference). Simple "programs" have been discussed above for *Dynamic Logic.* More complex specifications of activities and interaction, e.g. between different Web nodes, can be given in terms of *Process Algebras* that are discussed just below. *Transaction Logic* is another, rule based formalism for defining, executing, and even a restricted amount of planning that will be discussed later in Section 4.3.

Process Algebras. Process Algebras describe the semantics of processes in an algebraic way, i.e., by a set of elementary processes (carrier set) and a set of constructors. The semantics can either be given as *denotational semantics*, i.e., by specifying the denotation of every element of the algebra (e.g., CSP – Communicating Sequential Processes, [35]), or as an *operational semantics* by specifying the behavior of every element of the algebra (e.g., CCS – Calculus of Communicating Systems, [45, 46]). Processes defined by Process Algebras can e.g. be used for the specification of *communication*, i.e., for basic protocols, or for defining the behavior of interacting (Semantic) Web Services (note that process algebras provide concepts for defining infinite processes), or in the action part of ECA rules.

Basic Process Algebra (BPA). For a given set $\mathcal{A}$ of atomic actions,

$$BPA_{\mathcal{A}} = \langle \mathcal{A}, \{\bot, +, \cdot\} \rangle$$

is the basic algebra – i.e., containing the least reasonable set of operators – for constructing processes over $\mathcal{A}$. $\bot$ is a constant denoting a deadlock, $+$ denotes alternative composition, and $\cdot$ denotes sequential composition: if x and y are processes, then $x+y$ and $x \cdot y$ are processes (syntax and semantics are formally introduced later on with CCS). These are essentially the processes that have also been presented above in Dynamic Logic and Hennessy-Milner-Logic.

Calculus of Communicating Systems (CCS). CCS extends BPA by more expressive operators. The carrier set of a CCS [45–47] algebra is given by a set $\mathcal{A}$ of action names from which processes are built by using several connectives. Every element of the algebra is called a *process.* By carrying out an action, a process changes into another process. Considering the modeling as an LTS, a process can be regarded as a state or a configuration. Action names become labels and the transition relation is given by the rules specifying the execution of actions. A CCS algebra with a carrier set $\mathcal{A}$ is defined as follows:

1. With X a (process) variable, X is a process expression.
2. Every $a \in \mathcal{A}$ is a process expression.
3. With $a \in \mathcal{A}$ and P a process expression, $a : P$ is a process expression (prefixing; sequential composition).

4. With P and Q process expressions, $P \times Q$ is a process expression (parallel composition).
5. With I a set of indices, $P_i : i \in I$ process expressions, $\sum_{i \in I} P_i$ is a process expression (alternative composition).
6. With $A \subseteq \mathcal{A}$ a set of actions and P a process expression, $P \upharpoonright A$ is a process expression (restriction to a set of visible actions).
7. With I a set of indices, X_i variables, P_i process expressions, $\mathsf{fix}_j \boldsymbol{XP}$ is a process expression (definition of a communicating system of processes). The fix operator binds the variables X_i, and fix_j is one of the $|I|$ processes which are defined by this expression.

The fix operator can be omitted if defining equations of the form $Q := P$ are allowed, where Q is a new process identifier and P is a process expression. Process expressions not containing any free variables are *processes*.
The (operational) semantics of a CCS algebra is given by transition rules:

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

$$, \quad \frac{P \xrightarrow{a} P'}{P \upharpoonright A \xrightarrow{a} P' \upharpoonright A} \text{ (for } a \in A) \quad , \quad \frac{P_i\{\mathsf{fix}\, \boldsymbol{XP}/\boldsymbol{X}\} \xrightarrow{a} P'}{\mathsf{fix}_i \boldsymbol{XP} \xrightarrow{a} P'} \; .$$

Additionally, there are some derived operators and constants

$$\mathbf{0} := \sum_{\emptyset} P_i \quad , \quad P_1{+}P_2 := \sum_{i \in \{1,2\}} P_i \quad ,$$

and, for asynchronous communication and delays,

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

with the corresponding transition rules

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

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

In CCS and related concepts, such as CSP [35] and ACP [6], there is no explicit notion of states, the properties of a state are given by the (sequences of) actions which can be executed.

Example 2 *Consider again the scenario of Example 1. There, it has been described how to reason about a structure. Having now a notion of processes, we can describe how transitions belong together:*

- *a common money transfer is already a simple process:*

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

- *a standing order (i.e., a banking order that has to be executed regularly) is defined as a fixpoint process. The following process transfers a given amount from one account to another every first of a month:*

$$\mathsf{fix}\ X(rec_msg(first_of_month) : debit(Acc_1, Am) : deposit(Acc_2, Am) : \partial X)$$

  (assuming the receipt of a message as a communicating action).
- *A more detailed view could e.g. communicate with the repository for checking if the balance will stay positive:*

$$\begin{aligned} \mathsf{fix}\ X(\ & rec_msg(first_of_month) : send_msg(query_(Acc_1 > Am?)) : \\ & (\partial : rec_msg(yes) : debit(Acc_1, Am) : deposit(Acc_2, Am)+ \\ & \ (\partial : rec_msg(no) : send_msg(error))) : \partial X) \end{aligned}$$

In this example, the fact that it is the 1st of a month is communicated explicitly by sending (issued e.g. by a timer process) and receiving actions.

Another way would be to consider "1st of a month" as an event *and, instead of a fixpoint process, have a rule that states "if this event occurs, then do ...". Also, for querying the account, this model uses active waiting (∂) – here it would also be possible to have a rule that* reacts *on the incoming message.*

3.5 Event Languages and Event Algebras

The main difference between actions and events is roughly that an event is the visible consequence of an action. E.g., the action is to "debit of 200 E from Alice's bank account", and visible events are "a change of Alice's bank account" (that is immediately detectable from an update), or "the balance of Alice's bank account becomes below zero" (which has to be derived from an update). Additionally, there are system events and external events like temporal events ("1st of a month") and incoming messages. Obviously, actions and events are correlated, but an action can raise several events, raising the problem how events are *detected*. In this section we focus on languages for event specification and on event detection.

In the context of the Web, an (atomic) event is in general any detectable occurrence, i.e., local events (updates, temporal events, and transactional events), incoming messages including queries and answers, updates of data anywhere in the Web, or any occurrences somewhere in an application, that are (possibly) represented in explicit data, or signaled events.

Reactivity is in general not based on atomic events only, but uses the notion of *composite events*, e.g., "when E_1 happened and then E_2 and E_3, but not E_4 after at least 10 minutes, then do A". Composite events are usually defined in terms of an *event algebra*.

So, there is the need for several integrated languages: a language for atomic events and their metadata, and (application-specific) languages for expressing the contents of different types of events, and a declarative language for describing complex events, together with algorithms for handling complex events. The latter languages are not concerned with what the information contained in the event might be, but only with types of events.

Complex Events, Event Algebras.

Event Algebras. The term "algebra" describes a very generic (mathematical) concept that has many applications in Computer Science: Boolean Algebra, Relational Algebra, or natural numbers (with only an operator succ(.), or with operators "+" and "*") are algebras. An algebra consists of a "domain" (i.e., a set of "things"), and a set of operators (with a given arity). Operators map elements of the domain to other elements of the domain. Algebra *terms* are formed by nesting operators. Each of the operators has a "semantics", that is, a definition how the result of applying it to some input should look like. **Algebra expressions** are built over basic constants and operators (inductive definition).

For an event algebra, the constants are the atomic events, and the operators serve for combining composite events, e.g.: "A and B", "A or B", or "A and then B". Event algebras contain not only the aforementioned straightforward conjunctive, disjunctive and sequential connectives, but also additional operators. A bunch of event algebras have been defined that provide also e.g. "negative events" in the style that "when E_1 happened, and then E_3 but not E_2 in between, then do something", "periodic" and "cumulative" events, e.g., [13, 55].

An Example. In [13], an event algebra which is used for event detection in the context of ECA-rules ("on ⟨event⟩ if ⟨condition⟩ do ⟨action⟩") in active databases is proposed. Semantically, an event is a predicate $E : T \to \{\mathsf{true}, \mathsf{false}\}$ where T denotes a set of time instances (or, in embedding into the model of Kripke structures, transitions where the event could be detected). For a given set of elementary events, the set of events is defined inductively:

- If E and F are events, then $E \nabla F$ and $E \triangle F$ are events.
- If $E_1, \ldots, E_n$ are events and $m < n \in \mathbb{N}$, then $\mathsf{ANY}(m, E_1, \ldots, E_n)$ is an event.
- If E and F are events, then $E; F$ is an event.
- If E_1, E_2 and E_3 are events, then $A(E_1, E_2, E_3)$ and $A^*(E_1, E_2, E_3)$ are events.
- If E_1, E_2 and E_3 are events, then $\neg(E_1)[E_2, E_3]$ is an event.

The semantics of composite events is defined as follows, where detection of a complex event means that its "final" atomic subevent is detected:

(1) $(E \nabla F)(t) \;:\Leftrightarrow\; E_1(t) \vee E_2(t)$,
(2) $(E \triangle F)(t) \;:\Leftrightarrow\; E_1(t) \wedge E_2(t)$,
(3) $(E_1; E_2)(t) \;:\Leftrightarrow\; \exists t_1 \leq t : E_1(t_1) \wedge E_2(t)$,

$$(4)\ \mathsf{ANY}(m, E_1, \ldots, E_n)(t) \ :\Leftrightarrow\ \exists t_1 \leq \ldots \leq t_{m-1} \leq t,\ 1 \leq i_1, \ldots, i_m \leq n \text{ pairwise distinct s.t. } E_{i_j}(t_j) \text{ for } 1 \leq j < m \text{ and } E_{i_m}(t)\ ,$$

$$(5)\ \neg(E_2)[E_1, E_3](t) \ :\Leftrightarrow\ E_3(t) \wedge (\exists t_1 : E_1(t_1) \wedge \wedge (\forall t_2 : t_1 \leq t_2 < t : \neg(E_2(t_2) \vee E_3(t_2))))\ ,$$

$$(6)\ A(E_1, E_2, E_3)(t) \ :\Leftrightarrow\ E_2(t) \wedge (\exists t_1 : E_1(t_1) \wedge (\forall t_2 : t_1 \leq t_2 < t : \neg E_3(t_2)))\ ,$$

$$(7)\ A^*(E_1, E_2, E_3)(t) \ :\Leftrightarrow\ \exists t_1 \leq t : E_1(t_1) \wedge E_3(t)\ ,$$

when this event occurs, a specified action for every occurrence of E_2 has to be executed in t.

The constructs ∇ ("or") and $\triangle$ ("and") are standard and straightforward. "$(E_1; E_2)$" denotes the successive occurrence of E_1 and E_2, where in case that E_2 is a complex event, it is possible that subevents of E_2 occur *before* E_1 occurs. ANY denotes the occurrence of m events out of n in arbitrary order, which is also expressible by a special ∇-$\triangle$-;-schema. (5) is a complex event which detects the non-occurrence of E_2 in the interval between E_1 and the next E_3. (6) is an "aperiodic" event which is signaled whenever E_2 occurs after E_1 without E_3 occurring in-between. Note that $\neg(E_2)[E_1, E_3]$ occurs with the *terminating* event E_3 whereas $A(E_1, E_2, E_3)$ occurs with *every* E_2 in the interval (if E_3 never occurs, the interval is endless). The "cumulative aperiodic event" (7) occurs with E_3 and then requires the execution of a given set of actions corresponding to the occurrences of E_2 in the meantime. Thus, it is not a simple event, but more an active rule, stating a temporal implication of the form "if E_1 occurs, then for each occurrence of an instance of E_2, collect its parameters, and when E_3 occurs, report all collected parameters (in order to do something)".

In general, events are parameterized, and event specifications contain free variables that are bound in the event detection.

Example 3 *Consider again the scenario of Example 1. Events in this scenario are e.g. "1st of a month", "a deposit to account x" (in this case, the event directly corresponds to an action), or "the balance of account x goes below zero" (due to a debit action). Composite events are e.g.,"there were no deposits to an account for 100 days" which can be expressed in the above event language as*

$$E_1(Acct) := (\neg(\exists X : deposit(Acct, X)))[deposit(Acct, Am) \wedge t = date, date = t{+}100days]\ .$$

An aperiodic event is e.g. "the balance of account x goes below zero and there is another debit without any deposit in-between" which is expressed in the above event language as

$$E_2(Acct) := A(\ debit(Acct, Am_1) \wedge balance(Acct) < 0,\ debit(Acct, Am_2), deposit(Acct, Am_3))\ .$$

A cumulative periodic event is e.g. used for "after the end of a month, send an account statement with all entries of this month:

$$E_3(Acct, list) := A^*(first_of_month, (debit(Acct, Am) \nabla deposit(Acct, Am)), first_of_next_month)$$

where the event occurs with $first_of_next_month$ and carries a list of the debit and deposit actions.

Event Detection. As described above, an event algebra mainly consists of the definition of event combinators and their semantics (given in general in terms of sequences that "satisfy" the composite event). For practical issues, it is necessary to *detect* the event. Since events are *volatile* data (and for efficiency reasons), it is not possible to do this by querying, but event detection must be done incrementally *on-the-fly*.

Work on complex events does not only define the semantics of events and complex events, but in general also describes algorithms for efficient detection and tracing of events. Incremental *residuation* has been used in [55] in an approach that uses an event-style algebra for *scheduling* of tasks, which is similar to the reduction steps in the operational semantics given in Section 3.4 for CCS.

Past tense modalities with incremental bookkeeping have been employed for checking temporal constraints and temporal conditions in ECA-style rules, e.g. in [15, 56]. [15] uses full first-order past temporal logic, with $\exists$ and $\forall$ quantifiers. [56] replaces the quantifiers by a functional assignment $[X \leftarrow t]\varphi(X)$ that binds a variable X to the value of a term t in a given state. This ensures safety of formulas, but the full expressiveness of using a universal quantifier is not provided.

From the theoretical point of view, the used techniques amount to the same principles, although formalized differently by *residuation*, *automata*, *graph techniques* and *rewriting*.

They are in general restricted to the area of distributed/active databases where the location and communication of events is fixed.

Example 4 *Consider again the scenario of Example 3.*

Detection of the first event means to start event detection of $E_1(Acct)$ (with internal parameter t_0) for Acct whenever a deposit occurs at timepoint t_0. The detection ends when either another deposit to Acct occurs, or the date t_0+100 days is reached, and the event actually occurs.

Detection of the second event means to start event detection of $E_2(Acct)$ whenever a debit occurs and the resulting balance is below zero. The detection ends when either a deposit to Acct occurs (then, the event is not reported), or another debit happens (then, the event is actually detected and reported).

Detection of the third event means to start event detection of $E_3(Acct)$ at the first day of a month. Internal bookkeeping is done for every debit or deposit (action/event), and the event finally occurs at the first day of the next month.

Considering an event "(if) balance of Account changes (then immediately phone me)", the detection means to translate it as "either a debit or a deposit occurs".

In the (Semantic) Web, event detection requires to solve two issues:

- translating (atomic) event specifications into underlying actions (as the final one in the above example), and

– detecting remote events. Up to now, events were always considered to be local, or at least communicated explicitly by messages.

3.6 Combining Static and Dynamic Aspects

It is desirable that event sequences can be combined with requirements on the state of resources at given intermediate timepoints, e.g. "when at timepoint t_1, a cancelation comes in and somewhere in the past, a reservation request came in at a timepoint when all seats were booked, then, the cancelation is charged with an additional fee". In this case, the event detecting engine has to state a query at the moment when a given (sub)event is detected. For being capable of describing these situations, a formalism (and system) that deals with sequences of events and queries is required. This is not covered by the above approaches (except in some extent [15]).

A language that covers these issues will be presented in Section 4.3: Transaction Logic.

4 Rule-based Languages for Evolution and Reactivity

Rule-based languages for evolution and reactivity can mainly be grouped into two aspects:

– languages defining individual actions directly in terms of their effects on a structure. These languages can immediately be used for "programming" and reasoning.
– languages defining the higher-level interplay of actions, i.e., when and how a certain sequence of actions has to be executed. Here also *transaction* models have to be considered.

4.1 Action Languages

Action languages are formal models that are used for representing actions and for reasoning about the effects of actions [53, 4, 24, 25, 27–29, 18] that have been mainly developed in the Knowledge Representation and Reasoning community.

Central to this method of formalizing actions is the concept of a labeled transition system (LTS). Usually, the states are first-order structures, where the predicates are divided into static and dynamic ones, the latter called *fluents* (cf. [54]). Action programs (in languages such as language $\mathcal{B}$ and $\mathcal{C}$, below) are sets of sentences that describe the transitions by specifying which dynamic predicates change in the environment after the execution of an action. Evolving Algebras/Abstract State Machines, also described below, are a special kind of action programs.

Usual problems here are to predict the consequences of the execution of a sequence of (sets of) actions, or to determine a set of actions implying a desired conclusion in the future (planning). Several action query languages exist that

allow for querying such a transition system, going beyond the simple queries of knowing what is true after a given sequence of actions has been executed (allowing e.g. to reason about which sets of actions lead to a state where some goal is true, i.e. planning as in [18]).

Situation Calculus. The first, and most prominent concept here is the situation calculus (originally in [44], reprinted in [43], see also [53]).

States (or situations) are elements of the domain, occurring as an argument for distinguished predicates $\mathsf{holds}(p(x), s)$ and $\mathsf{occurs}(a(x), s)$ where p is a predicate of the application domain and a is an action. Events (mainly equivalent to actions) in a situation produce new situations: $\mathsf{do}\ a(s)$ denotes the situation which is obtained by executing an action a in a situation s. A situation is a first order functional term $\mathsf{do}\ a_n(\mathsf{do}\ a_{n-1}(\dots(\mathsf{do}\ a_1(s_0))))$, where a_i are actions and s_0 is a constant denoting the initial situation; the values of fluents in s_0 are specified by formulas of the form $\mathsf{holds}(p(x), s_0)$.
Actions are characterized by *preconditions*, e.g.

$$\mathsf{occurs}(a(x), S) \rightarrow \mathsf{holds}(p(x), S)$$

and their *normal effect*, e.g.

$$\mathsf{holds}(p(x), S) \wedge \mathsf{occurs}(a(x), S) \rightarrow \mathsf{holds}(q(y), do\ a(x)(S))$$

describing how an action changes some fluents. The frame problem is solved by adding axioms for assuming that fluents which are not explicitly changed, remain unchanged.

There exist different versions of the situation calculus, e.g., the one used in GOLOG [40], a logic programming language. There, the predicate holds is omitted and the preconditions are characterized by a distinguished predicate, i.e. $\mathsf{Poss}(a(x), s) \equiv \mathsf{holds}(p(x), S)$. In GOLOG, frame axioms are stated explicitly.

Statelog. Statelog [38] provides a logical framework for active rules which precisely and unambiguously defines the meaning of rules. Moreover, it allows to study fundamental properties of active rules like termination, confluence and expressive power.

A *Statelog rule* r is an expression of the form

$$[S{+}k_0]\,H \;\leftarrow [S{+}k_1]\,B_1,\;\dots\;,[S{+}k_n]\,B_n$$

where the head H is a Datalog atom, B_i are Datalog literals (atoms A or negated atoms $\neg A$), and $k_i \in \mathbb{N}_0$. Access to different database states is accomplished via *state terms* of the form $[S{+}k]$, where $S{+}k$ denotes the k-fold application of the unary function symbol "+1" to the *state variable* S. A rule r is called *local* it $k_0 = k_i$ for all $i = 1, \dots, n$, *progressive*, if $k_0 \geq k_i$ for all $i = 1, \dots, n$, and *1-progressive*, if $k_0 = k_i{+}1$ for all $i = 1, \dots, n$. A *Statelog program* is a finite set of progressive Statelog rules. In general, the rules of a Statelog program

define a sequence of (intermediate) *transitions*. A Statelog activity (raised by an external event that makes some progressive rule applicable) ends when no more progressive rules are applicable. Then the system is idle until the next external event occurs.

Logic Programming notions (e.g., *local stratification*) and declarative semantics (e.g., *perfect model*) developed for deductive rules can be applied directly to Statelog. It uses a notion of *state-stratified semantics* as the canonical model of a Statelog program wrt. a given database state: P is called state-stratified, if there are no negative cyclic rule dependencies *within a single state*. This notion is closely related to *XY-stratification* [65] and *ELS-stratification* [36].

Language $\mathcal{B}$. The $\mathcal{B}$ language [27, 25] is a generalization of the so-called language $\mathcal{A}$ [24](which itself represents the propositional fragment of the ADL formalism [50]). It allows conditional and non-deterministic actions and, unlike $\mathcal{A}$, also for the representation of actions with indirect effects. A program in $\mathcal{B}$ is a set of static and dynamic laws, of the forms, respectively:

$$L \textbf{ if } F, \quad \text{and}$$
$$A \textbf{ causes } L \textbf{ if } F$$

where L is a fluent literal, F a conjunction of literals, and A an action name. Intuitively a static law states that every possible state satisfying the conjunction F must also satisfy L, and a dynamic law states that if F is satisfied when action A occurs then L is true in the subsequent state. Given a set of static and dynamic laws, a labeled transition system is defined. Basically, states are all interpretations closed under the static laws, and there is an arc from a state s to a state s' with label a iff all Ls of dynamic rules of the form a **causes** L **if** F, where F holds in s, belong to s', and nothing else differs from s to s'.

A program in $\mathcal{A}$ is as in $\mathcal{B}$ but without static laws. Besides the above briefly described language $\mathcal{B}$, several other extension of the language $\mathcal{A}$ exist. Language $\mathcal{AR}$ [26], as for $\mathcal{B}$, also allows for modeling indirect effects of actions but in this case, instead of static laws, constraints of the form **always** F, where F is a propositional formula, are used. Language $\mathcal{A}_K$ [57] further extends $\mathcal{AR}$ for formalizing sensing actions (i.e. actions for determining the truth value of fluents). Another extension of $\mathcal{A}$ is the language $\mathcal{PDL}$ [41] which is particularly tailored for specifying policies. A survey and comparisons on extensions of $\mathcal{A}$ can be found in [19].

Language $\mathcal{C}$. As in language $\mathcal{B}$, also in $\mathcal{C}$ [29, 28] statements of the language are divided into static and dynamic laws. The main distinction between $\mathcal{C}$ and $\mathcal{B}$, besides the fact that $\mathcal{C}$ allows for arbitrary formulas to be caused by actions (rather than simply literals as in $\mathcal{B}$) and arbitrary formulas as conditions (rather than conjunction only), is that $\mathcal{C}$ distinguishes between asserting that a fluent "holds" and making the stronger assertion that "it is caused", or "has a causal explanation".

A program in $\mathcal{C}$ is a set of static and dynamic laws of the forms, respectively:

$$\textbf{caused } F \textbf{ if } G, \quad \text{and}$$
$$\textbf{caused } F \textbf{ if } G \textbf{ after } U$$

where F and G are formulas over fluent literals, and U is a formula with both fluent literals and action names.

Intuitively, a static law states that the formula G causes the truth of the formula F, and a dynamic rule states that after U, the static rule "**caused** F **if** G" is in force. The definition of a transition system for $\mathcal{C}$ is based on *causal theories* [28]. The idea behind causal theories is that something is true iff it is caused by something else. Every state s is now characterized by a set $M(s)$ and a causal theory $T(s)$ (consisting of the static rules from P and possibly additional ones). Given a causal theory T and a set M of fluents, the causal theory T_M of formulae is defined as follows:

$$T_M = \{F \mid \text{"}\textbf{caused } F \textbf{ if } G\text{"} \in T \text{ and } M \models G\}$$

We say M is a *causal model* of T iff M is the unique model of T_M. For all states s in the LTS, the interpretation $M(s)$ must be a causal model of $T(s)$.

Given a state s with $T(s)$, $M(s)$ and a set of actions K (executed in a transition), the resulting causal theory $T(s, K)$ is given by the static laws of P and the static laws enforced by the dynamic laws whose preconditions are true in $M(s) \cup K$. Then there is an arc with label K between s and a state s' iff $T(s') = T(s, K)$ and $M(s')$ is a causal model of $T(s, k)$. It is worth noting that in $\mathcal{C}$, contrary to $\mathcal{B}$, fluent inertia is not assumed by default.

Various extensions to $\mathcal{C}$ have recently appeared in the literature. Most prominently, the language $\mathcal{C}$++ [28] and the language $\mathcal{K}$ [18]. $\mathcal{C}$++ further allows for multi-valued, additive fluents which can be used to encode resources and allows for a more compact representation of several practical problems. The language $\mathcal{K}$ allows for representing and reasoning about incomplete states, and for solving planning problems.

For more details on these languages, as well on the implementation of fragments of them in logic programming, see [19, 25, 28].

Evolving Algebras/Abstract State Machines. The concept of "Evolving Algebras" has been introduced for specifying the operational semantics of processes in [30, 31]. Evolving Algebras have originally not been introduced from the logical point of view, but for describing the operational semantics of processes in the sense of Turing's Thesis: "Every algorithm can be described by a suitable Evolving Algebra". Thus, for any given algorithm, on any level of abstraction an Evolving Algebra can be given.

In universal algebra, a first-order structure over a signature where the equality symbol is the only relation symbol (i.e., everything is represented by functions), is called an *algebra*.

The *signature* Σ of an Evolving Algebra is a finite set of function symbols, each of them with a fixed arity, including 0-ary constants. Note that every relation can be represented by its characteristic function. The names in Σ are divided into two groups: static and dynamic functions (i.e., *fluents* as in e.g., Situation Calculus [53], GOLOG [40], also [54]). A state of an Evolving Algebra over Σ is then an interpretation of Σ, inducing an evaluation of terms.

An Evolving Algebra *EA* is given by an initial state *Z(EA)* (which also determines the interpretation of the static function symbols for all states) and a program *P(EA)* (a set of transition rules and rule schemata) describing the change of the interpretation of state-dependent function symbols in a Pascal-like syntax.

An *elementary update rule* is an update of the interpretation of a function symbol at one location: $f(t_1, \ldots, t_n) := t_0$, where f is an n-ary function symbol and t_i are terms.

The set of rules is defined by structural induction by defining blocks and conditionals (if-then); also rule schemata that contain free variables are allowed. A program *P(EA)* of an Evolving Algebra *EA* is a finite set of rules and rule schemata. A program is then executed by applying rules, inducing again a Kripke structure.

4.2 Event-Condition-Action Rules in Databases

Event-Condition-Action rules have already been motivated in Section 2.2 as a common means to express system behavior. They are intuitively easy to understand, and provide a well-understood formal semantics: when an event occurs, evaluate a condition, and if the condition is satisfied then execute an action. Above, we have discussed several approaches for the event and action parts. Additionally, several execution models can be chosen that specify how the rule is applied (before or after or deferred, statement-oriented or set-oriented, its transactional embedding etc.), modified by further policies of the ECA engine (e.g. for conflict resolution).

Depending on the choice of the above sublanguages and semantics, a broad range of behaviors can be designed. ECA languages based on atomic events are e.g. used for maintaining consistency (as in the well-known SQL triggers) in course of execution of a surrounding process. On the other end of the range, ECA languages that allow for complex events can themselves be used for *specifying* the behavior of a system in a rule-based way.

Types of ECA rules. Mainly, two kinds of ECA rules can be distinguished:

- low-level: rules that react directly on changes of the underlying data. These are provided as triggers in most database systems, e.g., SQL, of the form

  ```
  ON database-update WHEN condition BEGIN pl/sql-fragment END
  ```

  where the values of the updated tuple are accessible as `old` and `new`.
- application-level: ECA engines that react on application-level events that are raised by updates of underlying data, messages etc.

4.3 Transaction Logic

Transaction Logic $\mathcal{TR}$ [8] is another comprehensive rule-based formalism that does not have a strict ECA distinction, but follows the Logic Programming style. In $\mathcal{TR}$, in contrast to modal logic where states are given as first-order structures, states are given as abstract *theories* over a signature $\mathcal{L}$ – that can e.g. be first-order theories, or OWL-based worlds. The evaluation of formulas wrt. states is provided by an abstract *state data oracle* $\mathcal{O}^d$ that answers queries (possibly with free variables) for every individual state. Transitions are given by the *state transition oracle* $\mathcal{O}^t$ which maps pairs of database states to sets of ground formulas (over a set $\mathcal{A}$ of action names, corresponding to the labels of the elementary transitions). Thus, with $\mathcal{G}$ denoting the set of state identifiers, $(\mathcal{G}, \mathcal{O}^t, \mathcal{O}^d)$ gives the same information as a labeled Kripke structure, an LTS, or a path model for Process Logic.

Since in Transaction Logic, the internal representation and model of states is not predetermined, structures of any type are allowed as a basis. For example, a pure functional signature (e.g. static algebras), a pure relational signature (Datalog), first-order, or even object-oriented (F-Logic) or OWL models can be used in the data oracle.

Formally, the semantics of $\mathcal{TR}$ formulas is based on a version of *path structures*, i.e., the satisfaction of formulas is defined on paths, not on states: A *path* of length $k \geq 1$ is a finite sequence $\pi = \langle \mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_k \rangle$ of state identifiers; $\pi_1 \circ \pi_2 = \langle \mathcal{D}_1, \ldots, \mathcal{D}_i \rangle \circ \langle \mathcal{D}_i, \ldots, \mathcal{D}_k \rangle$ is a *split* of π.
A path structure (here: over a first-order $\mathcal{L}$) $\mathcal{M}$ is a triple $\langle \mathcal{U}, \mathcal{I}_\mathcal{F}, \mathcal{I}_{path} \rangle$ where

- $\mathcal{U}$ is the *domain* of $\mathcal{M}$,
- $\mathcal{I}_\mathcal{F}$ is a (state-independent) interpretation of the function symbols in $\mathcal{L}$,
- $\mathcal{I}_{path}$ assigns to every path $\pi = \langle \mathcal{D}_1, \ldots, \mathcal{D}_n \rangle$ a semantic structure $\langle \mathcal{U}, \mathcal{I}_\mathcal{F}, \mathcal{I}_\mathcal{P} \rangle$ where $\mathcal{I}_\mathcal{P}$ is an interpretation of the predicate symbols in $\mathcal{L} \cup \mathcal{A}$.

$\mathcal{I}_{path}$ is subject to two restrictions:

- Compliance with the data oracle: $\mathcal{I}_{path}(\langle \mathcal{D} \rangle) \models \phi$ for every $\phi \in \mathcal{O}^d(\mathcal{D})$,
- Compliance with the transition oracle: $\mathcal{I}_{path}(\langle \mathcal{D}_1, \mathcal{D}_2 \rangle) \models a$ whenever $a \in \mathcal{O}^t(\mathcal{D}_1, \mathcal{D}_2)$.

Transaction formulas are built by the connectives $\neg, \vee, \wedge, \oplus, \otimes$, and the quantifiers $\exists$ and $\forall$. Let π be a path and β a variable assignment. Then,

for formulas of the state language $\mathcal{L}$:

$$
\begin{array}{lcl}
(\mathcal{M}, \langle \mathcal{D} \rangle, \beta) \models_{\mathcal{TR}} s\text{–}fml & \Leftrightarrow & (\mathcal{I}_{path}(\langle \mathcal{D} \rangle), \beta) \models_{(\mathcal{O}^d, \mathcal{O}^t)} s\text{–}fml \\
 & \Leftrightarrow & s\text{–}fml \in \mathcal{O}^d(\mathcal{D}) \ ,
\end{array}
$$

for formulas of the transition language $\mathcal{A}$:

$$
\begin{array}{lcl}
(\mathcal{M}, \langle \mathcal{D}_1, \mathcal{D}_2 \rangle, \beta) \models_{\mathcal{TR}} t\text{–}fml & \Leftrightarrow & (\mathcal{I}_{path}(\langle \mathcal{D}_1, \mathcal{D}_2 \rangle), \beta) \models_{(\mathcal{O}^d, \mathcal{O}^t)} t\text{–}fml \\
 & \Leftrightarrow & t\text{–}fml \in \mathcal{O}^t(\mathcal{D}_1, \mathcal{D}_2) \ , \\
(\mathcal{M}, \pi, \beta) \models_{\mathcal{TR}} \phi \otimes \psi & \Leftrightarrow & (\mathcal{M}, \pi_1, \beta) \models_{\mathcal{TR}} \phi \text{ and } (\mathcal{M}, \pi_2, \beta) \models_{\mathcal{TR}} \psi \\
 & & \text{for some split } \pi = \pi_1 \circ \pi_2 \text{ of } \pi \ , \text{ and} \\
(\mathcal{M}, \pi, \beta) \models_{\mathcal{TR}} \phi \oplus \psi & \Leftrightarrow & (\mathcal{M}, \pi_1, \beta) \models_{\mathcal{TR}} \phi \text{ or } (\mathcal{M}, \pi_2, \beta) \models_{\mathcal{TR}} \psi \\
 & & \text{for } \textit{every} \text{ split } \pi = \pi_1 \circ \pi_2 \text{ of } \pi \ .
\end{array}
$$

Due to the restriction of $\mathcal{O}^t$ to elementary actions, parallel composition of actions in a single transition is not possible. In [9], an interleaving semantics for parallelism is given.

Example 5 *Consider again the states from Example 1 (note that the function balance is interpreted state-dependently), consider the path $\pi := \langle g_0, g_1, g_2 \rangle$. For each state g_i, $\mathcal{O}^d(g_i)$ is the theory induced by $\mathcal{M}(g_i)$. The transition oracle $\mathcal{O}^t$ represents the transition relation $\mathcal{R}$, i.e., $\phi \in \mathcal{O}^t(g, g')$ if and only if $(g, g') \in \mathcal{R}(\phi)$ for action literals ϕ:*

$$\mathcal{O}^t(g_0, g_1) = \{debit(Alice, 20)\} \quad \text{and} \quad \mathcal{O}^t(g_1, g_2) = \{deposit(Bob, 20)\} \ .$$

We have $(\mathcal{M}, \pi) \models debit(Alice, 20) \otimes deposit(Bob, 20)$ and also – mixing state and transition queries

$$(\mathcal{M}, \pi) \models (balance(Alice){+}balance(Bob) = 300) \otimes debit(Alice, 20) \otimes deposit(Bob, 20) \otimes (balance(Alice){+}balance(Bob) = 300) \ .$$

ECA Semantics by Serial Implication. In the same way as standard implication is derived from disjunction as $A \rightarrow B \Leftrightarrow \neg A \vee B$, *(right) serial implication* is defined as $A \Rightarrow B \Leftrightarrow \neg A \oplus B$ (which is the main application of the serial disjunction). With this, temporal constraints in the style of ECA rules can be defined.

Example 6 *Consider again Examples 1 and 5. The rule "if there is a debit and the resulting balance is below zero, then send a message" is specified by*

$$debit(Acct, Am) \otimes balance(Am) < 0 \Rightarrow sendmsg(...) \ .$$

Analogously, *left serial implication* allows for stating preconditions.

Transaction Bases and the Serial Horn Fragment. A *transaction base* is a set of formulas of the form $a_0 \leftarrow a_1 \otimes \ldots \otimes a_n$, which play a special role for *Transaction Logic programming*, providing a top-down SLD-style proof procedure. With such rules, *transactions* can be defined, providing a declarative specification of the database evolution. To execute a_0 (in LP terminology: to prove a_0) in a state $\mathcal{D}$ means to execute or prove $a_0 \leftarrow a_1 \otimes \ldots \otimes a_n$ by generating intermediate states; thus, the final state $a_0(\mathcal{D})$ is "specified" as $a_1 \otimes \ldots \otimes a_n(\mathcal{D})$. Note that depending on the "nature" of the a_i they can denote events (that cannot be forced, but whose presence constrains the ways to make the body true), conditions on states (also acting as constraints), or actions (which are then to be executed/proven as heads of other rules).

Example 7 *Consider again Examples 1 and 5. The "money transfer" transaction is defined as*

$$transfer(Am, Acc_1, Acc_2) \leftarrow debit(Acc_1, Am) \otimes deposit(Acc_2, Am) \ .$$

Consider now the case that debit and deposit are not atomic actions, but instead there is an underlying (relational) database with a table $balance(Acct, Amount)$, manipulated by delete and insert actions. Then, debit and deposit actions can be specified by their effect on balance:

$$debit(Acc, Am) \leftarrow$$
$$\quad balance(Acc) = N \otimes balance.delete(Acc, N) \otimes balance.insert(Acc, N{-}Am)$$
$$deposit(Acc, Am) \leftarrow$$
$$\quad balance(Acc) = N \otimes balance.delete(Acc_1, N) \otimes balance.insert(Acc, N{+}Am) \ .$$

The combination of the above mechanisms for expressing and enforcing constraints, expressing ECA rules, defining transactions, planning with SLD resolution and further features of Transaction Logic provides a very expressive framework.

4.4 Transactional Requirements

In general, evolution consists not of arbitrary execution of independent actions, but of execution of certain processes (e.g. defined as a process algebra in Section 3.4). The interaction between these processes can be more or less close:

- each of them runs mainly on independent data, doing only some communication as provided by CCS, or
- processes run on shared data.

In both cases, besides communication/cooperation there are certain additional requirements. Here, the database community uses the notion of *transactions* for guaranteeing correct behavior. Usually, transactions adhere to the ACID paradigm:

Atomicity: A transaction is (logically) a unit that cannot be further decomposed: its effect is *atomic*, i.e., all updates are executed completely, or nothing at all ("all-or-nothing").

Concistency: A transaction is a correct transition from one state to another. The final state is not allowed to violate any integrity condition (otherwise the transaction is undone and rejected).

Isolation: Databases are multi-user systems. Although transactions are running *concurrently*, this is hidden against the user (i.e., after starting a transaction, the user does not see changes by other transactions until finishing his transaction, *simulated* single-user).

Durability: If a transaction completes successfully, all its effects are durable (=persistent).

In the Web environment, not only "simple" transactions, but also *long transactions* and *hierarchical transactions* are used.

Summary. The previous sections – especially those on Kripke Structures and Modal Logics, ECA rules and Transaction Logics – also illustrate the duality

between seeing evolution as a rule-driven process, and describing it declaratively via constraints: Modal Logics are primarily used for *describing* structures via their constraints and *reasoning* about them. The semantics of ECA rules is targeted to *generate* such a structure (or even more, to generate one possible path and proceed along it, moving and forgetting from one state to the next). Transaction Logic can be interpreted as both ways: reasoning about possible paths, and, as *Transaction Logic Programming*, running an evolving system. Moreover, temporal logics etc. allow also to reason about systems of ECA rules (correctness, termination etc.).

5 Evolution and Reactivity on the Web

The previous two sections have introduced the abstract concepts and some sample formalisms for handling evolution and reactivity. In this section, we present the current basis and prerequisites for extending and applying these concepts to the Web and to the Semantic Web. The high-level concepts like Kripke structures, modal logics, rules, ECA rules, event algebras with event detection mechanisms, and transactions apply with slight adaptations to the Semantic Web. They have to be *instantiated* for this environment: What are the *actions* and *events* in this setting? What syntactical and semantical frameworks are used for the high-level concepts?

In today's Web environment, XML (as a format for storing and exchanging data), RDF (as an abstract data model for states), OWL (as an additional framework for state theories), and communication issues (Web Services, SOAP, WSDL) provide the natural underlying concepts.

5.1 States and Nodes in the Semantic Web

In the Semantic Web as a network of autonomous nodes, there is not a single "state", but the notion of "current state" has to deal with different data models, incompleteness, and inconsistency (which is dealt with in another chapter of this volume on *querying*). Thus, every node has its own current view of the global state. The same holds for events: only events that are somehow known to the node can be considered.

The knowledge of a node in the Semantic Web is represented in RDF, RDFS, and/or OWL. OWL provides a model theory, thus, instead of first-order structures and first-order logic used in classical approaches, Kripke structures and logics for OWL are a prospective basis.

In the Semantic Web, the state of a node in this setting consists of the common notion of "state wrt. an application", and additionally derivation rules and behavioral rules. In a wider sense also the state of event detection algorithms belongs of the state of a node. Preferably, all this is expressed in RDF/OWL; larger internal databases are actually stored in plain XML, but mapped to an RDF/OWL ontology.

Thus, actions have to be able to change this state: XML updates, RDF updates, ontology updates, and service calls.

5.2 Existing Languages for Updates

XML Updates. There are several proposals for languages that provide update capabilities for XML data. Usually, update languages are designed as an extension of a query language with update capabilities. At least, an addressing mechanism for selecting parts of XML documents that are to be modified is needed.

XML:DB's XUpdate. XUpdate [64] is an update language developed by the XML:DB group, its latest language specification was released in late 2000 as a working draft. Note that, at that time, the query languages XPath, XQL, and XML-QL and the transformation language XSLT were already defined, but XQuery did not yet exist. Thus, also the name "XUpdate" is not related to XQuery. Similar to XSLT, XUpdate is written in XML syntax and makes use of XPath [62] expressions for selecting nodes to be processed afterwards. Simple atomic update operations to XML documents are possible with XUpdate. Several XML database systems implement this language, e.g. eXist [21].

XQuery Update extensions. A proposal to extend XQuery [63] with update capabilities is presented in [59]. XQuery is extended with a `FOR ... LET ... WHERE ... UPDATE ...` structure. The `UPDATE` part contains specifications of update operations (i.e. delete, insert, rename, replace) that are to be executed in sequence. For ordered XML documents, two insertion operations are considered: insertion before a child element, and insertion after a child element. Using a nested `FOR...WHERE` clause in the `UPDATE` part, one might specify an iterative execution of updates for nodes selected by an XPath expression. Moreover, by nesting update operations, updates can be expressed at multiple levels within an XML structure. Update operations very similar to those described in [59] have been specified and implemented in [39], extended e.g. by means to specify conditional updates. The solution has been incorporated into Software AG's Tamino[3] product.

XChange. XChange [11] is a declarative language for specifying evolution of data on the (Semantic) Web. XChange builds upon Xcerpt [10, 12], a declarative query and transformation language for the (Semantic) Web. The XChange update language uses rules to specify *intensional updates*, i.e. a description of updates in terms of queries.

RDF Updates. Basically, languages for RDF updates are built in the same way as for XML and SQL updates by extending a query language. There is not yet a definitive decision about an RDF query+update language.

[3] `http://www.tamino.com`

5.3 Atomic Events in the Semantic Web

In the context of the (Semantic) Web, the *global handling* of events must also be investigated. In addition to local events, there are remote and "global" events. Similar to the classical case, there are (local) data level events and rules, and (local, remote, and global) application-level ones. The notion of composite events is then defined as usual.

Local Events. Local events are comparable with those discussed before for the classical case: temporal events, receipts of messages, local data level events and local application level events. Data level events are e.g. updates of underlying XML or RDF repositories (we discuss the concrete syntax and semantics later). Application-level events in the Semantic Web are also described or translatable to RDF (a special case are e.g. SOAP calls).

Remote Events. As illustrated above, detection of, and reaction upon, remote events is an important feature for the Semantic Web. An event detection engine must also be able to detect/discover remote events that are not explicitly communicated. This is especially the case when working with complex events (see below). It can be done by using remote event bases (when the location of an event is known, and it is known that it is traced in an event base), or by regularly polling remote data (e.g. fuel prices at my favorite petrol station, or stock courses). In this case, again, publish/subscribe systems or continuous-query services can be applied (especially, when they maintain a history).

For concrete *atomic events*, it must also be distinguished between the event itself (carrying application-specific information), and its metadata, like the type of event (update, temporal event, receipt of message, ...), time of occurrence, the time of detection/receipt (e.g., to refuse it, when it had been received too late) and the event origin or its generator (if applicable; e.g. in terms of its URI).

Implicit Events. Most of the events *can* be expressed alternatively as detection of updates of a given database (communicated via publish/subscribe systems), or by queries but, especially in the *Semantic* Web, a declarative specification from the point of view of an application-level *event* is intended. The reduction of the detection to an actual update is then left to the semantic component.

Example 8 (Events) *Consider the situation when Oracle bought the Retek company on 22.3.2005; 11.25 $ per share, 631 million Dollar total. Firstly, this is an application-level real world event. It is noted by the (Semantic) Web e.g. as a (local, low-level) database event at New York Stock Exchange as a database update at 09:00 h AST.*

Stock tickers and agents will immediately be informed by push *propagation. An agent in Europe receives a message (raising a local* incoming message event*) sent at 9:01h AST, received at 14:02h MET, coming from NYSE (trustable), with RDF body, containing the above facts. Analyzing the message body, the agent detects the application level event that Oracle bought Retek together with the detailed financial facts.*

Possible other events that are "detected" in turn by this agent that is probably running investment rules are e.g., that "Oracle bought some company", "an IT company has been bought", "SAP did not succeed in buying Retek", etc., possibly contributing to the detection of composite events.

The original message is also posted to a "Semantic Web Newspaper" service, where smaller clients poll messages e.g. in the evening. For such a client, the incoming event consists e.g. of the information that "at 20:32 PST, I became aware that at 9:00 AST, ...". It can now process the pure facts (that probably explain why the oracle stocks raised/fell during the day), or incorporate the awareness time, e.g., when processing a rule "if I become aware of a large acquisition less than 3 hours after the fact, do something ...".

5.4 ECA Rules in the Semantic Web

There are several abstraction levels on which active rules can be defined:

- programming language level: triggers as built-in constructs of a given database model, like SQL triggers. Usually they are implemented inside the database. This level can e.g. directly be based on the *DOM Level 2/3 Events* [17] or on the triggers of relational storage of RDF data.
- logical level – XML. Here ECA rules consist of distinguished event-condition-action parts that are also marked up in XML/RuleML; one of the results of the research in I5 (jointly with I1) should be an ECA-ML language.

  This requires a definition of atomic update events on XML data; probably on the same level and granularity as updates in XUpdate located by XSL patterns or by using an update language like XChange.
- semantic level: RDF. Here, several aspects can (also independently) be lifted from XML:
  - use XML-ECA rules on underlying RDF/OWL data,
  - use RDF/OWL descriptions of events, conditions, and actions in the XML-ECA framework,
  - use an RDF/OWL ontology even on the rule level. (Conversely, rules in this ontology can themselves use event/condition/action parts in XML, and even data in XML).

Updates and Actions. There are different ways how to express the actions to be taken.

- Explicit updates: In this case the action is an explicit update statement e.g. described in XUpdate, XQuery+Updates, XChange, or in an RDF Update language. This requires knowledge of the underlying schema.
- Explicit actions: In this case by calling a procedure/method (SOAP),
- Semantic/Intensional: This requires the declarative specification of what has to be changed, using an RDF/OWL ontology of changes (to RDF data).

5.5 Trigger-Like Local ECA Rules

Trigger-like local ECA rules have to react directly on the changes of the database, which is assumed to be in XML or RDF format. While triggers in relational databases/SQL were only able to react on changes of a given tuple or an attribute of a tuple, the XML and RDF models call for more expressive event specifications according to the (tree or graph) structure.

XML. Work on triggers for XQuery has e.g. been described in [7] with **Active XQuery** and in [3], emulating the trigger definition and execution model of the SQL3 standard that specifies a syntax and execution model for ECA rules in relational databases. *Active XQuery* uses the same syntax and switches as SQL's `CREATE TRIGGER`.

The following proposal has been developed in [2]: For modifications of an XML tree, the following atomic events could be considered:

- `ON DELETE OF` *xsl-pattern*: if a node matching the *xsl-pattern* is deleted,
- `ON INSERT OF` *xsl-pattern*: if a node matching the *xsl-pattern* is inserted,
- `ON MODIFICATION OF` *xsl-pattern*: if anything in the subtree is modified,
- `ON UPDATE OF` *xsl-pattern*: the value (text or attribute) of a node matching the *xsl-pattern* is modified,
- `ON INSERT INTO` *xsl-pattern*: if a node is inserted (directly) into a node matching the *xsl-pattern*,
- `ON INSERT [IMMEDIATELY] BEFORE|AFTER` *xsl-pattern*: if a node is inserted (immediately) before or after a node matching the *xsl-pattern*.

All triggers should make relevant values accessible, e.g., `OLD AS ...` and `NEW AS ...` (like in SQL), both referencing the complete node to which the event happened, additionally `INSERTED AS`, `DELETED AS` referencing the inserted or deleted node.

Similar to the SQL `STATEMENT` and `ROW` triggers, the granularity has to be specified for each trigger; the following granularities are proposed here:

- `FOR EACH STATEMENT` (as in SQL),
- `FOR EACH NODE`: for each node in the *xsl-pattern*, the rule is triggered only at most once (cumulative, if the node is actually concerned by several matching events) per transaction,
- `FOR EACH MODIFICATION`: each individual modification (possibly for some nodes in the *xsl-pattern* more than one) triggers the rule.

For data-dependent information propagation, mainly `FOR EACH NODE` and `FOR EACH MODIFICATION` are adequate.

The implementation of such triggers in XML repositories is probably to be based on the *DOM Level 2/3 Events* [17].

RDF. RDF triples, describing properties/values of a resource are much more similar to SQL. In contrast to XML, there is no assignment of data with subtrees

(which makes it impossible to express “deep” modifications in a simple event; such things have then to be expressed in the condition part). A proposal can e.g. be found in [49]. The following proposal has been developed in [2]:

- `ON DELETE OF` *property* `[OF` *class*`]`,
- `ON INSERT OF` *property* `[OF` *class*`]`,
- `ON UPDATE OF` *property* `[OF` *class*`]`.

If a property is removed from/added to/updated of a resource of a given class, then the event is raised.
Additionally,

- `ON CREATE OF` *class* is raised if a new resource of a given class is created.

Probably, also metadata changes have to be detected:

- `ON NEW CLASS` is raised if a new class is introduced,
- `ON NEW PROPERTY [OF CLASS` *class*`]` is raised, if a new property (optionally: to a specified class) is introduced.

All triggers should make relevant values accessible, e.g., `OLD AS ...` and `NEW AS ...` (like in SQL), both referencing the original/new value of the property, `RESOURCE AS ...` and `PROPERTY AS ...` refer to the modified resource and the property (as URIs), respective.
Trigger granularity is `FOR EACH STATEMENT` or `FOR EACH TRIPLE`.

5.6 Local and Global ECA Rules

While “triggers” are restricted, programming-language concepts, general ECA rules provide an abstract concept using an own language. Especially in our setting, they are usually separated from the database. Thus, they do not react on “physical” events in the database, but on *logical* events (that nevertheless are actually raised by events in a database).

Local ECA Rules. Local ECA rules are more general than triggers. They still react on local events only, but they use an own event language that is based on a set of atomic events (that are not necessarily simple update operations) and that usually also allows for composite events. Their event detection mechanism is not necessarily located in the database. Detection of atomic logical events can be based on

- database triggers that generate events that are visible/detectable outside the database, or
- they have to poll the database regularly if such an event occurred.

Global ECA Rules. Global ECA rules have to be used if a composite event consists of subevents at different locations (or if the source of an event is not able to process local rules). When considering global rules in the Web and in the Semantic Web, the local ECA concept has to be extended stepwise:

– "distributed" variants of the above local ECA rules, with events that explicitly mention a database/node where the event is located (e.g., "change of *xpath-expr* at *url*"),
– rules that react on events in a set of known databases (e.g., "when a new researcher is added at one of the participants nodes" (which itself is a dynamic set)),
– high-level rules of an application, that are not based on schema knowledge of individual databases, often even not explicitly on a given database (e.g., "when a publication p becomes known that deals with ..."). Here, Semantic Web reasoning comes heavily into play even for detecting atomic events "somewhere in the Web". Such rules will probably be used in the "Travel Planning Scenario".

Requirements. The target of development and definition of languages for (ECA) rules, events, and actions in the Semantic Web should be a semantic approach, i.e., based on an (extendible) ontology for rules, events, and actions that also allows for reasoning about these concepts.

6 A Framework Proposal, Conclusions and Further Issues

Languages. For developing an ECA proposal for the Semantic Web, several languages with well-defined interfaces are needed. In [2], a preliminary framework for expressing ECA rules for the Semantic Web has been proposed.

ECA rules are marked up in the language that we will probably call ECA-ML (XML), or even formulated more abstractly in RDF, using an OWL ECA ontology. In general, the rules use sublanguages for describing *events* (metadata, including a contents part that contains the actual event), according to an *event ontology* (EventML), conditions (allowing to embed XQuery), and (trans)actions (embedding SOAP for service calls as atomic actions). A language for actual messages (XML, to be exchanged) is also needed.

An important principle here is to provide a *framework* that covers the *concepts* described above, not specific *languages* – there are multiple possible event, condition (query), and (trans)action languages. Thus, we propose a metamodel with a (basic) set of languages embedded in a modular concept of languages:

Rule Language. The rule language ECA-ML (namespace eca), provides rule elements with event, condition, and action subelements. These in turn contain subelements of event, condition, and action markup languages. The concrete language can be indicated as a language attribute (e.g., as a commonly known name, or as a URI where further information can be found).

Example 9 *Consider the* REWERSE *Personalized Portal scenario as described in [2]. The scenario consists of participants' nodes, working group nodes, and a central project node.*

The following rule propagates the change of a person's phone number from a participant's node to the information server of a working group. It reacts on a change of a phone number in the local database. If the person whose number changes, belongs to the working group (checked by an XPath query against the WG's database), the change is propagated to a remote server (by an explicit XQuery+Update statement against the WG's database):

```
<eca:rule>
  <variable name="WGUrl">http://...</variable>
  <eca:event>
    <evt:atomic>
      <change-of select="person//phone">
        <variable name="phone" select="."/>
        <variable name="person" select="$phone/ancestor::person"/>
      </change-of>
    </evt:atomic>
  </eca:event>
  <eca:condition language="XPath">
    $WGUrl//person[matches(name,$person/name)]
  </eca:condition>
  <eca:action language="XQuery+Updates">
    update $WGUrl
    set //person[matches(name,$person/name)]/phone := $phone
  </eca:action>
</eca:rule>
```

Event Language. The proposal contains a simple event language that allows to express terms in an event algebra. The basis are atomic events that can again be given in several languages. The generic approach proposes an XSL-style language for detecting changes in the local database (syntactic XML sugar to the trigger events in Section 5.5). The event language comprises constructs like <seq>, <disj>, <conj>, <forany> and <forall> with <variable> subelements, <cumulative> with appropriate switches, etc.

Condition Language. For condition languages, we propose to use existing languages like XPath, XQuery, RDQL, Xcerpt etc.

Action Language. The proposal contains a simple action language that allows to express composite actions. The basis are atomic actions that can again be given in several languages (e.g., XUpdate, XQuery+Update, XChange, or SOAP calls). The action language comprises constructs like <seq>, <conj>, <if test="..."> and <while test="..."> with appropriate switches, and <forall> with <variable> subelements, providing similar constructs as for CCS process specifications.

Implementation Issues. The modular design of the languages must be mirrored in a modular design of the architecture. For providing composability, the modules

must adhere to standardized interfaces. The overall architecture must provide addressing and coupling mechanisms for addressing modules and services that implement concrete languages over the Web.

Conclusion. Research in Evolution and Reactivity for the Semantic Web requires a profound knowledge of existing concepts, logics, and formal methods in the areas of (active and distributed) databases, software engineering, and Web technology such as semistructured data and communication mechanisms.

Further Issues. Due to the restricted space (and time), a lot of issues has not been discussed here: evolution at the level of RDF/OWL, evolution of rules of knowledge bases and behavioral rules, evolution in communities of peers, and super-peers and concepts from agent and multi-agent systems.

References

1. J. J. Alferes, J. Bailey, M. Berndtsson, F. Bry, J. Dietrich, A. Kozlenkov, W. May, P. L. Pătrânjan, A. Pinto, M. Schroeder, and G. Wagner. State-of-the-art on evolution and reactivity. Technical Report IST506779/Lisbon/I5-D1/D/PU/a1, REWERSE, September 2004.
2. J. J. Alferes, M. Berndtsson, F. Bry, M. Eckert, N. Henze, W. May, P. L. Pătrânjan, and M. Schroeder. Use-cases on evolution. Technical Report IST506779/Lisbon/I5-D2/D/PU/a1, REWERSE, September 2004.
3. James Bailey, Alexandra Poulovassilis, and Peter T. Wood. An event-condition-action language for XML. In *Int. WWW Conference*, 2002.
4. C. Baral, M. Gelfond, and Alessandro Provetti. Representing actions: Laws, observations and hypotheses. *Journal of Logic Programming*, 31(1–3):201–243, April–June 1997.
5. M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. In *8th Annual ACM Symp. on Principles of Programming Languages*, 1981.
6. J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 1(37):77–121, 1985.
7. Angela Bonifati, Daniele Braga, Alessandro Campi, and Stefano Ceri. Active XQuery. In *Intl. Conference on Data Engineering (ICDE)*, pages 403–418, San Jose, California, 2002.
8. A. J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133(2):205–265, 1994.
9. A. J. Bonner and M. Kifer. Concurrency and communication in transaction logic. In *ICDT'95: Advances in Logic-Based Languages*, 1995.
10. François Bry and Sebastian Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. In *Intl. Conf. on Logic Programming (ICLP)*, number 2401 in LNCS, pages 255–270, 2002.
11. François Bry, Paula Lavinia Pătrânjan, and Sebastian Schaffert. Xcerpt and XChange: Deductive Languages for Data Retrieval and Evolution on the Web. In *Proc. of Workshop on Semantic Web Services and Dynamic Networks, Ulm, Germany, (22nd – 24th September 2004)*. GI, 2004.

12. François Bry and Sebastian Schaffert. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proc. of Extreme Markup Languages 2004, Montreal, Quebec, Canada, (2nd – 6th August 2004)*, 2004.
13. S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proceedings of the 20th VLDB*, pages 606–617, 1994.
14. Jianjun Chen, David J. deWitt, Feng Tian, and Yuang Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pages 379–390, 2000.
15. Jan Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Transactions on Database Systems (TODS)*, 20(2):149–186, 1995.
16. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. of the IBM Workshop on Logics of Programs*, number 131 in Lecture Notes in Computer Science, 1981.
17. Document object model (DOM). `http://www.w3.org/DOM/`, 1998.
18. Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. *ACM Transactions on Computational Logic*, 5(2):206–263, 2004.
19. Thomas Eiter, Wolfgang Faber, Gerald Pfeifer, and Axel Polleres. Declarative planning and knowledge representation in an action language. In Ioannis Vlahavas and Dimitris Vrakas, editors, *Intelligent Techniques for Planning*. Idea Group, Inc., 2004.
20. E. A. Emerson and C.-L. Lei. Modalities for model checking: Branching time strikes back. In *12th Annual ACM Symp. on Principles of Programming Languages*, 1985.
21. eXist: an Open Source Native XML Database. `http://exist-db.org/`.
22. D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *ACM Symposium on Principles of Programming Languages*, pages 163–173, 1980.
23. Dov Gabbay. The declarative past, and imperative future: Executable temporal logic for interactive systems. In B. Banieqbal, B. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, number 398 in Lecture Notes in Computer Science, pages 409–448. Springer, 1989.
24. M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.
25. M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on Artificial Intelligence*, 2(3-4):193–210, 1998.
26. E. Giunchiglia, G. Kartha, and V. Lifschitz. Representing actions: Indeterminacy and ramifications. *Artificial Intelligence*, 95:409–443, 1997.
27. E. Giunchiglia, J. Lee, V. Lifschitz, N. Mc Cain, and H. Turner. Representing actions in logic programs and default theories: a situation calculus approach. *Journal of Logic Programming*, 31:245–298, 1997.
28. E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153:49–104, 2004.
29. E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *AAAI'98*, pages 623–630, 1998.
30. Y. Gurevich. Logic and the challenge of computer science. In *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.
31. Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

32. D. Harel. *First-Order Dynamic Logic.* Number 68 in Lecture Notes in Computer Science. Springer, 1979.
33. D. Harel. Dynamic Logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic, Volume II - Extensions of Classical Logic*, pages 497–604. Reidel Publishing Company, 1984.
34. D. Harel, D. Kozen, and R. Parikh. Process Logic: Expressiveness, decidability, completeness. *Journal of Computer and System Sciences*, 25(2):144–170, 1982.
35. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
36. David B. Kemp, Kotagiri Ramamohanarao, and Peter J. Stuckey. ELS Programs and the Efficient Evaluation of Non-Stratified Programs by Transformation to ELS. In Tok Wang Ling, Alberto O. Mendelzon, and Laurent Vieille, editors, *Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, number 1013 in Lecture Notes in Computer Science, pages 91–108, Singapore, 1995. Springer.
37. L. Lamport. 'sometimes' is sometimes 'not never'. In *7th Annual ACM Symp. on Principles of Programming Languages*, 1980.
38. Georg Lausen, Bertram Ludäscher, and Wolfgang May. On logical foundations of active databases. In Jan Chomicki and Gunter Saake, editors, *Logics for Databases and Information Systems*, chapter 12, pages 389–422. Kluwer Academic Publishers, 1998.
39. Patrick Lehti. Design and Implementation of a Data Manipulation Processor for an XML Query Language (diploma thesis), August 2001. Technische Universität Darmstadt.
40. H.J. Levesque, R. Reiter, Y. Lesprance, F. Lin, and R. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–83, 1997.
41. J. Lobo, R. Bhatia, and S.Naqvi. A policy description language. In *National Conference on Artificial Intelligence (AAAI)*, 1999.
42. Wolfgang May, José Júlio Alferes, and François Bry. Towards generic query, update, and event languages for the Semantic Web. In *Principles and Practice of Semantic Web Reasoning (PPSWR)*, number 3208 in Lecture Notes in Computer Science, pages 19–33. Springer, 2004.
43. John McCarthy. *Formalizing Common Sense.* Ablex, Norwood, 1990.
44. John McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4, 1969.
45. R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.
46. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
47. R. Milner. Operational and algebraic semantics of concurrent processes. In J. v. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 19, pages 1201–1242. Elsevier, 1990.
48. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 1(100):1–77, 1992.
49. George Papamarkos, Alexandra Poulovassilis, and Peter T. Wood. Event-condition-action rule languages for the semantic web. In *Workshop on Semantic Web and Databases (SWDB'03)*, 2003.
50. E. Pednault. Exploring the middle ground between STRIPS and the Situation Calculus. In *Proc. of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pages 324–332. Morgan Kaufmann Publishers Inc., 1989.
51. G. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

52. V. R. Pratt. Semantical considerations on Floyd-Hoare Logic. In *17.th IEEE Symp. on Foundations of Computer Science*, pages 109–121, 1976.
53. R. Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64(2):337–351, 1993.
54. E. Sandewall. *Features and Fluents: A Systematic Approach to the Representation of Knowledge about Dynamical Systems.* Oxford University Press, 1994.
55. Munindar P. Singh. Semantical considerations on workflows: An algebra for intertask dependencies. In *Intl. Workshop on Database Programming Languages*, electronic Workshops in Computing, Gubbio, Italy, 1995. Springer.
56. A. Prasad Sistla and Ouri Wolfson. Temporal Conditions and Integrity Constraints in Active Database Systems. In *Proceedings ACM SIGMOD International Conference on Management of Data (SIGMOD 1995)*, pages 269–280, 1995.
57. T. Son and C. Baral. Formalizing sensing actions - a transition function based approach. *Artificial Intelligence*, 125(1-2):19–91, 2001.
58. C. Stirling. Temporal logics for CCS. In *Linear Time, Branching Time and Partial Order in Logics and Models of Concurrency*, number 354 in Lecture Notes in Computer Science, pages 660–672. Springer, 1989.
59. Igor Tatarinov, Zachary G. Ives, Alon Halevy, and Daniel Weld. Updating XML. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pages 133–154, 2001.
60. Feng Tian, Berthold Reinwald, Hamid Pirahesh, Tobias Mayr, and Jussi Myllymaki. Implementing a scalable XML publish/subscribe system using relational database systems. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 2004.
61. J. van Benthem and J. Bergstra. Logic of transition systems. *Journal of Logic, Language, and Information*, 3:247–283, 1995.
62. World Wide Web Consortium, http://www.w3.org/TR/xpath. *XML Path Language (XPath)*, Nov 1999.
63. World Wide Web Consortium, http://www.w3.org/TR/xquery/. *XQuery: A Query Language for XML*, Feb 2001.
64. XML:DB Initiative, http://xmldb-org.sourceforge.net/. *XUpdate - XML Update Language*, September 2000.
65. Carlo Zaniolo. A unified semantics for active and deductive databases. In N. W. Paton and M. W. Williams, editors, *Proceedings of the 1st International Workshop on Rules in Database Systems*, Workshops in Computing, pages 271–287. Springer-Verlag, 1994. ISBN 3-540-19846-6.