

# Reactive Rules on the Web

Bruno Berstel<sup>1</sup>, Philippe Bonnard<sup>1</sup>, François Bry<sup>2</sup>, Michael Eckert<sup>2</sup>, and  
Paula-Lavinia Pătrânjan<sup>2</sup>

<sup>1</sup> ILOG

9, rue de Verdun, 94250 Gentilly, France

{berstel,bonnard}@ilog.fr — <http://www.ilog.com>

<sup>2</sup> University of Munich, Institute for Informatics

Oettingenstr. 67, 80538 München, Germany

{bry,eckert,patranjan}@pms.ifi.lmu.de — <http://www.pms.ifi.lmu.de>

**Abstract.** *Reactive rules* are used for programming rule-based, reactive systems, which have the ability to detect events and respond to them automatically in a timely manner. Such systems are needed on the Web for bridging the gap between the existing, passive Web, where data sources can only be accessed to obtain information, and the dynamic Web, where data sources are enriched with reactive behavior. This paper presents two possible approaches to programming rule-based, reactive systems. They are based on different kinds of reactive rules, namely *Event-Condition-Action rules* and *production rules*. Concrete reactive languages of both kinds are used to exemplify these programming paradigms. Finally the similarities and differences between these two paradigms are studied.

## 1 Introduction

*Reactivity on the Web*, the ability to detect events and respond to them automatically in a timely manner, is needed for bridging the gap between the existing, passive Web, where data sources can only be accessed to obtain information, and the dynamic Web, where data sources are enriched with reactive behavior. Reactivity is a broad notion that spans Web applications such as e-commerce platforms that react to user input (e.g., putting an item into the shopping basket), Web Services that react to notifications or service requests (e.g., SOAP messages), and distributed Web information systems that react to updates in other systems elsewhere on the Web (e.g., update propagation among biological Web databases).

The issue of enriching (relational or object-oriented) database systems with reactive features has been largely discussed in the literature and software solutions (called active database systems) have been employed for some years by now. Differences between (generally centralized) active databases and the Web, where a central clock, a central management are missing and new data formats (such as XML and RDF) are used, give reasons for developing new approaches. Moreover, approaches that cope with existing and upcoming Semantic Web technologies (by gradually evolving together with these technologies) are more likely

to leverage the Semantic Web endeavor. Along this line, of crucial importance for the Web is the usability of (Semantic) Web technologies that should be approachable also by users with little programming experience.

The rule-based approach to realizing reactivity on the Web is discussed in this lecture as an example of an easy to use (Semantic) Web technology. Compared with general purpose programming languages and frameworks, rule-based programming brings in declarativity, fine-grain modularity, and higher abstraction. Moreover, modern rule-based frameworks add natural-language-like syntax and support for the life cycle of rules. All these features make it easier to write, understand, and maintain rule-based applications, including for non-technical users.

The *Event-Condition-Action (ECA) rules* and *production rules* fall into the category of *reactive rules*, which are used for programming rule-based, reactive systems. In addition to the inherent benefits of rule-based programming mentioned above, the interest of reactive rules and rule-based technology for the Web is underlined by the current activity within W3C working groups on these subjects.<sup>3</sup>

Due to the emphasis they put on events, ECA rules have traditionally been used in reactive systems such as telecommunication network management. As such, they are well-suited for the reactive, event-based aspect of (distributed) Web applications. Production rules originate from non-monotonous expert systems, where they were used to encode the behavior of a system based on domain-specific knowledge. This makes them relevant to address the stateful, expertise-based aspect of (higher-end) Web applications.

ECA rules have the structure *ON Event IF Condition DO Action* and specify to execute the *Action* automatically when the *Event* happens, provided the *Condition* holds. Production rules are of the form *WHEN Condition DO Action* and specify to execute the *Action* if an update to the (local) data base makes the *Condition* true. This shows that the similarities in the structure of these two kinds of rules come with similarities, but also some differences, in the semantics of the two rule paradigms.

Since most Web applications have both an event-based and an expertise-based aspect, and because the two paradigms are semantically close to each other, Web applications can choose one paradigm or the other, depending on where they put the emphasis. They can also leverage the advantages of both, by choosing to implement a part of their logic using ECA rules, and another part using production rules.

This paper provides an introduction to programming Web systems with reactive rules, by discussing concrete reactive languages of both kinds, thus trying to reveal differences and similarities between the two paradigms. To illustrate the two approaches to realizing reactive behavior, the ECA rules language XChange and the ILOG Rule Language (IRL) have been chosen. XChange is an ongoing

---

<sup>3</sup> The W3C Rule Interchange Format Working Group is chartered to develop a format for rules that should enable rules to be translated between different rule systems, <http://www.w3.org/2005/rules/wg.html>

research project at the University of Munich and part of the work in the Network of Excellence REWERSE<sup>4</sup> (Reasoning on the Web with Rules and Semantics), which is a research project mainly funded by the European Commission. IRL is a production rule language marketed by ILOG<sup>5</sup> as part of their production rules system ILOG JRules.

## 2 Reactive Behavior on the Web: Application Examples

The Web has traditionally been perceived as a distributed repository of hypermedia documents and data sources with clients (in general browsers) that retrieve documents and data, and servers that store them. Although reflecting a widespread use of the Web, this perception is not accurate.

With the emergence of Web applications, Web Services, and Web 2.0, the Web has become much more dynamic. Such Web nodes (applications, sites, services, agents, etc.) constantly react to events bringing new information or making existing information outdated and change the content of data sources. Programming such reactive behavior entails (1) detecting situations that require a reaction and (2) responding with an appropriate state-changing action [?].

We present in this section several application example of such reactive behavior on the Web.

### 2.1 Distributed Information Portal

Many data sources on the Web are evolving in the sense that they change their content over time in reaction to events bringing new information or making existing information outdated. Often, such changes must be mirrored in data on other Web nodes – updates need to be propagated. For Web applications, such as distributed information portals, where data is distributed over the Web and part of it is replicated, update propagation is a prerequisite for keeping data consistent.

As a concrete application example, consider the setting of several distributed Web sites of a fictitious scientific community of historians called the Eighteenth Century Studies Society (ECSS). ECSS is subdivided into participating universities, thematic working groups, and project management. Universities, working groups, and project management have each their own Web site, which is maintained and administered locally. The different Web sites are autonomous, but cooperate to evolve together and mirror relevant changes from other Web sites.

The ECSS Web sites maintain (XML or RDF) data about members, publications, meetings, library books, and newsletters. Data is often shared, for example a member's personal data is present at his home university, at the management node, and in the working groups he participates in. Such shared data needs to be kept consistent among different nodes. This can be realized by communicating

---

<sup>4</sup> <http://rewerse.net>

<sup>5</sup> <http://ilog.com>

changes as events between the different nodes using reactive rules. Events that occur in this community include changes in the personal data of members, keeping track of the inventory of the community-owned library, or simply announcing information from email newsletters to interested working groups. These events require reactions such as updates, deletion, alteration, or propagation of data, which can also be implemented using reactive rules.

Full member management of the ECSS community, a community-owned and distributed virtual library (e.g., lending books, monitions, reservations), meeting organization (e.g., scheduling panel moderators), and newsletter distribution are desirable features of such a Web-based information portal. And all these can be elegantly implemented by means of reactive rules.

## 2.2 E-Shopping Web Site

**Shopping cart example** This example shows how a simple reactive rule set calculates the shopping discount of a customer. The business rules describing the discount allocation policy are listed hereafter:

1. If the total amount of the customer's shopping is higher than 100, then perform a discount of 10%.
2. If it is the first shopping of the customer, then perform a discount of 5%.
3. If the client has a gold status and buys more than 5 discounted items, then perform an additional discount of 2%.
4. Rule 1 and 2 must not be applied for the same customer, the first rule has the priority against the second. The third rule is applied only if rule 1 or rule 2 have been applied.

Those policies might be taken into account by a reactive rule service (Web service, procedural application, etc.). This service receives the customer and his shopping cart information as input. The discount calculation is then processed following the previous rules and returns the discount value to the service caller.

**Credit analysis example** This second example shows how a simple reactive ruleset defines a loan acceptance service. It determines whether a loan is accepted, depending on the client's history and the loan request duration. A client's score is calculated according to the following business policy. If the client's score is high enough, the loan is accepted and its rate is calculated.

1. If the loan duration is lower than five years then set the loan rate to 4.0% and add 5 to the score, else set it to 6.0%.
2. If the client has filed a bankruptcy, subtract 5 to the score.
3. If the client's salary is between 20000 and 40000, add 10 to the score.
4. If the client's salary is greater than 40000, add 15 to the score.
5. If the score is upper than 15, then the loan is accepted.

Those policies are usually implemented by a rule service (Web service, application). This service receives the loan request as input information, applies the rule on them in order to check the acceptance, and finally returns to the caller the loan characteristics.

### 3 Event-Condition-Action Rules

#### 3.1 General Ideas

Many Web-based systems need to have the capability to *update data* found at (local or remote) Web resources, to *exchange information* about events (such as executed updates), and to *detect and react* not only to simple events but also to situations represented by a temporal combinations of events. The issue of updating data plays an important role, for example, in e-commerce applications receiving and processing buying or reservation orders. The issues of notifying, detecting, and reacting upon events of interest begin to play an increasingly important role within business strategy on the Web and event-driven applications are being more widely deployed.

Different approaches can be followed for implementing Web applications having the capabilities touched on above. Section 1 has discussed the advantages of a rule-based approach for realizing reactive applications compared to general purpose programming languages and frameworks. Event-Condition-Action rules are high-level, elegant means to implement reactive Web applications whose architecture imply more than one Web components/nodes and their communication is based on exchanging events.

For communicating events on the Web two strategies are possible: the *push* strategy, i.e. a Web node informs (possibly) interested Web nodes about events, and the *pull* strategy, i.e. interested Web nodes query periodically (poll) persistent data found at other Web nodes in order to determine changes. Both strategies are useful. A push strategy has several advantages over a strategy of periodical polling: it allows faster reaction, avoids unnecessary network traffic, and saves local resources.

#### 3.2 ECA-Language Design Issues

**Rules** In the introduction, it has already been mentioned that ECA rules have the general form *ON Event IF Condition DO Action*. Before going into depth on events, conditions, and actions, we examine the notion of an ECA rule as a whole.

*Rule execution semantics* The general idea for interpreting a single ECA rule is to execute the *Action* automatically when the *Event* happens, provided the *Condition* holds. However, things become more complex when we consider not just a single rule but a set of rules (also called a rule base).

Consider the following example of two (informally specified) rules:

```
ON item out of stock
DO set item's status to 'not available'
```

```
ON item out of stock
IF item is in stock at one of the shop's suppliers
DO reorder item and set status to 'reordered'
```

These two rules are in a conflict when we try to execute both in response to an out of stock event for an item that is in stock at one of its suppliers. A language's rule execution semantics, determine what happens in such a situation. Possible rule execution semantics include (see also [?,?]):

- Selecting *one single* rule (or rather rule instance) from the so-called conflict set, the set of executable rules, for execution. This requires a selection principle in the language such as numeric priorities which are assigned to rules, a priority relation between rules, the textual order of rules in their definition, or the temporal order in which rules have been added to the rule base. (The last two are also often used as a “tie-breaker” when two rules have the same priority.) Some languages are simply non-deterministic, i.e., select a rule randomly or by an unspecified principle. The principle by which rules are chosen is often called the conflict resolution strategy.
- Executing *all* rules (or rule instances) from the conflict set sequentially in some order, which is determined similar to the selection above. When during the execution another event is generated by a rule, one can either suspend the execution of the other rules in the current conflict set to (recursively) execute any rules triggered by that event or first execute the complete conflict set and then (iteratively) turn to any rules triggered by any events generated in the meantime.
- Simply rejecting the execution of any rule (instance), possibly reporting an exception.

Usually, it is possible to avoid such conflicts by writing the rules differently. In the above example, the first rule could be modified to include a condition “item is *not* in stock at supplier.” Unfortunately, when rule sets grow larger, this can lead to quite lengthy conjunctions of conditions. If, in the above example, we also want to consider the case that an item is in stock at a different branch, we might have to add the negation of this condition to all other rules.

This short discussion has only scratched the surface of execution semantics for ECA rules. While they have been studied quite extensively in the area of Active Database Management Systems [?,?], i.e., in the context of typically closed and centralized systems, rule execution semantics have not been explored very much for the Web as an open and distributed system.

*Flow on information in a rule* ECA Rules exhibit a flow of information between their three parts. In the earlier example, the “out of stock” event part has to provide some identifier for the item. The condition part of the second rule makes use of this identifier in determining whether there is a supplier for the item, possibly providing the supplier's name to the action part.

A common way to provide for such a flow of information in an ECA language is to use variables, which are bound and exchanged between the different parts. This requires that the “sub-languages” in which the different parts are written share the same notion of a variable binding or at least that there is some conversion mechanism if different notions are used.

*Variants of ECA rules* In some cases, especially when reasonably complex decisions are involved, the same piece of knowledge must be distributed over several rules. We've already seen an example of this with the two rules processing out of stock events. To support a better modeling of such cases, some languages offer extended forms of ECA rules such as so-called ECAA rules [?]. ECAA rules have the form *ON Event IF Condition DO Action ELSE Alternative-Action*, specifying to execute *Alternative-Action* when the *Event* happens but the *Condition* does not hold. The example rules can be thus merged into one:

```
ON   item out of stock
IF   item is in stock at one of the shop's suppliers
DO   reorder item and set status to 'reordered'
ELSE set item's status to 'not available'
```

Note that any ECAA rule can generally be rewritten as two ECA rules, one with the original condition and one with the negated condition. A further variant are  $EC^nA^n$  rules, which specify a number of condition-action pairs; typically only the first action whose condition holds is executed.

*Rule base modifications* Some ECA rule-based systems allow to modify the rule base at run-time without restarting the system. The modifications are adding rules, by either registering a completely new rule or enabling a previously deactivated rule, removing rules, by either unregistering or disabling an existing rule, as well as replacing a rule. Often it is necessary to apply a number of modifications in an atomic fashion, i.e., all changes come into effect at the same time.

**Events** Event drive the execution of ECA rule programs, which makes the event part an important determinant for expressivity and ease-of-use of an ECA language or system.

*The notion of an event* It is hard to give a clear definition of what is an event. Often, an event is defined as an observable and relevant state change in a system; however with this definition, what is observable depends on the boundary and abstraction level of the system (which is particularly hard to grasp in an open system such as the Web), and what is relevant depends on the considered application. For processing purposes, an event is usually given a *representation* as an object or a message, and for the purpose of ECA languages this gives a practical definition of what is an event.

Examples of events one might want to react to include:

- Updates of local or remote data.
- Messages (or notifications) coming from some external source such as a human user (e.g., by filling out a Web form), another program (e.g., a request or reply from a Web service), or sensors (e.g., RFIDs).
- System events such as reports of the system status (e.g., CPU load) or of exceptions and errors (e.g., broken network connections, hardware failures).

- Timer events such as an alarm at a particular date and time or a periodic alarm (e.g., every day at 7am). A particular form of timer events are dynamic events (a term coined in [?]), where the time is specified not directly in the event part of a rule, but given by some entry in a database.
- Composite events, that is a combinations of events occurring over time. In contrast to so-called atomic or primitive events, which include the previous examples, composite events are usually not represented by a single object or message in the stream of events. Rather, they are just a collection of atomic events that satisfy some pre-defined pattern. The pattern is often called a composite event query, and accordingly composite events can be seen as answers to such composite event queries. Composite events will be discussed in more detail later.

Events, or rather their representations, usually contain information detailing the circumstances of the event such as:

- Who has generated the event (*generator*, sender)?
- When has the event happened (*occurrence time*, sending time) and when was it detected (*detection time*, receiving time)? Note that these times are often not the same due to delays in transmission and often given according to different clocks that are not (and cannot be) perfectly synchronized.
- Which kind of event has happened (*event type*, class)? Event types are ultimately application-dependent and of varying abstraction levels; examples could be an insertion of an element into an XML document (quite low-level in the data layer) or a customer putting an item into the shopping cart (higher-level in the application layer). Event-driven systems often employ a type system for events (e.g., a class hierarchy), but the example of XChange shows that this is not a necessity. (Note though that XChange does not preclude typing event messages with XML schema or a similar mechanism.)
- What data has been affected by the event (*event data*)? In the above example of an insertion, the event could include information about where the insertion has taken place (document URI, position in the document) and what (XML fragment) has been inserted. In the example of putting an item into the shopping cart, the event could include information about that item (product name, quantity) and the customer (name or another identifier).
- Why has the event happened, i.e., which previous events are responsible for making the current event happen (*causality information*)?

*The event part of a rule* The event part of an ECA rule has a two-fold purpose: it determines when to react, i.e., specifies a class (or set) of events that trigger the rule, and extracts data from the event (usually in the form of variable bindings) that can then be used in the condition and action part. Accordingly, the event part of a rule is in essence a query against (the stream of) incoming events. In contrast to answers to traditional database queries, however, answers to event queries are associated with an occurrence time, mirroring the temporal nature of events.

*Composite events and composite event queries* Often, a situation that requires a reaction cannot be detected from a single atomic event. Such situations are called *composite events* (as opposed to single atomic events), and they are especially important on the Web: in a carefully developed application, atomic events might suffice as designers have the freedom to choose events according to their goal. On the Web, however, many different applications are integrated and have to cooperate. Situations which have not been considered in an application's design must then be inferred from several atomic events.

There are at least the following four complementary dimensions that need to be considered for an event query language:

- Data extraction: As mentioned above, event carry data that is relevant to whether and how to react. The data must be provided (typically as bindings for variables) to the condition and action part of an ECA rule.
- Event composition: To support composite events, event queries must support composition constructs such as the conjunction, disjunction, and negation of events (or more precisely of event queries).
- Temporal conditions: Time plays an important role in many reactive Web applications. Event queries must be able to express temporal conditions such as “events  $A$  and  $B$  happen within 1 hour and  $A$  happens before  $B$ .”
- Event accumulation: Event queries must be able to accumulate events of the same type to aggregate data or detect repetitions. For example, a stock market application might require notification if “the average over the last 5 reported stock prices raises by 5%,” or a service level agreement might require a reaction when “3 server outages have been reported within 1 hour.”

The most prevalent style for event query languages uses composition operators such as conjunction and sequence to combine primitive event queries into composite event queries. We will see an example of this in Section 3.3 on XChange. This approach is not without problems, though: for the sequence operator alone, four different interpretations are conceivable as suggested in [?]. We will therefore also look at an alternative approach called XChange<sup>EQ</sup> in Section 3.3.

**Conditions** The condition part of an ECA rule expresses usually a query to persistent data sources. As with event queries, condition queries have the two-fold purpose of determining whether the rule fires (i.e., the action is executed) and extracting data in the form of variable bindings that is then used in the reaction. Querying XML, RDF, and other Web data is well-studied and a multitude of query languages have been devised [?]. Criteria to be considered for the Web query language used to express the condition part include:

- What is the query language's notion of answers (variable bindings, newly constructed data)?
- How are answers delivered, can they be used to “parameterize” further queries or the action? Can, for example, a variable bound in an event query

be a parameter in a condition query, i.e., the value delivered by the event query be accessed and used in the condition query?

- What evaluation methods for queries are possible (backward chaining, forward chaining)?
- Which data models are supported (XML, RDF, OWL)? Is it possible to access data in different data models within one query?
- How does the query language deal with object identity?
- Which reasoning or deductive capabilities does the query language provide (views, deductive rules, etc.)?

The choice of a query language has significant influence on the design of a reactive language and should thus be made carefully. While the primary purpose of the query language is to query persistent data (in the condition part), event messages often come in the same formats as persistent data. Accordingly, the query language is often also be used to query data in atomic events in the event part of ECA rules.

**Actions** While the event and condition part of an ECA rule only detect that a system has (entered) a certain state without affecting it, the action part intends to modify the current state and yield a new state. Typical actions are:

- Updating persistent data on the Web. For example, the event that a customer puts an item into her shopping basket requires recomputing the total price.
- Raising new events to communicate with other agents on the Web (Web sites, Web services, etc.). Usually the new event message is sent as a notification in an *asynchronous* manner and execution of the current rule or other rules proceeds immediately without waiting for an answer. For example, upon a checkout event, a new event containing a list with the bought items and the customer’s address is sent to the warehouse to initiate delivery.
- Procedure calls to some host environment or a Web service. In contrast to raising new events, a procedure call is usually *synchronous* and the rule has to wait for the call to complete before execution is continued.
- Modifications to the rule base. This includes the possibility to enable and disable rules<sup>6</sup> or to register (add) new rules and unregister (delete) existing rules. Such modifications of the rule base are not without problems, however, as self-modifying programs are generally conceived to be hard to analyze and understand.

*Updates* Updates modify the contents of Web resources by inserting, deleting, or replacing data items. Depending on the data format of the Web resources such data items are XML fragments, RDF triples, or OWL facts.

---

<sup>6</sup> Note, however, that alternatively to a specific enable/disable action, this can also be modeled by adding a condition on some object (a boolean value or similar), which signals whether the rule is “enabled” or “disabled,” to rules and modifying this object through and update

Updates can be specified conveniently in some update language. There is a strong connection between update languages and query languages, and most existing update languages are based on a query language. The query language can be used to locate items or positions in the resource where an update should be performed as well as to construct new data that will be inserted or used to replace old data.

*Combinations of actions* Being able to execute only one primitive action such as a single update or raising one new event is usually far too limiting. Actions that have to be taken can be quite complex and require several primitive actions to be performed. The most common way to put together several primitive actions into a compound action is to perform them in a sequence. However other ways to form compound actions such as a specification of alternatives (if one action can fail) or a conditional execution are useful, too.

Usually we expect a compound action to be executed in a transactional manner, i.e., either the whole compound action takes effect or it has no effect at all. In a distributed setting such as the Web this requires that all participants involved in a compound action agree on a commit protocol such as the two-phase commit (2PC; see, e.g., [?]).

Solutions to realize compound actions based on specifying a compensating action for each action have not been investigated deeply in the framework of ECA rules on the Web. Such issues have been investigated for databases, e.g., Sagas [?] (and a myriad of follow-up work on advanced transaction models), as well as in Web Services, e.g., with the notion of a “Business Activity” in WS-Transaction [?]. However it should be noted that the primary aim of these proposals is increasing parallelism for long running transactions. They still require two-phase commits (in particular at the end of the transaction) to give transactional guarantees in a distributed setting.

### 3.3 XChange as an Example of ECA Rules Language

This section presents *XChange*, a high-level, ECA rules-based language for realizing reactivity on the Web. We first introduce the paradigms upon which the language XChange relies and then present and exemplify the core constructs of the language.

**Paradigms** Clear paradigms that a programming language follows provide a better language understanding and ease programming. Hence, explicitly stated paradigms are essential for Web languages, since these languages should be easy to understand and use also by practitioners with little programming experience.

*Event vs. Event Query.* As discussed in Section 3.2, one can conceive every kind of changes on the Web as events. For processing them, XChange represents each event as one XML document. Event queries are queries against the XML data representing events. Event query specifications differ considerably from event representations, e.g. event queries may contain variables for event

data items. Most proposals dealing with reactivity do not significantly differentiate between event and event query. Overloading the notion of event precludes a clear language semantics and thus, makes the implementation of the language and its usage much more difficult. Event queries in XChange serve a double purpose: detecting events of interest and temporal combinations of them, and selecting data items from events' representation.

*Volatile vs. Persistent Data.* The development of the XChange language – its design and its implementation – reflects the view over the Web data that differentiates between *volatile data* (event data communicated on the Web between XChange programs) and *persistent data* (data of Web resources such as XML or HTML documents). Volatile data *cannot* be updated but persistent data can. To inform about, correct, complete, or invalidate former volatile data, new messages containing information about events that have occurred are communicated between Web nodes.

*Pattern-Based Approach.* XChange is a *pattern-based language*: Event queries describe *patterns* for events requiring a reaction. Web queries describe patterns for persistent Web data. Action specifications build also upon pattern specifications, as we will see later. Patterns are templates that closely resemble the structure of the data to be queried, constructed, or modified, thus being very intuitive and also straight forward to visualize [?].

*Strategy for Event Communication.* Possible communication strategies (i.e. pull and push) have been touched on in Section 3.1. The pull strategy is supported by languages for Web queries (e.g. XQuery [?] or Xcerpt [?]). XChange uses the *push* strategy for communicating events.

*Processing of Events.* Event queries are evaluated locally at each Web node. Each such Web node has its own local *event manager* for processing incoming events and evaluating event queries against the incoming event stream (volatile data). For efficiency reasons, an incremental evaluation is used for detecting composite events.

*Bounded Event Lifespan.* Event queries are such that no data on any event has to be kept forever in memory, i.e. the event lifespan should be bounded. Hence, design enforces that volatile data remains volatile. If for some applications it is necessary to make part of volatile data persistent, then the applications should turn events into persistent Web data by explicitly saving events.

**Rules** An XChange program is located at one Web node and consists of one or more ECA rules of the form *Event query* – *Web query* – *Action*. Events are communicated between XChange programs by ECA rules that raise and send them as event messages. Every incoming event (i.e., event message) is queried using the *event query* (introduced by keyword ON). If an answer is found and the *Web query* (introduced by keyword FROM) has also an answer, then the specified action (introduced by keyword DO) is executed.

Rule parts communicate through variable substitutions. Substitutions obtained by evaluating the event query can be used in the Web query and the

action part, those obtained by evaluating the Web query can be used in the action part.

The following example rule shows the structure and the information passing mechanism of an XChange ECA rule. Concrete examples of event queries, Web queries, and actions are given in the next sections.

```
ON <new discounts for books of type T applied by supplier S>
FROM <stock of books of type T low>
DO <send new order of books of type T to S >
```

The next sections introduce the Web query, event query, and action part of an XChange ECA rule. We start with the Web queries since XChange event queries and updates build upon and extend the Web queries.

**Web Queries** XChange embeds the Web query language Xcerpt [?,?] for expressing the *Web query* part of ECA rules and for specifying deductive rules in XChange programs. Using Xcerpt one can query and reason with tree- or graph-structured data such as XML or RDF documents. A deductive rule has the following form in Xcerpt<sup>7</sup>:

```
CONSTRUCT construct-term
FROM query-term
END
```

Such deductive rules allow for constructing views over (possibly heterogeneous) Web resources that can be further queried in the *Web query* part of XChange ECA rules.

Xcerpt is a *pattern-based language*: it uses query patterns, called *query terms*, for querying Web data and construction patterns, called *construct terms*, for re-assembling data selected by queries into new data items. For conciseness, Xcerpt represents data, query terms, and construct terms in a term-like syntax; the same approach is also taken in XChange. For example, for representing XML documents as terms, element names become term labels and child elements are represented as subterms surrounded by curly braces or square brackets (in case of ordered child elements).

Both partial (i.e. incomplete) or total (i.e. complete) query patterns can be specified. A query term  $t$  using a partial specification denoted by double brackets or braces) for its subterms matches with all such terms that (1) contain matching subterms for all subterms of  $t$  and that (2) might contain further subterms without corresponding subterms in  $t$ . In contrast, a query term  $t$  using a total specification (denoted by single square brackets [ ] or curly braces { }) does

<sup>7</sup> XChange integrates the Web query language Xcerpt – XChange constructs are based on and extend Xcerpt constructs and the prototypical implementation of XChange uses a prototypical implementation of Xcerpt. The keyword **FROM** has been used instead of **IF** for introducing the condition part of XChange ECA rules for achieving language uniformity without changing Xcerpt’s language design and implementation.

not match with terms that contain additional subterms without corresponding subterms in  $t$ .

Query terms contain *variables* for selecting subterms of data items that are bound to variables. In Xcerpt and XChange, variables are placeholders for data, very much like logic programming variables are. Variables are preceded by the keyword `var`. Variable restrictions can be also specified, by using the construct `->` (read *as*), which restricts the bindings of the variables to those terms that are matched by the restriction pattern (given on the right hand side of `->`). The following Xcerpt query term queries the list of suppliers at `http://suppliers.com` to determine the names and URIs of companies supplying books. This information is used e.g. when some books are out of stock and need to be reordered.

```
in { resource {"http://suppliers.com/list.xml", XML},
    desc supplier {{
      items {{ desc type { "Book" } }},
      contact {{
        name { var N },
        URI { var U }
      }}
    }}
}
```

Xcerpt query terms may be augmented by additional constructs like *subterm negation* (keyword `without`), *optional subterm specification* (keyword `optional`), and *descendant* (keyword `desc`) [?]. Query terms are “matched” with data or construct terms by a non-standard unification method called *simulation unification* dealing with partial and unordered query specifications. More detailed discussions on simulation unification can be found in [?,?]. In the above given example, the variable substitutions  $N \mapsto \text{"Springer"}$  and  $U \mapsto \text{"www.springer.de"}$  could be obtained as result of simulation unifying the query term with the given XML document.

**Event Queries** *Event messages* denote XChange event representations and communicate events between (same or different) Web nodes. An XChange *event message* is an XML document with a root element labelled `event` and the five parameters (represented as child elements as they may contain complex content): `raising-time` (i.e. the time of the event manager of the Web node raising the event), `reception-time` (i.e. the time at which a node receives the event), `sender` (i.e. the URI of the Web node where the event has been raised), `recipient` (i.e. the URI of the Web node where the event has been received), and `id` (i.e. a unique identifier given at the recipient Web node).

Each XChange-aware Web node monitors such incoming event messages to check if they match an event query of one of its XChange ECA rules. Differences between volatile and persistent data make Web query languages not sufficient as candidates for querying event data: Many situations need for their detection not just one event to occur, but more than one event to occur. The temporal

order of these (component) events and the specified temporal restrictions on their occurrence time need also to be taken into account in detecting situations. Mirroring these practical requirements, XChange offers not only *atomic event queries* but also *composite event queries*.

*Atomic Event Queries* Atomic event queries detect occurrences of single, atomic events. They are query patterns for the XML representation of events and may be accompanied by an absolute time restriction, which are used to restrict the events that are considered relevant for an event query to those that have occurred in a specified (finite) time interval. Such a time interval may be given by fixed start and end time points (keyword **in**) or just by an end time point (keyword **before**), in which case the interval starts with the time point of event query definition.

The following XChange atomic event query detects announcements of discounts applied by a supplier. The information about the supplier (sender URI) and the discount for a type of items are to be bound to the variables **S**, **D**, and **T**, respectively.

```
xchange:event {{
  xchange:sender { var S },
  discount {{
    items {{
      type { var T },
      discount { var D }
    }}
  }}
}}
```

*Composite Event Queries using Composition Operators* The need for detecting not only atomic events but also composite events has been motivated in Section 3.2. XChange offers *composite event queries* for specifying and detecting composite events of interest.

A composite event query consists of (1) a connection of (atomic or composite) event queries with event composition operators and (2) an optional temporal range limiting the time interval in which events are relevant to the composite event query. Composition operators are denoted with keywords such as **and** (both events have to happen), **andthen** (the events have to happen in sequence), **or** (either event can happen), **without** (non-occurrence of the event in a given time frame). Limiting temporal ranges can be specified with keywords such as **before** (all events have to happen before a certain time point), **in** (all events have to happen in an absolute time interval), **within** (all events have to happen within a given length of time). For a more in-depth discussion of XChange composite event queries see [?,?,?].

Composite events (detected using composite event queries) do not have time stamps, as atomic events do. Instead, a composite event inherits from its components a start time (i.e. the reception time of the first received constituent event

that is part of the composite event) and an end time (i.e. the reception time of the last received constituent event that is part of the composite event). That is, in XChange composite events have a *duration* (a length of time).

The following composite event query evaluates successfully if no acknowledgment for the order `s-rw2007-0023` is received between the 1st and 15th of October 2007:

```
without {
  xchange:event {{
    acknowledgement {{
      order {{ id { "s-rw2007-0023" } }}
    }}
  }}
} during [ 2007-10-01T10:00:00 .. 2007-10-15T14:00:00 ]
```

*Composite Event Queries using XChange<sup>EQ</sup>* Querying composite events based on composition operators (as presented above) has been well-investigated in active databases systems and works well with small queries. However, queries involving a larger number of events can sometimes become difficult to express and to understand.

Consider an example where we want for events  $a$ ,  $b$ ,  $c$  and  $d$  to happen and have the constraints that  $a$  happens before  $b$ ,  $a$  also happens before  $c$ , and  $c$  before  $d$ . Note that the query cannot be expressed as `and{ andthen[a, b], andthen[a, c], andthen[c, d] }`, since this query would allow different instances of  $a$  and  $c$  events to be used. A correct way to express the query would be: `andthen[a, and{b, andthen[c, d] }]`. If we now only add an additional constraint that  $b$  happens before  $d$ , the new query bears only little resemblance to the old: `andthen[a, and{b, c}, d]`. In fact, even though we *added* a constraint in our specification, the query has one operator *less*.

Composition operators mix the event querying dimensions explained in Section 3.2 (in the case of `andthen` event composition and temporal relationships are mixed). It can be argued that this leads to the exemplified difficulties in expressing and understanding queries and also to a certain incompleteness in the expressivity of such event query languages.

An alternative to using composition operators in XChange is investigated with the high-level event query language XChange<sup>EQ</sup>. In XChange<sup>EQ</sup>, the four orthogonal event querying dimensions are treated separately. The above example can be expressed as: `and{event i: a, event j: b, event k: c, event l: d} where {i before j, i before k, k before l, j before l} .` (Keep in mind that  $a$ ,  $b$ ,  $c$ ,  $d$  are generally multi-line atomic event queries, so that the increase in length compared to the composition-based approach is insignificant and outweighed by better readability.)

XChange<sup>EQ</sup> also adds support for deductive rules on events, relative temporal event (e.g., “five days longer than event  $i$ ,” written `extend[i, 5 days]`), and enforces a clear separation between time specifications that are used as events (and waited for) or only as restrictions (conditions in the `where`-part).

The following example rule detects an *overdue* event when an order that has been received before October 15 has not been acknowledged within 5 days.

```

DETECT
  overdue { var I }
ON
  and {
    event i: order {{ id { var I } }},
    event j: extend[i, 5 days],
    while j: not acknowledgment{{ id{ var I } }}
  } where { i before datetime("2007-10-15:14:00") }
END

```

More detail on XChange<sup>EQ</sup> can be found in [?,?].

**Actions** XChange rules support the following primitive actions: executing simple updates to persistent Web data (such as the insertion of an XML element) and raising new events (i.e., sending a new event message to a remote Web node or oneself). To specify more complex actions, compound actions can be constructed as from the primitive actions.

*Updating Web Data* An XChange *update term* is a (possibly incomplete) pattern for the data to be updated, augmented with the desired update operations (i.e., an update term is an Xcerpt query term enriched with update specifications). An update term may contain different types of update operations: An *insertion operation* specifies an Xcerpt construct term that is to be inserted, a *deletion operation* specifies an Xcerpt query term for deleting all data terms matching it, and a *replace operation* specifies an Xcerpt query term to determine data terms to be modified and an Xcerpt construct term as their new value. The following XChange update term updates the *offer.xml* document upon arrival of new books:

```

in { resource {"http://myshop.de/offer.xml", XML},
  offer {{
    books {{
      items {{
        type { var T },
        insert new-arrival { var B }
      }}
    }}
  }}
}

```

*Raising New Events* Events to be raised are specified as (complete) patterns for the event messages, called *event terms*. An event term is simply an Xcerpt

construct term restricted to having a root labelled `event` and at least one sub-term `recipient` specifying the URI of the recipient. The following XChange event term is used to order 50 Reasoning Web 2007 books at Springer:

```
xchange:event {
  xchange:recipient {"http://www.springer.de"},
  order {
    id { "s-rw2007-0023" },
    book { "Reasoning Web -- Third International Summer School 2007,
          Tutorial Lectures" },
    count { "50" }
  },
  delivery-info {
    company { ... }, address{ ... }
  }
}
```

*Specifying Compound Actions* The primitive actions described by update terms and event terms can become powerful by combining them. XChange hence allows specifying complex actions as combinations of (primitive and compound) actions. Actions can be combined with disjunctions and conjunctions. Disjunctions specify alternatives, only one of the specified actions is to be performed successfully. (Note that actions such as updates can be unsuccessful, i.e., fail.) Conjunctions in turn specify that all actions need to be performed. The combinations are indicated by the keywords `or` and `and`, followed by a list of the actions enclosed in braces or brackets. The list of the actions can be ordered (indicated by square brackets, `[]`) or unordered (indicated by curly braces, `{}`). If the actions are ordered, their execution order is specified to be relevant. If the actions are unordered, their execution order is specified as irrelevant, thus giving more freedom for parallelization.

**Declarative and Operational Semantics** XChange combines an event language, a query language, and an update language into ECA-rules. Accordingly, the declarative and operational semantics are given separately for each rule part. The semantics of an XChange ECA-rule follows immediately from the semantics of its parts; the “glue” between the parts is given by the substitutions for the variables. The semantics of event queries is the most interesting aspect of XChange semantics and is discussed in [?, ?, ?]. Semantics of XChange<sup>EQ</sup> are provided as a model theory and fixpoint theory and discussed in [?]. The underlying ideas for the semantics of Web queries and updates can be found in [?] and their detailed description is given in [?] and [?], respectively.

**Current Status** XChange is an ongoing research project. The design, the core language constructs, and the semantics of XChange are completed. For revealing the strengths and limits of the language, a couple of use cases have been

developed: Travel organization as an application of Web-based reactive travel planning and support and e-Book store as a simple Semantic Web scenario are presented in [?]. XChange has also been used for determining the suitability of the ECA rules approach for business process modeling and in particular for implementing the EU-Rent case study [?,?].

A proof-of-concept implementation<sup>8</sup> exists, which follows a modular approach that mirrors the operational semantics. The XChange prototype has been implemented in Haskell, a functional programming language; choosing Haskell has been strongly motivated by an existing Xcerpt prototype implementation, which has been extended for implementing XChange. The XChange prototype has been employed for implementing the application scenario *Distributed Information Portal* described in Section 2; the developed demonstration of XChange is presented in [?,?]. Issues of efficiency of the implementation, esp. for event detection and update execution, have not been a priority in developing the prototype and are subject to future work.

There are a couple of further research issues that deserve attention within the XChange project, such as the automatic generation of XChange rules (e.g. based on the dependencies between Web resources' data) or the development of a visual counterpart of the textual language (along this line, the visual rendering of Xcerpt programs – visXcerpt [?] – is to be extended).

### 3.4 Implementation of ECA Systems

Implementation and architecture of ECA rules systems have been studied extensively in the area of Active Database Management Systems (see [?] for an overview). Unlike the Web, which is open, distributed, and decentralized, active databases are rather closed and centralized systems. It is therefore not clear how well their architectures would transfer to a Web context and there has not been much research on this issue. We therefore concentrate in this section mainly on the algorithms used in implementations of the event, condition, and action part, respectively, rather than overall architectural issues.

**Event Part – Atomic Events** The evaluation of atomic event queries has two main issues, mainly with regard to efficiency: the detection of updates in documents and databases (mainly XML, but also other Web data formats) that satisfy given (update) event queries and the evaluation of a potentially large number of event queries against events that are received from other Web nodes as messages.

Detection of relevant updates has been studied extensively in relational databases [?,?], often under the term “trigger processing.” The only work we are aware of where this issue has been studied from an XML perspective is Active XQuery [?], which will be described in Section 3.5.

<sup>8</sup> XChange Prototype, <http://reactiveweb.org/xchange/prototype.html>

The other issue is that when an event message (an XML document) is received, a potentially large number of atomic event queries (e.g., XPath expressions or Xcerpt/XChange query terms) have to be evaluated against this message. From an optimization perspective, this is the inverse of the classical database query optimization: instead of evaluating a single query against a relatively large amount of data, we have to evaluate a large amount of queries against relatively small data. Therefore, atomic event query evaluation requires multi-query optimization where queries (rather than data) are indexed and similarities between queries exploited. A number of approaches for multi-query optimization of XPath expressions have been devised, which are based on finite state machines [?,?]. However, the issue has been treated only in isolation and not as part of a full ECA rule engine.

**Event Part – Composite Events** For the evaluation of composite event queries, a data-driven approach is best-suited. Since it can work incrementally, it is preferable for efficiency reasons: work done in one evaluation step of an event query should not be redone in future evaluation steps. For example, the composite event query “events  $A$  and  $B$  happen” requires to check every incoming event if it is  $A$  or  $B$  and thus multiple evaluation steps. When event  $A$  is detected, we want to remember this for later when  $B$  is detected to signal the composite event. In contrast, a non-incremental, query-driven (backward-chaining-like) evaluation would have to check the entire history of events for an  $A$  when a  $B$  is detected. Popular data-driven approaches used in the past include finite automata [?,?,?] and event trees (or graphs) [?,?,?,?,?].

In the finite automata approach, states signify the “progress” made in detecting a composite event and state transitions are caused by incoming atomic events. This simple intuition is complicated though by the need to backtrack or reset the automata after the first event has been detected in order to detect further events. Further, when data is correlated between events, automata have to be extended to accommodate this, too.

The event tree approach is similar to the Rete algorithm, which will be described in detail in Section 4.4 in the context of production rules. The basic idea is to represent an event query as an operator tree where leaf nodes correspond to atomic event queries and inner nodes to composition operators such as conjunction or sequence. New events (or event data) flow bottom up in this tree and inner nodes have a storage to memorize previously detected events. When an inner node detects a composite event from the new and the memorized events, it “forwards” this composite event to its parent node. When event queries share subexpressions, this can be exploited by using a directed acyclic graph instead of a tree. So far, this is also the basic idea of Rete; however, certain operators such the sequence allow to disable evaluation of subtrees depending on the stored events: for example, to evaluate the sequence of events  $E_1$  followed by  $E_2$ , the subtree for  $E_2$  must only be evaluated after an  $E_1$  instance has been detected.

Another approach discussed in the literature are (special types of) Petri nets [?]. However this can be seen as a variant of the event tree approach, since for

each possible operator a separate Petri net is given and for a given event query the Petri nets for all its operators are then connected in essentially the same manner as the inner nodes in the event tree.

**Condition Part – Query Evaluation** For the evaluation of Web queries in the condition part, ECA rules systems usually rely on existing query evaluation engines. Accordingly, query processing takes only place whenever a rule is triggered by an event. This means that even though the condition part of an ECA rule can be considered a standing query (whose result could be precomputed whenever the underlying data changes), it is not treated this way but only posed as a spontaneous query whenever necessary. This kind of evaluation of a (single) Web query is a well-researched issue, see, e.g., [?].

We will see in Section 4.4 that evaluating the condition part only when an event triggers an ECA rule is in contrast to the continual evaluation of the condition part in production rule systems.<sup>9</sup> An advantage of treating conditions as spontaneous queries is that no restrictions are being posed on the accessed data sources, they can be any resources anywhere on the Web and event queries that “crawl” from Web resource to Web resource are conceivable. In contrast, precomputing query answers would usually be restricted to a local and closed set of resources.

**Action Part – Update Execution** For specifying the updates in the Action part of ECA rules, a update language is employed. Due to the absence of a standard update language for XML or RDF data, each ECA rule language uses its own update language and the supported updates are usually implemented in an ad-hoc fashion.

At least for the path-based update languages for XML, good chances exist to change this situation as the W3C works towards standardizing a update extension to XQuery. The W3C XQuery Update Facility<sup>10</sup> Working Draft, released in July 2006, presents the syntax and semantics of such an XQuery extension. The draft defines update primitives such as insertion or deletion of a node, modification of a node while preserving its identity, or creation of a updated node with a new identity. Variants of these primitives are also proposed, e.g. insert after or insert as last. The notion of *pending update list* is defined as an unordered collection of update primitives, which is the base for update execution. Guidelines for constructing the pending update list are also given.

The issue of *snapshot semantics* is currently discussed in the W3C XML Query Working Group for processing the specified updates. The following snapshot semantics is used for the UpdateX [?] language, an XQuery-based update language for XML: A first processing step determines the scope of the updates

<sup>9</sup> It would of course be conceivable to use the condition query evaluation techniques of production rules for the evaluation of conditions of ECA rules. However we are not aware of any systems doing this.

<sup>10</sup> XQuery Update Facility, <http://www.w3.org/TR/xqupdate/>

(i.e. for a FLWUpdate expression, the variables declared in the FOR and LET XQuery clauses are bound) and evaluate (not apply) each update primitive; the list of update primitives is thus formed. A checking step follows, where different kinds of constraints (e.g. given by a DTD) are performed. If their execution would not give invalid results, the updates in the constructed list are applied sequentially. Following these steps, the UpdateX has been implemented within the Galax<sup>11</sup> project.

An interesting implementation approach is followed in the Active XQuery language [?], which uses the update extensions to XQuery proposed in [?]. The main notions of SQL [?] triggers are used here but the execution model of SQL-3 is revised so as to cope with the hierarchical nature of XML data. Active XQuery update specifications may involve insertion or deletion of (whole) fragments of XML documents. These statements are called *bulk update statements* in this work. Problems may occur when executing such bulk updates directly: Consider the example of inserting a whole subtree  $S$  into an XML tree  $T$ . An ECA rule whose Event part waits for insertions of portions of  $S$  into  $T$  to fire would not detect the insertion without a mechanism supporting this. Thus, bulk update statements in Active XQuery are transformed (i.e. expanded) into equivalent collections of simple update operations. An algorithm for update expansion is outlined in [?]. The output of the algorithm is a list of simple update operations together with evaluation directives, which guide the language processor in firing all triggered ECA rules.

Alternative techniques for implementing the update operations are presented in [?] for the case when XML data is stored in a relational database (i.e. XML update statements are translated into SQL statements). This is the only work on updates for the Web that reports on implementation performance. Using three sets of test data (i.e. synthesized data with fixed structure, synthesized data with random structure, and real life data from the DBLP [?] bibliography database) experimental results were done in order to compare the techniques proposed for the core update operations (here, insert and delete).

### 3.5 An Overview of Existing ECA Languages and Systems

There are not great many ECA languages developed so far or under development at moment and most of them are results of research efforts done in the academia in the last couple of years.

The *General Semantic Web ECA Framework* [?,?,?] is a research endeavor that proposes a general framework<sup>12</sup> for reactive behavior on the Semantic Web. The generality here is given by the heterogeneity of the ECA rule components, which can be specified by using different event, condition, and action languages. Just the information flow between the rule components in form of variable substitutions constrains the languages of choice. The language used in writing an

<sup>11</sup> Galax, <http://www.cise.ufl.edu/research/mobility/>

<sup>12</sup> General Semantic Web ECA Framework, <http://www.dbis.informatik.uni-goettingen.de/eca/>

ECA component is given by means of the ECA-ML markup language for ECA rules offered by the framework; e.g. the URI of the languages is given as an attribute:

```
<eca:rule xmlns:eca="http://.../eca/2006/eca-ml">
  ...
</eca:rule>
```

ECA rule components are processed at Web nodes where a processor for the given language exists. For determining whom to forward the processing task, a Language and Service Registry is queried. A reference implementation for determining an appropriate Web processing node together with an event detection module based on a SNOOP-like event specifications have been completed. An action component given by a process algebra, the Calculus of Communicating Systems (CCS) [?], has been proposed and its implementation is underway. Also, an ontology of behavior is under development with the aim of using it as basis for reasoning and for easing the editing of ECA rules. As basis for it, the OWL-DL ontology<sup>13</sup> of the Resourceful Reactive Rules<sup>14</sup> (r3) project is considered. The goal of the r3 project is to develop a prototypical implementation of a Semantic Web reactive rule engine based on the ideas of the General Semantic Web ECA Framework.

Prova<sup>15</sup> is a combination of Java with Prolog-style rules. It employs ECA rules as means for distributed and agent programming. ECA rules react to incoming messages or pro-actively poll for state changes. By using Prova, composite events can be detected and different kinds of actions can be executed. Complex work-flows can be specified in the action part as all BPEL constructs are directly available in the language. Prova has been used in a number of academic projects and also as basis for a commercial product for information integration.

The ruleCore<sup>16</sup> system provides an engine for executing ECA rules and also a couple of GUI tools such as the ruleCore Monitor, which gives run-time status information on the engine. The ruleCore engine detects situations specified by means of composite events. One can detect for example sequences, conjunctions, disjunctions, and negation of events. The engine supports also the detection of events that happen within a given time interval. Several event sources and of different kinds can be connected to the ruleCore engine, which processes events represented as XML documents. The action part of ECA rules can contain a number of action items, which specify that scripts are to be executed or events are to be generated. ruleCore has been developed by Analog Software. It can be used in research projects and can also be licensed for commercial use.

The Reaction RuleML<sup>17</sup> effort of the RuleML Initiative<sup>18</sup> aims at a general language that should enable inter-operation between industrial products and

<sup>13</sup> r3 Ontology, <http://reverse.net/I5/r3/DOC/2005/index.html>

<sup>14</sup> Resourceful Reactive Rules, <http://reverse.net/I5/r3/>

<sup>15</sup> Prova, <http://www.prova.ws>

<sup>16</sup> ruleCore, <http://www.rulecore.com/index.html>

<sup>17</sup> Reaction RuleML, <http://ibis.in.tum.de/research/ReactionRuleML/>

<sup>18</sup> RuleML Initiative, <http://www.ruleml.org>

academic research results following different approaches to reactivity. The work on the ECA Logic Programming language (ECA-LP) and the ECA Rule Markup language (ECA-RuleML) as its XML serialization syntax has started during 2006. These languages are general enough to support ECA rules and their variant ECAP rules, but also production rules. They allow the specification of composite events to be detected and of different kinds of actions (notifications, updates and sequences of updates, etc.) to be executed.

An ECA rule language for XML data is proposed in [?] and adapted for RDF data as the RDF Triggering Language (RDF-TL) [?]. These languages have the capability to react only to single events and do not provide constructs for querying for complex combinations of events. As actions, simple insertions or deletions to XML or RDF data and sequences thereof are supported. At moment of writing these two research projects are not developed further.

ECA rules are discussed in the context of XSL [?] and Lorel [?] as means to realize active document management systems, i.e. XML repositories with reactive capabilities [?]. An ECA rule consists here of an event part and a condition-action part, which mixes the condition and the action specifications. Events considered here are just simple modifications of XML documents and there is no support for composite events. Conditions are (XSL or Lorel) queries to XML documents, and actions consist of constructing new documents and/or modifying existing documents in the document base, and then placing them into folders, publishing them on the Web, or sending them by e-mail.

Active XQuery [?] extends the Web query language XQuery by ECA rules, which are adapted from SQL-3 and thus called triggers in this work. The event part of such a trigger specifies an affected XML fragment by means of an XPath expression and the update operation (insert, delete, replace, or rename) on this fragment. The condition part is given by an XQuery WHERE clause. The actions available are the previously mentioned, simple update operations and external operations such as sending of messages.

Before and after triggers can be specified in Active XQuery: Before triggers consider the condition and action parts before the given event actually occurs. For after triggers the occurrence of the event is a prerequisite of evaluating the condition and action parts. The trigger components communicate through transition variables – two system-defined variables referring to the old and new nodes and additional variables defined by means of an XQuery LET clause. One can also associate priorities to Active XQuery triggers and specify a triggering granularity for the triggers – statement-level triggers fire once for each set of nodes affected by the change and node-level triggers fire for each node in such a set.

A research work supporting a path-based specification and the detection of composite events *for XML documents* [?] has been also proposed. How this approach does (or even would) scale to the Web is unclear; for example, one cannot relate (primitive or composite) events that have occurred in XML documents distributed on the Web, as the communication of event data is not supported. It

does not represent a full reactive language for the Web, but it could be extended and integrated into a reactive language.

## 4 Production Rules

### 4.1 General Ideas

*What Is a Production Rule?* A production rule is a piece of knowledge organized along an *WHEN condition DO action* structure. The intent of a rule is to evolve the state of the system by executing the action. To guide this evolution, the action will only be applied from a state where the condition is true. The state that results from the execution of the action of a production rule can be incompatible with the state in which the rule was applied. This is common in production rule programs, and can be seen as a noticeable difference with other, monotonic rule programming paradigms such as logic rules.

*Production Rules Based Software Applications* Production rules are used in software applications to encode their logic, or parts of it. As a result, an industrial application may rely on thousands of rules, each rule representing a piece of the knowledge of the company policy. Since industrial applications have a rich life cycle, spanning over several years and involving dozens of persons with various roles, production rule systems have to provide the support for managing this huge amount of information in the long term. This is the purpose of a Business Rule Management System such as ILOG JRules, as illustrated in Section 4.3.

*Production Rules Based Web Applications* Business (production) rules is not the only paradigm on which real-world applications rely. Examples of other paradigms are a multi-tier architecture, and the Web. Web applications will typically use production rules to encode their policy-intensive aspects, that is, the part of their logic that requires the complex handling of the system state, based on the business knowledge of the company. The use of production rules will thus ease the implementation of Web applications with several agents playing different roles.

*Production rules versus integrity rules* Integrity rules are introduced in a relational data base to enforce its correctness and its consistency. In order to maintain referential integrity between primary and foreign keys as data is inserted or deleted from the database, certain insert and delete rules must be defined. Like production rules, integrity rules have a condition part determining in which context they are executed. In fact, integrity rules might be implemented by a production rules system adapted to RDBMS environment. However, specialized integrity system are likely to be more efficient to process huge amount of data characterizing actual databases. On the other hand, the purpose of PR systems is not limited to DB referential integrity and could be applied on various domains.

## 4.2 Description of a Production Rules System

**The Working Memory and the Underlying Data Model** As programs handle data, a programming language defines (more or less implicitly) a data model. Being in essence reactive, rule-based programming languages must ensure, either in the definition of the data model or through specific constructs of the rule language itself, that a rule-based program is able to react to changes in the data. While ECA rule languages introduce the concept of event in the data model and the rule language to this end, production rule languages introduce the concept of *working memory* in the data model, and an *update* statement in the rule language.

The working memory is the finite set of data items (facts, objects... names vary) against which the rules are executed. Data items are explicitly added to, and removed from, this set by the program through dedicated statements (usually *assert* and *retract*). Since changes in data are explicitly notified to the rule engine through the *update* statement, the data can follow basically any data model. Some production rule languages, such as OPS5, include a custom data metamodel in their definitions. Most modern production rule languages are designed to operate on foreign data models, such as those of other programming languages (e.g. Java or .NET's CLR) or XML dialects. Their data model then heavily relies on the introspection mechanisms provided by the foreign data models, such as reflection in programming languages, or the XML schemata.

*Relation with RDF concepts* In the same manner than a production rule language can be adapted to an XML dialect or to a programming language model, it can process RDF models and data. However, some specificities of RDF induce new constraints on the rule language and on the rule engine. For example, RDF resources are type-mutable, and new type labels may be added to a resource during the execution of the rules. Moreover, the model itself could change at runtime through the addition of new properties or types. Another example can be found in RDF with subproperties, or in OWL with transitive, commutative, or inverse relations. To support these additional modeling features, the rule engine must elaborate its handling of the type system, and extend its pattern matching function to take the specific capabilities of properties into account. Depending on the production rule system considered, all or only part of these features will be supported, by additional constructs in the rule language and abilities of the rule engine. This will represent an element of choice for the users, depending on their actual need of RDF specific features.

**Structure and Semantics of a Production Rule** As mentioned before, the overall structure of a production rule is *WHEN condition DO action*. The condition part expresses in which situation the rule should be elected for execution; the action part describes what should be performed as part of executing the rule.

The condition part of a rule contains patterns describing the data that will trigger the rule. When evaluating a rule condition, the production rule engine will search the working memory for data that match all the patterns of the rule

condition. The nature of the constraints expressed by these patterns depends on the data model; typical examples are constraints on the class of objects, constraints on the value of attributes of objects, or constraints on elements of an XML document. Constraints that involve a single data item from the working memory are called *discrimination tests*; constraints that involve several data items from the working memory are called *join tests*. The data items involved in the condition part can be bound to variable names, for reference in the action part.

Each collection of data items from the working memory that match all the patterns of a rule condition gives birth to a *rule instance*. Executing a rule instance consists in interpreting the statements in the rule action on these data items. The statements that can be found in the action part of a production rule usually are those that can be found in any procedural language: assignments, conditionals, loops.

If the production rule language relies on a foreign data model borrowed from a programming language, it may naturally also borrow its statements: for instance a production rule language using the Java object model is likely to express the action parts of its rules in Java, or a Java-like scripting language. If the production rule language matches XML documents, specific statements have to be introduced to express the action parts of rules. Here again, programming or scripting languages can be reused, provided that a mapping is established between the XML data model and the underlying data model of the language used. In all cases, specific statements must be added to handle the working memory: *assert*, *retract*, and *update*. Note that production rule languages that use a custom data model can save the *update* statement if their interpretation of assignment integrates the notification to the rule engine.<sup>19</sup>

```
rule highValuePurchaseByYoungCustomer {
  when {
    c: Customer(age < 21);
    s: ShoppingCart(owner == c; value > 1000.0);
  } then {
    s.manualCheck = true;
    update s;
  }
}
```

**Fig. 1.** Example of a production rule (using the IRL language).

The example in Fig. 1 demonstrates the basic elements of a production rule, here formulated in the ILOG Rule Language (IRL), which is detailed in Sec-

<sup>19</sup> This is true also with foreign programming languages that provide a mechanism for extending the access to their data model with notifications, such as the daemons in some dialects of Lisp.

tion 4.3. In this example, the rule matches two objects in its condition part: an instance of the `Customer` class and an instance of the `ShoppingCart` class. The condition of the rule will be satisfied iff: the value of the `age` attribute of the customer is less than 21, the value of the `value` attribute of the shopping cart is greater than 1,000 (these are discrimination tests), and the value of the `owner` attribute of the shopping cart is a reference to the customer (this is a join test). Note that `Customer` and `ShoppingCart` can be classes of any language such as Java or C#; they can as well be element types from an XML schema.

For each pair of a customer and a shopping cart from the working memory that match all the discrimination and join tests, an instance of the rule is created. In any such instance, the `c` and `s` variables are bound to the customer and shopping cart of the instance. When the action part is executed for one rule instance, the `manualCheck` attribute of the shopping cart is set to true, and the rule engine is notified that the shopping cart has been modified, with an *update* statement.

It must be noted that, although the condition part of the rule is still satisfied by the customer and shopping cart after the rule is executed, the rule will not be executed again. This fundamental principle of production rules, called the *refraction principle*, states that once a rule instance has been executed, the condition of the rule must become false **on the data items of the instance** before the rule can be considered again for execution on these data items. The implementation of this principle is discussed in Section 4.4. As one can imagine, this principle is key in avoiding trivial loops.

Two additional constructs of interest can be used in the condition part of a production rule, namely *not* and *collect*. These construct leverage the finiteness of the working memory to allow the rule author to express conditions on either the *absence* of objects matching a given pattern, or the *collection* of all objects matching a pattern. Rule `tooManyCarts` of Fig. 2 detects a situation where a customer is the owner of two shopping carts or more, while rule `noCart` detects when a customer has no associated cart in working memory.

**Stateless and Stateful Semantics of a Production Rule Engine** We have described above the semantics of the basic operations on a rule, namely: evaluating the condition part of a rule against a working memory, creating a rule instance on a matching tuple of data items, and executing a rule instance. Similarly the *assert*, *retract*, and *update* respectively add or remove an item to/from the working memory, and notify the rule engine that a data item has changed in the working memory. Defining how these operations on rules and on the working memory interact, defines the semantics of the rule engine, and thus of the execution of a rule program. And there are several possible combinations. We present here the two most useful ones, which are related to a stateless and a stateful usage of a rule engine.

The *stateful* case corresponds to applications that correlate data items, or that infer information from the existing data items. A typical example is network or plant supervision, where data from various sources is correlated to synthesize

```

rule tooManyCarts {
  when {
    c: Customer();
    carts: collect ShoppingCart(owner == c) where (size() > 1);
  } then {
    out.println("Customer " + c.name + " has too many (" +
               carts.size() + ") carts.");
  }
}

rule noCart {
  when {
    c: Customer();
    not ShoppingCart(owner == c);
  } then {
    out.println("Customer " + c.name + " has no cart.");
  }
}

```

**Fig. 2.** Example of the *collect* and *not* constructs.

a global picture. In these applications, and in contrast with the stateless case described below, the action of one rule may heavily influence the eligibility of other rules, by modifying the values of attributes involved in the condition parts. As a consequence, the rule engine must carefully take *update* notifications into account in order to ensure that the truth value of the rule conditions, and thus the list of eligible rules, is known at any time. How to efficiently implement this is the cornerstone of the many variants of the Rete algorithm, described in Section 4.4.

The principle of the rule execution algorithm in the stateful case is to maintain at all times which rules are eligible for execution, based on the state of the working memory. The set of these candidate rule instances is called the *conflict set*. As described in Fig. 3, the rule engine picks a rule instance from this set and executes its actions. This may affect the working memory, either by adding data items to it, or by removing items from it, or by updating items that are in working memory. In reaction to this the rule engine updates the conflict set, that is, it creates rule instances for the rules whose condition parts become true, and removes the rule instances whose condition parts become false. The engine operates in this way until the conflict set is empty.

The *stateless* case corresponds to what is called filtering applications, where the rules are used to scan a flow of objects on a one-by-one, or tuple-by-tuple, basis. Examples include data validation, call dispatching, or even some simple form of scoring. In these applications, all the rules typically have the same signature, that is, they match the same number of objects of the same classes. More important, the attributes involved in the patterns of the condition parts of the rules are **never** modified by the action parts of the rules. This property

**Algorithm** STATEFULPREENGINE

1. compute conflict set  $CS$  from working memory  $WM$
2. **while**  $CS$  is not empty **do**
3.   pick a rule instance  $(r, t)$  from  $CS$
4.   execute the actions of  $r$  on the tuple  $t$  of data items
5.   update  $CS$  from the updated  $WM$
6. **end**

**Fig. 3.** Production rule execution algorithm in a stateful context

of the rules entails that the eligibility of the rules on a given tuple of data items will not vary during the execution of the rules. In other words, given a tuple of data items, the engine can evaluate each rule condition in turn, and immediately execute the rule actions if the condition is satisfied. This will yield the same results as the conflict set approach, where all the rule conditions would first be evaluated on the tuple, and then instances of the matching rules would be executed. Furthermore in the stateless case working memory updates can be ignored, or at least delayed until the processing of all the rules on the tuple. This approach is followed by the Sequential algorithm exposed in Section ???. Under the conditions stated above on the rules, the stateless semantics can be viewed as an optimization of the stateful one.

The rule execution algorithm in the stateless case, described in Fig. 4, thus relies on an inner loop working on a given tuple of data items, where the rule engine evaluates the condition part of each rule against the tuple, and if satisfied executes the action part of the rule. In an outer loop the tuples of data items are formed, and fed to the inner loop. How these tuples are formed may vary: they may come from the content of the working memory, in which case the rule engine will have to take care in the outer loop of the *assert*, *retract*, and *update* operations; or they may be handled outside of the rule program, in particular in the rather common case where the rule actions do not add nor remove data items to/from the working memory.

**Algorithm** STATELESSPREENGINE

1. **for each** tuple  $t$  of data items **do**
2.   **for each** rule  $r$  **do**
3.     **if**  $t$  satisfies the condition part of  $r$  **then**
4.       execute the actions of  $r$  on  $t$
5.   **end**
6. **end**

**Fig. 4.** Production rule execution algorithm in a stateless context

### 4.3 ILOG JRules as an Example of a Production Rules System

ILOG JRules<sup>20</sup> is a complete Business Rules Management System (BRMS), that is, a collection of development tools and runtime libraries that help both IT and business people in writing business rules, maintaining them over time and across the enterprise, and deploying them for execution. This section details the concepts leading to the introduction of *business rules*, and then presents the *tools* and *languages* in ILOG JRules that provide support in addressing the challenges arising in the life cycle of a business rules application.

**Business Policies and Business Rules** Business policies gather the knowledge of a company, an organization, etc. describing how operations are to be conducted. They are *a priori* not meant to be processed by a computer, but rather by humans (or business people). As such, they are typically worded in natural language, and stored on paper.

When automation of business policies is considered, a more software-centric embodiment is introduced as business rules, and an unambiguous and executable form of rules is looked for, for instance production rules. Yet, the desire to preserve the interesting property of business policies to be usable by non-technical people has led to the design of Business Rule Management Systems, where domain experts can author rules in a business-friendly format, which is then automatically translated into a format suitable for execution, namely a programming language.

**The Life Cycle of a Business Rules Application** The development and the maintenance of a real production system is a complex activity involving several actors throughout the life cycle of the application, and of the rules themselves. The life cycle of the rules can be summarized as follows, and as illustrated in Fig. 5.

The first protagonist is usually an Architect who analyzes the application and builds the data model on which the rules will be expressed. This model will typically derive from the application's underlying data model, such as a Java object model or a collection of XML schemas. The rules themselves are then authored based on a description of the company policy, for instance a paper documentation. The authoring is either performed by a Policy Manager, or the rules are drafted by a Business Analyst and validated by a Policy Manager. Once the rules are ready they are deployed by an Administrator to the production machine where they will be executed. The authoring-validation-deployment cycle can be repeated as the policies change. Rules may eventually be retired and archived away from the system.

As illustrated in Fig. 5, this rich life cycle has to be supported by a collection of tools. These tools are designed to be used by the various actors in the life cycle, that is, both technical and business people. This also leads to the design

---

<sup>20</sup> ILOG, <http://www.ilog.com/>

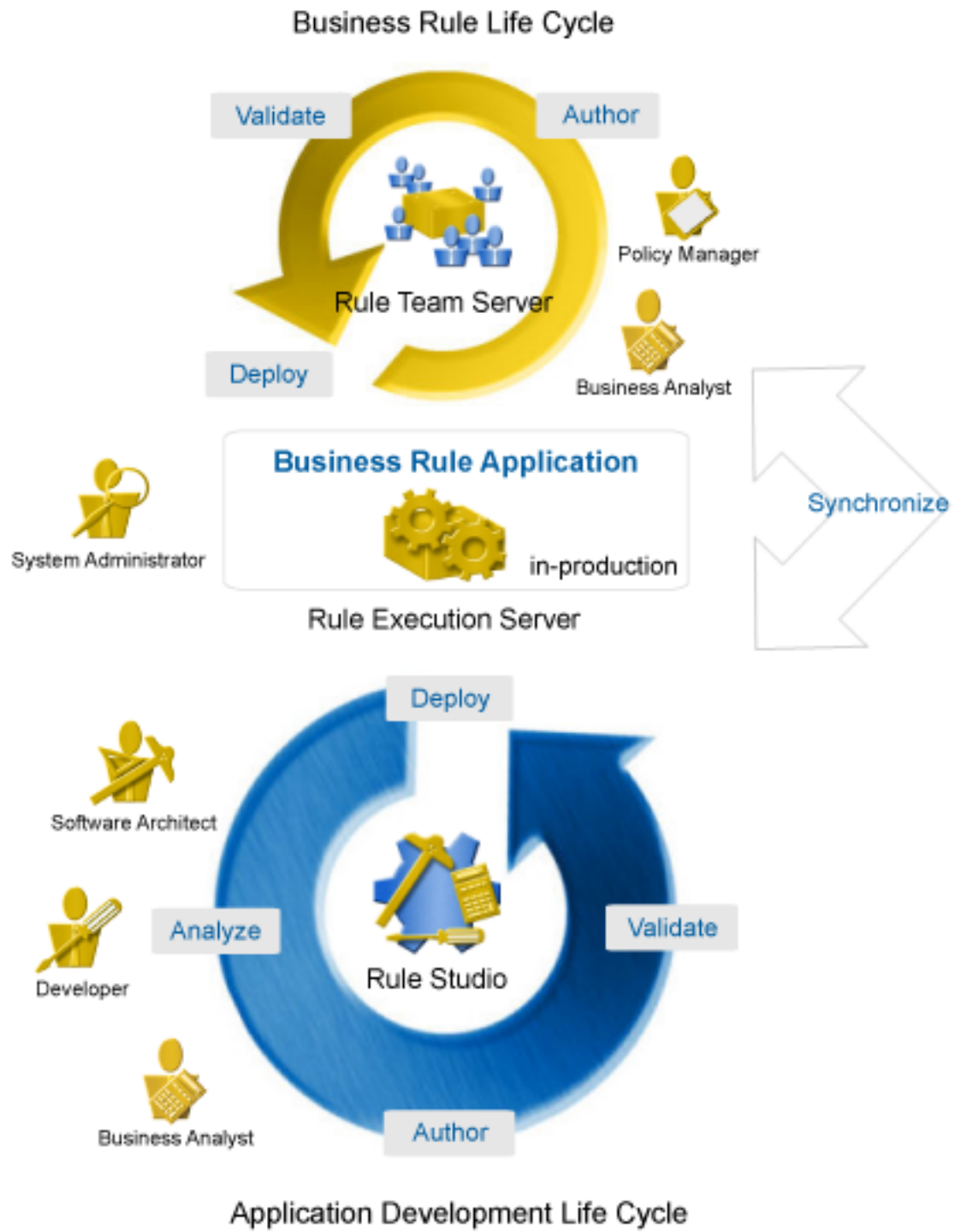


Fig. 5. The life cycle of a production rules based software application.

of various levels of rule languages, all with a sound semantics, but some more adapted to being handled by non-technical rule authors such as Policy Managers, while others are more suited for execution by a rule engine.

**Tools for Business Rules Management** The ILOG JRules Business Rules Management System provides a collection of components to support the complex life cycle of a business rules application. Each of these components is targeted toward some specific actors in the cycle.

*ILOG Rule Studio* is made of a set of plugins to the Eclipse development platform. It is meant to be used by the Architect and the Business Analyst to build the data model on which the rules will be expressed and to define the overall architecture of the rule application. It also allows them to author rules, as well as to create templates for the Policy Manager to fill.

*ILOG Rule Team Server* is a Web application with a Web-based interface. It serves as a workspace where non-technical users such as Policy Managers can work collaboratively to author, edit, validate, organize, and search for the business rules. This is key in evolved applications that can contain thousands of rules and may involve dozens of participants. The projects and the elements they contain are stored in a rule repository, that is, a database connected to ILOG Rule Team Server. Authentication and privilege procedures ensure that actors have rights to perform modification or deployment on the rule repository.

*ILOG Rule Execution Server* is targeted at System Administrators that need to push rule sets to J2EE applications. It allows them to monitor the deployment of rules and to define versions of rule sets.

*ILOG Rule Scenario Manager* is a component accessible both from ILOG Rule Execution Server and ILOG Rule Team Server, and designed to help users test their rules against real data.

**From Business Policies to Executable Rules** If for example a company has the policy “When a customer under 21 buys for more than \$ 1,000 the transaction must be manually checked”, this policy could be expressed by the business rule in Fig. 6. This rule would typically be authored using ILOG Rule Team Server. In order to be executed, it would first be translated into the IRL rule given in Fig. 1, then pushed to an application embedding a rule engine.

*Business Rule Languages* ILOG JRules defines several formats for business rules: the Business Action Language (BAL), the Decision Tables, and the Decision Trees. The Business Action Language, illustrated in Fig. 6, provides a great level of expressiveness, while allowing non-technical users to author rules with a minimal learning curve.

Decision tables and trees provide a concise view of a set of business rules as a spreadsheet or tree. Spreadsheets and trees help the rule author navigate and manage large sets of business rules. Decision tables are rules composed of rows and columns and are used to lay out in tabular form all possible situations which a business decision may encounter, and to specify which action to take

```

definitions
  set c to a customer;
  set s to a shopping cart;
if all of the following conditions are true:
  - the age of c is less than 21
  - the value of s is greater than 1,000
  - c is the owner of s
then
  make it true that s must be manually checked;

```

**Fig. 6.** Example of a business rule (using the BAL language).

in each of these situations. Decision tables allow the user to view and manage large sets of business rules with homogeneous conditions. Decision trees provide the same functionality, but are composed of branches that have decision nodes as their inner nodes, and action nodes as their leaves. Decision trees allow the user to manage a large set of rules with some conditions in common but not all.

ILOG JRules even provides a framework for defining new, specialized business rule languages; this Business Rule Language Definition Framework (BRLDF) will be used by software engineers to tailor a dedicated rule language for business experts to author rules in a specific domain or application.

*Ruleflow* ILOG JRules introduces an additional concept, which is only remotely connected to production rules, but which acknowledges the fact that any program of a reasonable size has both declarative and procedural aspects. This concept is named *ruleflow*, and is used to describe the flow of execution of a program. A ruleflow can be edited using a graphical editor, and is eventually translated into an IRL representation. A ruleflow is composed of tasks, which can be of three kinds:

- A *rule task* is made of a selection from the set of all rules. It also has a number of parameters, among which the choice of the stateless or stateful semantics (see Section 4.2). When a rule task is executed, the corresponding algorithm is activated on the rules composing the task.
- A *function task* executes a function, that is, in essence a series of actions as could be found in the action part of a rule.
- A *flow task* orchestrates a collection of other task (rule, function, and flow tasks) using standard statements such as sequence, conditionals, loops, fork-join, etc.

In addition to the tasks, a ruleflow contains global variables (known as “rule-set variables”) that can be used to vehiculate data between rules and tasks, and marks one of the tasks (usually a flow task) as the main task. Organizing rules into a ruleflow allows the user to handle larger rule-based programs, and to better master their operational semantics. As a result, executing a ILOG JRules program amounts to populating the working memory and then launching the main task of the ruleflow.

*Executable Rules* ILOG JRules defines one language for executable rules, named ILOG Rule Language (IRL). All kinds of business rules are eventually translated in IRL for execution. However rules can be directly authored in IRL using ILOG Rule Studio (but not using ILOG Rule Team Server, which is aimed at non-technical users). They are then referred to as technical rules. As illustrated by the examples in Fig. 1 and 2, the ILOG Rule Language resembles classical programming languages. It allows to use more advanced constructs, such as loops in the action part of the rules.

**Integration in an Application** The rules encode usually only part of the logic of an application. And beyond the logic there is also the logistics, that is, the user interface, the connection to other software such as databases, etc. The interaction and co-operation of various parts of an application, including the part that is implemented using rules, has to be addressed by any rule-based system.

ILOG JRules is a system written in Java, and as such is designed to be interfaced with Java applications, or as a consequence with any programming language that a Java program can be interfaced with. As far as Java is concerned, the main two cases are standard J2SE applications, and J2EE-based systems. In both cases, the principle is that the application embedding the rule engine is the master of the control flow. It is responsible for providing the rules and ruleflow to the engine, of populating the working memory, and of triggering the execution of the rules. The engine then performs the rule execution and returns the control to the application. Since the working memory has been populated with **references** (as opposed to copies) to the Java objects of the host application, the execution of the rules directly implements the application logic on the actual objects handled by the application.

In the J2SE case, this co-operation scheme between the application and the rule engine is implemented using simple Java calls to an Application Programming Interface (API) defined in the ILOG JRules documentation.

*J2EE Deployment* For the integration of rules into a J2EE-based application, the ILOG Rule Execution Server (RES) provides, for the main application servers, ready-to-use J2EE rule execution services that implements this behavior. In addition, the Web-based console of the RES allows system administrators to manage rule-based applications by deploying new versions of rule sets to rule execution services, by enabling or disabling them, and through basic monitoring and statistical analysis tools.

The console provides remote management and monitoring through the JMX technology. The model persists all changes made to a ruleset. The version log maintained by ILOG JRules records the details of the different versions of the ruleset including information on the user who modified data, time of modification, and any comments that have been made.

*Generating Web Services From Rules* An additional feature of ILOG JRules is the ability for the user to generate a Web Service implementation from a rule

set. Deploying production rule sets as Web Services allows users to define and change the behavior of Web servers during their execution. As Web Services are commonly used over the Web to process information in a stateless or a stateful context, this deployment helps bringing production rules into Web applications.

The deployment itself consists in generating a specific Web Service operation for the rule set. The signature of the operation is based upon the signature of the rule set, described in ILOG JRules with formal variables called *ruleset parameters*. As Web Services operates only on XML data, the types in the signature should be compliant with the XML-XSD type system. A binding between XML types and their related object types (Java, C++, C# ) may help to adapt the rules to XML information. The execution of the operation is composed of the following steps: providing the input parameter values to the engine, executing the rules, and returning the output parameter values to the Web Service caller as XML documents. In order to cope with scalability in terms of number of operation calls, ILOG JRules provides the way for pools of rule engines to be managed inside the application server embedding the Web Service.

*Processing Web Data With ILOG JRules* ILOG JRules provides two way to process Web data inside rules by the mean of two automatic bindings: the XML Binding and the Web Service Binding.

Most Web data is defined through XML documents, modeled by XML Schemata. The XML Binding feature transforms a schema into a runtime object model, in such a way that an XML document can be deserialized into a memory object. ILOG JRules enables to execute production rules against such memory objects. Hence, most of XML documents coming from the Web can be processed.

The Web Service Binding feature enables a programmer to invoke external Web Service operations from a production rule as if they were usual Java methods. The WSDL model of the Web Service is first translated into an object model. The port types and their operations are mapped onto classes and methods. As soon as this mapping is achieved, ILOG JRules is able to send or retrieve data automatically represented as objects, to or from the Web Services. This feature is important as Web Services are identified as a usual source of information and processing over the Web.

**Validating and Testing Rules** The ILOG Rule Studio and ILOG Rule Team Server components provide a number of rule validation services based on static analysis techniques. In addition, the ILOG Rule Scenario Manager (RSM) component is an execution test tool to dynamically verify deployable rules. The RSM console is intended for policy managers to manage the rule testing environment, to run rulesets on predefined sets of input data, or to monitor sets of performance tests. Using the RSM console, the users define and manage scenarios, organize them into scenario suites, and set up simulations that compare scenario suites.

Each scenario specifies deployed rulesets to execute and input data to execute the rules on. A set of tests can be applied to track the performance of the execution of the rules. Testing against a baseline report allows for non-regression

testing of the rules as they evolve. In scenario suites and simulations, the user can specify key performance indicators to follow the performance evolution of scenarios over modifications to the rules.

#### 4.4 Implementation of a Production Rule Engine

**Overview of the Rete Algorithm** Forward-chaining inference algorithms, including Rete ([?]), use a match-select-execute cycle. During the *match* stage, the engine creates rule instances by evaluating the rule conditions against the data in the working memory. The *select* stage consists in choosing one of the above-created rule instances. In the *execute* stage, the actions of the selected rule instance are executed, which may modify the working memory and trigger a new *match* stage. This cycle follows the stateful semantics described in Section 4.2 by Fig. 3.

A characteristic of the Rete algorithm is to perform the *match* stage each time the working memory is modified. As a result, the set of potentially executable rule instances is always up-to-date relative to the working memory. A naive implementation of this stage may be time-consuming, due to the huge number of combinations between the data items to be considered. To address this risk, Rete compiles the rule conditions into a *network* so as to minimize the number of patterns that need to be evaluated. The two underlying mechanisms are the sharing of patterns that are common to several rules, and the incrementality of change propagation.

Rete also defines a selection strategy for choosing a rule instance in the conflict set that results from the evaluation of the rule conditions. The conflict set is implemented as an *agenda* of rule instances which are sorted according to this strategy. The engine cycle ends only when the agenda is empty.

**The Rete Network Structure and Behavior** The Rete network is a compact representation of all the patterns expressed in the conditions of the rules. It is a directed acyclic graph structured in four layers, namely:

- The discrimination tree, where the nodes represent the discrimination tests found in the rule conditions;
- The alpha nodes, which form the data item tuples from the individual items;
- The join network, in which the nodes represent the join tests found in the rule conditions; and
- The rule nodes, which form the rule instances.

The input of the graph is the working memory; the output is the agenda. Each node in the network is equipped with a local memory, in which are stored all the data items or tuples that satisfy the pattern associated with the node, as well as the ones associated with the ancestor nodes in the network.

The network reacts to three kinds of events coming from the working memory: insertion of a data item into the working memory, modification of a data item in the working memory, and removal of a data item from the working memory. The