I5-D5

A first prototype on evolution and behaviour at the XML-Level

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Lisbon/I5-D5/D/PU/a1
Responsible editors:	José Júlio Alferes and Wolfgang May
Reviewers:	Piero Bonatti
Contributing participants:	Goettingen, Lisbon, Munich
Contributing workpackages:	I5
Contractual date of deliverable:	31 August 2006
Actual submission date:	31 August 2006

Abstract
This report describes the state of the prototype implementations of the *General Framework for Evolution and Reactivity in the Semantic Web* and of XChange. Besides this report, the deliverable consist also of the prototypes themselves, which are all freely available online from `http://rewerse.net/I5`.

Keyword List
ECA rules, Reactivity, Evolution and updates of data, Language and data heterogeneity

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

A first prototype on evolution and behaviour at the XML-Level

José Júlio Alferes[1], Ricardo Amador[1], Erik Behrends[2], Michael Eckert[3], Oliver Fritzen[2], Wolfgang May[2], Paula Lavinia Pătrânjan[3] and Franz Schenk[2]

[1] Centro de Inteligência Artificial - CENTRIA, Universidade Nova de Lisboa

[2] Institut für Informatik, Universität Göttingen

[3] Institut für Informatik, Ludwig-Maximilians-Universität München

31 August 2006

Abstract

This report describes the state of the prototype implementations of the *General Framework for Evolution and Reactivity in the Semantic Web* and of XChange. Besides this report, the deliverable consist also of the prototypes themselves, which are all freely available online from `http://rewerse.net/I5`.

Keyword List

ECA rules, Reactivity, Evolution and updates of data, Language and data heterogeneity

Contents

Chapter 1

Overview

In the REWERSE WG I5 "Evolution and Reactivity", two approaches are investigated, both based on the paradigm of *Event-Condition-Action (ECA) Rules*: one is the *General Framework for Evolution and Reactivity in the Semantic Web* that supports ECA Rules over *heterogeneous* component languages, i.e., integrates arbitrary event formalisms, query languages and action languages. The other is the Xcerpt/XChange approach whose goal is a *homogeneous* ECA language. Both designs have been described in depth in the previous deliverable [1].

The core deliverable, reporting the current state of the development of prototypes, is thus relatively short. We give a short account of the state of the developed prototypes, including references to the online demonstrators and documentation[1].

The Appendix then contains more detailed background information:

- the continuation of the work on the design of the General Framework, extending the corresponding part of previous deliverable [1], and
- the description of the XChange prototype (from [55]).

The inclusion of this material in the Appendix, though not essential for the deliverable, serves two purposes. First, it makes this document self-contained, not requiring the material from the previous deliverable that defined the languages, framework and architecture. Moreover, it retains in a single document all the updated material needed for a newcomer to understand the framework. This last aspect is particularly important for integrating students to work in this WG.

The General Framework for Evolution and Reactivity in the Semantic Web

Two prototypes of the *General Framework for Evolution and Reactivity in the Semantic Web* described in the previous deliverables are under implementation, independently in Lisbon and in Göttingen. Although with different implementations, the two prototypes are compatible, and partly realize the mentioned general framework. First prototypes and demonstrators are freely available online.

The Prototype at Göttingen

Since Göttingen University has only a small amount of dedicated manpower for REWERSE tasks, the modules at Göttingen University are implemented by students as Bachelor and Master Theses.

[1] Since the prototypes are still under development, the demonstrators can be unstable.

This implies that the deadlines do not match with REWERSE deadlines, and also leads to a deferred integration of the standalone modules with the framework.

The description of the current state can be found in Appendix A. Chapters A.1 – A.5 of that appendix are refined versions of the previous ones dealing with the ECA engine, component modules, and domain modules. Chapter A.6 describes the global architecture that underlies the prototype, and Chapter A.7 describes the development and current state of the implementation. An online demonstrator is available at `http://www.semwebtech.org/eca/frontend`.

The R3 Prototype (Lisbon)

The other prototype of the General Framework defined in the previous deliverable is the r3 prototype, that is being developed in Lisbon. The prototype and its online documentation can be found at `http://rewerse.org/I5/r3`.

Unlike the prototype above, that directly uses the ECA-ML described in the previous deliverable, in r3 reactive rules are understood as RDF-resources (represented according to an OWL-DL foundational ontology, the r3 ontology), and the different components of each reactive rule may be specified (or even composed) using different languages, each of them implemented by specific evaluator sub-engines. The integration of the two prototypes is reduced to a matter of translation between a concrete XML markup (ECA-ML) and the abstract RDF model/syntax of r3. r3 does not enforce a particular concrete syntax/markup. Any request that r3 gets is expected to be translated into an RDF model, which is then added to an internal ontology that includes every resource known to r3. Natively r3 "talks" RDF/XML, but any other XML serialization (concrete markup) of an RDF model is acceptable, provided an appropriate (bi-directional) translator is available, as those for the concrete ECA-ML markup.

Together with r3, and available for demonstrating it, several (sub)engines for component languages are already provided. In particular engines for the XChange/Xcerpt are provided, that make it possible to fully integrate XChange with the general framework (making it possible e.g. to use Xcerpt for querying data in the condition part of rules, or to use XChange for updating XML data in the action part of a rule). Another engine is provided for the language Prova [36] that is being developed within REWERSE in Dresden. Regarding Prova, it is worth mentioning that parts of the r3 prototype were implemented in Prova, thus taking advantage of the collaboration in REWERSE.

XChange

XChange [55] is an ECA language that extends the Xcerpt query language which is being developed in parallel by the REWERSE working group I4 to a *homogeneous* ECA language where each component follows the ideas of Xcerpt. Seeing a sequence of events as a "sequence" of XML elements allows to interpret event detection by *event queries*, using constructs that are derived from a query language and extended by suitable temporal semantics. The matching of the contents of individual events again uses Xcerpt as query language against events as XML fragments. Actions are specified using update patterns again based on Xcerpt terms. Further information can be found at `http://www.reactive-web.de`. A detailed description of the prototype can be found in Appendix B.

Appendix A

The General Framework for Evolution and Reactivity in the Semantic Web

Chapter (Appendix A: ECA Framework) A.1

Introduction

The static aspect of the Semantic Web deals with providing computer-understandable semantic information of Web data. An important task for this is to provide homogeneous languages and formats throughout the Web as an extension to today's *portals* – et least as views over heterogeneous data sources. With URIs, XML, RDF and recently OWL, there is a clear perspective what the Semantic Web will look like.

In contrast to the current Web, the *Semantic Web* should be able not only to support querying, but also to propagate knowledge and changes in a semantic way. This *evolution* and *behavior* depends on the cooperation of nodes. In the same way as the goal of the *Semantic Web* is to bridge the heterogeneity of data formats, schemas, languages, and ontologies used in the Web to provide semantics-enabled unified view(s) on the Web, the heterogeneity of concepts for expressing behavior requires for an appropriate handling on the semantic level. When considering dynamic issues, the concepts for describing and implementing behavior will surely be diverse, due to different needs, and it is unlikely that there will be a unique language for this throughout the Web. Since the Web nodes are prospectively based on different concepts such as data models and languages, it is important that *frameworks* for the Semantic Web are modular, and that the *concepts* and the actual *languages* are independent. As such, besides having a concrete language for dealing with evolution and reactivity, the Semantic Web calls for the existence of a framework able to deal with this heterogeneity of languages.

In this respect, *reactivity* and its formalization as *Event-Condition-Action (ECA) rules*, provide a suitable common model because they provide a modularization into clean concepts with a well-defined information flow. ECA rules can be used for providing a generic uniform framework for specifying and implementing communication, local evolution, policies and strategies, and altogether global evolution in the Semantic Web.

In this paper, we propose an ontology-based approach for describing (reactive) behavior in the Web and evolution of the Web that follows the ECA paradigm. We propose a modular framework for *composing* languages for events, conditions, and actions by separating the ECA semantics from the underlying semantics of events, conditions and actions. This modularity allows for high flexibility wrt. the heterogeneity of the potential sublanguages, while exploiting and supporting their meta-level *homogeneity* on the way to the Semantic Web.

Another important aspect when considering the *Semantic Web* is that of abstraction levels of behaviour. As it will be detailed below, it is our opinion that evolution and reactivity will appear at several levels in the Semantic Web: there will be local basic events and actions, similar to local database triggers; events on local XML data; global rules on the XML level that are able to react on views that include remote data; and application-level events and rules referring to the terms of the ontology of an application. The XChange language mainly deals with events that are communicated (in a push strategy) from outside but are then dealt with locally; also actions are either updates of XML data, or the raising of events. In the general framework below we propose to deal also with different levels of behaviour.

Moreover, the ECA rules do not only operate on the Semantic Web, but are themselves also part of it. In general ECA rules (and their components) must be communicated between different nodes, and may themselves be subject to being updated; also reasoning about evolution might be desired. For that, the ECA rules themselves must be represented as objects of the (Semantic) Web, adhering to an own ontology of rules, and marked up in an (XML) Markup Language of ECA Rules. A markup proposal for active rules can be found already in RuleML [57], but it does not tackle the complexity and language heterogeneity of events, actions, and the generality of rules, as described here. In this report we sketch a markup for active rules, that will be the basis for a (near) future discussion with the WPI1 of REWERSE (Rule Markup) in order to establish the final markup proposal.

Structure of the Presentation. The next chapter, Chapter A.2, develops an ontology for behavior in the Semantic Web. The abstract semantics from the point of the view of the rule level, abstracting from the actual semantics of the components is described in Chapter A.3. In Chapter A.4 we proceed to the component level and investigate the abstract semantics and communication structure of the Event/Query/Test/Action component languages, including some (sample) concrete languages. Here we focus on the event component by describing how the event algebra similar to that of SNOOP [17] is mapped into our framework. Chapter A.5 deals with the (external) functionality of domain nodes wrt. the requirements of the ontology given in Chapter A.2, and with domain brokering. The Web architecture and the formal ontology and metadata of the framework is described in Chapter A.6. The language and service metadata contained in this ontology is also used for establishing actual communication between he component services. The subsequent chapters then are drafts and collections of further ideas.

Publications from this report. A preliminary version of the general framework described in this chapter has been published in [2], the ontology of rules, rule components and languages, and the service-oriented architecture proposal have been published in [45], and the languages and their markup, communication and rule execution model can be found in [44]. A first version of the ECA engine has been described in [6]. The proposal to use CCS in the action part is discussed in [5].

Chapter (Appendix A: ECA Framework) A.2

Ontology of Rule-Based Behavior

We start by summarizing some requirements posed by evolution and reactivity on the *Semantic Web*. Here, after some brief points on the important differences between dealing with the level of the Web versus the level of the Semantic Web, we elaborate on the various abstraction levels of behaviour that need to be considered in the semantic level. This leads a discussion of how to extend the domain ontologies with actions and events, and the various types of events that need to be considered in a general framework. Then, we summarize what kinds of rules, according to their tasks in the framework, have to be covered. Section A.2.2 analyzes ECA rules on the lowest level, namely triggers on the data model level.

We then discuss the main issue of dealing with the above-mentioned heterogeneity of languages. For this we start, in Section A.2.3, by proposing a general structure, rule level ontology and corresponding markup, for (ECA) reactive rules. In it, basically each rule consists of:

- An *event part*, in which there must be a description of something that, if it happens, fires the rule;
- A *condition part*, which, depending on the detected event, may collect some further (static) data and test conditions on both the event and the collected information to check that an action has actually to be executed. Accordingly, this condition part can be further decomposed into one or more *query parts* for collecting data, and one *test part* for checking the (mainly boolean) condition;
- An *action part*, describing what to do when the event is detected and the condition test succeeds.

The components of a rule use different sublanguages for expressing events, queries, tests or actions. Not only the sublanguages of each family share some properties, but there are common properties of all kind of such sublanguages. We analyse the structure of these languages from the semantical and structural aspects and summarize their common aspects: they are algebraic languages, consisting of nested expressions.

A.2.1 Requirements Analysis

This section analyses the requirements for ECA-based evolution and reactivity in the Semantic Web and sets some working hypotheses. We investigate the abstraction levels used in the Semantic Web and its infrastructure, the structure of domain ontologies when dealing with dynamic aspects, then have a more detailed look on the kinds of events that have to be modeled, and classify several kinds of rules that are then actually needed. Finally we "initialize" the way towards our approach by discussing *triggers* as very simple ECA rules and illustrate why this approach provides a good base, but that a compherensive framework must extend this idea in many aspects.

A.2.1.1 Web vs. Semantic Web

Whereas in the conventional XML/HTML Web, ECA models and languages that operate on the data level (as often in the literature, we distinguish between the *data level* and the *information or knowledge level*, which includes an additional knowledge base and reasoning) and on explicit events are sufficient, the situation for a Semantic Web framework is much more complex:

- Model (RDF): With RDF, the same resource can be described at different physical locations, using its URI. Thus, changes in the description of "something" are not necessarily located at a given node.
- Model (OWL): While some research has already been done in the area of queries and static reasoning on the OWL level, the extension to events and actions is still completely open. Domain ontologies must define their (derived) atomic events in terms of changes to the underlying data, and, in case in addition to just execution of rules, *reasoning* is intended, also actions must be described in terms of their effects on the data.

  Developing an approach and case-studies for this has been identified as an important task for understanding what functionality and expressiveness should be provided by languages for describing behavior in the Semantic Web.
- Model and Languages: Rules in the Semantic Web exist on different abstraction levels (see Section A.2.1.2) and should "cover" existing approaches. For this, composite events, conditions involving several nodes, and complex actions must be supported. Here, the existing and expected future heterogeneity has to be taken into account.
- Model, Languages and Architecture: Rules are themselves part of the Semantic Web. For this, they have to be seen as resources. On a smaller granularity, rule components and smaller identifiable parts like individual event descriptions are also resources. Rules and rule components have to be described not only syntactically in terms of a *programming language*, but on the *(rule) ontology level* which is then *translated* to actual, executable specifications in one or more programming languages.
- Languages and Architecture: Languages are resources (that have to be described by a suitable ontology). From this point of view, semantics and processors that implement this semantics are also resources and have to be described and correlated on the ontology level.

A.2.1.2 Abstraction Levels

Data Model Abstraction Levels. As described above, the *Semantic Web* can be seen as a network of autonomous (and autonomously evolving) nodes. Each node holds a *local* state consisting of extensional data (facts), metadata (schema, ontology information), optionally a knowledge base (intensional data), and, again optional, a behavior base. In our case, the latter is given by the ECA rules under discussion that specify which actions are to be taken upon which events under which conditions.

According to the Semantic Web Tower, there are already from the static point of view several abstraction levels.

In classical database systems, the *physical level/model/schema*, the *logical level/model/schema* (i.e., an *abstract data type* that can have different implementations/physical models, and as a database model comes with a generic *database query language*), and sometimes the *export schema* are distinguished.

In the early *network data model*, there was only the physical model where the query language constructs were also directly based on. For *relational databases*, the physical model includes the tables (i.e., ordered sets) and storage data structures (including indexes etc.), the logical model is the relational/SQL schema, and the export schema is in most cases also a relational schema, including views.

With *object-oriented* databases, there came different physical models, including relational and "native" storage. The logical model is the object-oriented model with an ODL/OQL schema; the export model is also the object-oriented one. *Object-relational* architectures are those where the physical model is the relational one, the logical model is split into two levels, the low-level one is the relational model, and the high-level is the object-oriented model, which also serves as export model.

With *XML*, the relationships became even more complex. There are several physical models that can serve for XML data; one of them is again the relational one. XML data can also be stored in object-oriented structures, as done in the early products Tamino (based on Adabas) and Excelon (based on Object Store). Several products claimed to use a "native" XML data model. The "lowest" well-specified XML-related notion is then the *Document Object Model (DOM)* [19] as an *abstract datatype*. Since this model is only on the level of an abstract datatype and does not support any query language, it is not a logical data model. The *logical data model* then is XML, with query languages like XQuery. On the other hand, XML serves as an *export data model* when XML views are defined over relational data [22].

In the research community, models like OEM [50] or F-Logic [33] (for which several internal physical models can be used, e.g., a frame-based one, or a relational one together with Datalog) came up that are used as logical models, or as export models for integrating data from other models.

Adding more abstraction with *RDF*, RDF is seen as the *logical data model*. Then, between it and –numerous– physical models, there can be a layer that uses the XML data model or the relational data model. Using "native" RDF databases, the physical data model can be any data structure that stores triples, or a frame-based structure like F-Logic. When exporting or integrating relational or XML data in RDF, RDF serves as export model.

Considering *OWL*, it qualifies as an *export model* since –by its reasoning– it defines views over RDF data as *logical model* that are queried by the user. From the point of view of the OWL/RDF user, XML then is not a logical model, but "below" this.

In general, the user works on the *export level* (which uses in general a data model which is the same or closely related to the *logical level*). Queries against the export level are mapped down to the logical level.

Abstraction Levels in the Conventional Web and in the Semantic Web. For the conventional (XML) Web and for the Semantic Web, there are different "towers" of data models:
In the conventional Web, there are two levels; the upper of which is XML:

- Data level. Files, SQL databases, XML databases etc. Here *local* behavior of the *databases* (e.g., integrity maintenance) is located.

- Logical Level: XML. Here *local behavior* of the *nodes* (e.g., local application behavior) is located. Remote actions between tightly coupled nodes (i.e., that use common XML Schemas etc.) are also possible on this level. Interfaces for Web-Services like SOAP are also on this (syntactical) level.

In the Semantic Web, the structure of the levels has to be seen from a local and from a global aspect:

- Data level. Files, SQL databases, XML databases etc. Here again *local* behavior of the *databases* (e.g., integrity maintenance) is located.

- Local Logical Level: this level is provided by an XML or relational model, sometimes omitted (for RDF databases). Here *local behavior* of the *nodes* (e.g., local application behavior) is located. Remote actions between tightly coupled nodes (i.e., that use common XML Schemas etc.) are also possible on this level.

- Global Logical/Integrated Level: RDF. Here, *integrated* behavior (i.e., simple push/pull communication) will be located; messages between loosely coupled nodes that communicate in an application domain will be exchanged on this level.
- Semantic Level: OWL. Here, *intelligent integrated behavior* will be located, i.e., business rules, policies and strategies that often use derived data (and derived events).

Abstraction Levels of Behavior. In the same way as there are different levels from the static point of view, behavior can be distinguished wrt. these levels (programming language/data structure level, logical level, integrated level, and semantic level), and with different scope (local or global).

On all these levels, there is behavior. The user and the applications rely on the behavior on the *export model level* (OWL), whereas the actual, persistent changes to the database take place on the *physical level* (SQL, XML). On this lowest level, several proposals for "triggers" for XML data analogous to the SQL triggers exist, where with the distributed environment of XML data on the Web now, two types can be distinguished:

- Local triggers where event, condition, and action components use only the local database (like for SQL triggers),
- "Web-Level triggers" whose event component is based on data-level events in the local database, the condition component uses the local database and possibly also remote ones, and the action component can include arbitrary actions on the Web level (sending messages, interactions via HTTP, SOAP),

Such trigger concepts for XML and RDF as simple ECA rules will be investigated in more detail in Section A.2.2.

For realizing behavior in the Semantic Web, also vertical transmission between the levels is required, including the XML and RDF models. A classification of types of rules wrt. their functionality and roles in the whole framework will be given in Section A.2.1.5.

A.2.1.3 Domain Ontologies including Dynamic Aspects

The coverage of *domain ontologies* differs already in the classical data models (and conceptual models): in the relational model and in the Entity-Relationship model, a domain ontology consists only of the static notions, expressed by relations and attributes, or entity types with attributes and relationships (similar in first-order logic). In the object-oriented model and in UML, the static issues are described by classes, properties and relationships, and the dynamic issues are described by actions; in UML also their effects can partly be described.

A *complete* ontology of an application domain requires to describe not only the static part, but also the dynamic part, including actions and events (cf. Figure A.2.1):

- describing actions in terms of agents, preconditions, and effects/postconditions,
- describing events, i.e., correlating actions and the resulting events, and specifying composite events, and
- describing composite actions (processes),
  - in fact, business rules themselves can also be seen as parts of the ontology of an application.

For designing a Semantic Web application or service, in general ontologies of several domains interfere:

- Application domain ontologies define the static and dynamic notions of the application domain (banking, traveling, etc.), i.e., predicates or literals (for queries and conditions), and events and actions (e.g. events of train schedule changes, actions of reserving tickets).

- Application-independent domains that *talk about* an application; mostly related to classes of services (messaging, transactions, calendars, generic data manipulation). They can be generically used in combination with arbitrary application domains. They also provide static and dynamic notions.

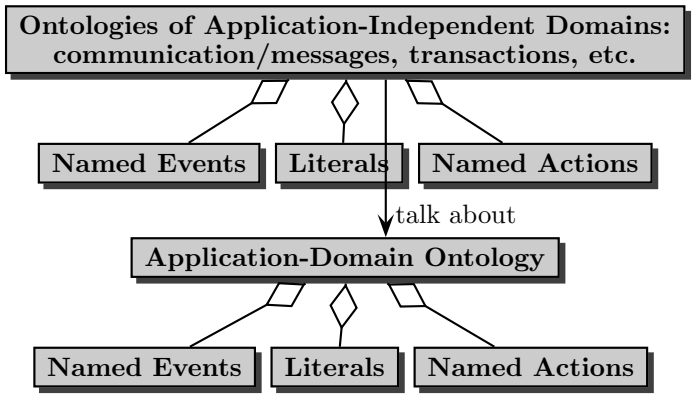


Figure A.2.1: Kinds and Components of Ontologies

A.2.1.4 Events

An important aspect is the analysis of types of events that have to be considered. The ontology of events has to consider the abstraction levels and the application-dependent and application-independent domain ontologies. There are different kinds of (atomic) events (see explanations below and Figure A.2.2):

- events of a given application domain (e.g., in banking, travel organizing, administration); such atomic events are described in terms of the ontologies of the application domain,
- generic parametric events that are not from any specific application domain but that instantiate generic event patterns, e.g., communication ("receive a message about ...") and transactional events that *talk about* application domains.

  Data level events are also a special kind of such generic (= generic to the data model) events.

We start with the conceptually simpler events in application-independent domains.

Events in Application-Independent Domains. The application-independent domains provide *patterns* of atomic events that are ontologically independent from the actual application, but *talk about* an application; mostly related to classes of services, e.g., messaging or operations in a data model (cf. Figure A.2.3). In general, such events are associated with a certain node.

Data Model Events. In the same way as for SQL data, there are atomic generic data model-level (i.e., XML or RDF) events (as will be discussed in more detail Section A.2.2 for triggers). The actions that raise such events are operations of the underlying data model. Thus, they are *generic* in the sense that they apply to the schema level of a given data model and only their parameters, i.e., names of classes and relationships and actual data, are taken from the application.

Other Generic Events. In the same way as the above data model events, there are *generic* events that are not raised by data model-level updates but belong to high-level application-independent ontologies that often *deal* with application-specific information:

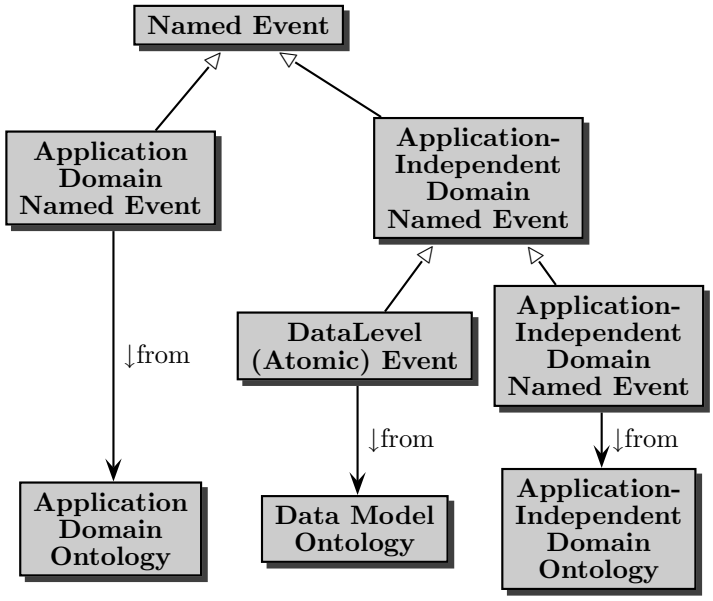


Figure A.2.2: Ontology of Named Events

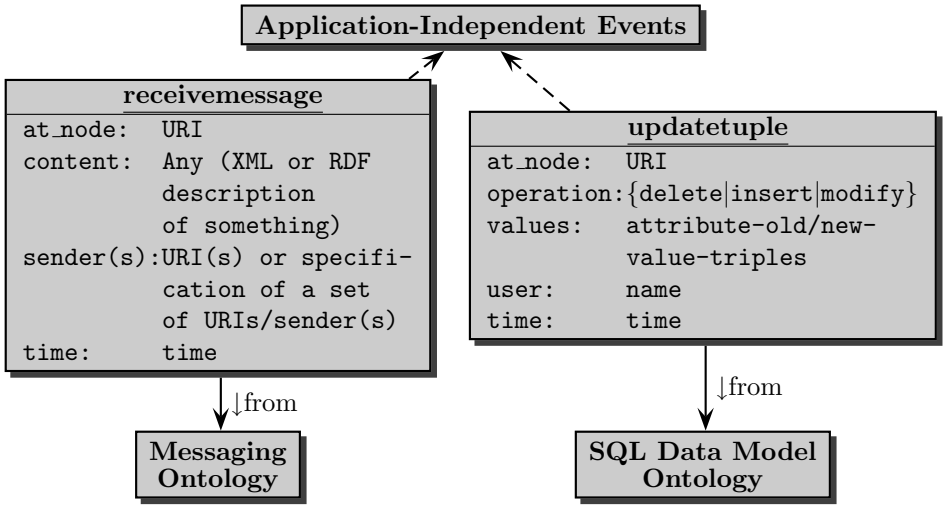


Figure A.2.3: Example Events of Application-Independent Domains

- Communication: receiving or sending messages. Such events are parametric with e.g., sender and receiver (as URIs) and application-specific data (content of the message, possibly the roles of sender/receiver in this communication).
- Transactions: start, commit, and rollback of transactions. E.g., the "decision" or the message from another node that a transaction cannot be executed successfully raises such an event.

In the following, we also call such events *generic parametric events*.

Application Domain Events. Atomic application domain events are the visible happenings in the application domain. High-level rules, e.g., *business rules* use application specific events (e.g., professor_hired($object, $subject, $university)). Such events must be described by the ontology of an application.

Actions vs. Events. In contrast to simple data-level events on XML or RDF data, on the application-level there is an important difference between *actions* and *events*: an event is a visible, possibly indirect or derived, consequence of an action. For instance, the action is to "debit 200E from Alice's bank account", and visible events are "a debit of 200E from Alice's bank account", "a change of Alice's bank account" (that is also immediately detectable from the update operation), or "the balance of Alice's bank account becomes below zero" (which has to be derived from an update). Another example is the action "book person P on flight LH123 on 10.8.2005" which results in the internal action "book person P for seat 42C of flight LH123 on 10.8.2005" and the events "a person has been booked for a seat behind the kitchen", "flight LH123 on 10.8.2005 is fully booked", "there are no more flights on 10.8.2005 from X to Y", or "person P has now more than 10.000 bonus miles". All these events can be used for formulating (business) rules.

Reaction to Events. For actually reacting on events, a node must be *aware* of them. Here, some issues have to be considered in the Semantic Web since events can be derived ones, and that they are not necessarily located at a certain node.

- Explicit events are events that have a *direct* relationship with an action (and, from the implementation point of view, with a code fragment where they can be "caught"). The database-level events are obviously explicit ones: an update of the database is an action *and* is seen as an event. The receipt of a message is also an explicit event (although originally resulting from a different (sending) action, it can be caught on the socket level of a node).

- Implicit and derived events: an event is derived if it is defined based on other events, e.g. "flight LH123 on 10.8.2005 is fully booked" or "there are no more flights on 10.8.2005 from X to Y" are derived from "person P is booked for seat 5C of flight LH123 on 10.8.2005" under certain conditions. Note that there can be multiple derivations for a derived event (for the second one above, also e.g. "flight AF678 on 10.8.2005 is canceled").

Raising and Derivation of Events. The task of becoming aware of implicit events is not the task of the rule execution, but of application-level reasoning, based on the application ontology. Thus, there are *derivation rules* also for events (that can be seen as event-condition-action rules where the action consists of raising an event); see Section A.2.1.5.

Localization of Events. Orthogonal to being derived or not, application-level atomic events can be associated with a certain node or can describe happenings on the Web-wide level:

- local events: these happen locally at the node. E.g., data model events are in most cases used locally (by triggers, which then can raise higher-level events or trigger a remote action)

- remote events: From the point of view of a rule, a *remote* event is an event that is local (and can be localized) at another node, e.g., "if Amazon offers a new book on X". Here, event detection can be done e.g. by monitoring the node (*continuous querying*) or by a *publish-subscribe-service*.

- global events: global events happen "somewhere in the Web", e.g., "a new book on the Semantic Web is announced", or "election of a new chancellor of Germany". *Global* events are (mostly) application-level events where it is not explicitly specified where they actually occur.

In these cases, event detection is even more complicated since it must also be searched and derived *where* and *how* the event can be detected. Rules using global events require appropriate communication and notification mechanisms by the Semantic Web infrastructure (that can in turn also be based on ECA rules). For dealing with global (not located or locatable at a certain node) implicit (derived), the Semantic Web must provide *event broker* functionality.

Temporal Delay and Event Wrapping. Event brokering leads to the effect that the time-points of actual events and the event detection may differ. The actual infrastructure will probably also use messaging functionality, i.e., not raising an event, but packing it in a message or event "I became aware that ...".

Example 1 *Consider a customer C who wants to buy a Christmas tree, C can state the following rules:*

- *"if some X announces to sell Christmas trees, go there". The rule is correct, but not "complete": Probably, C will not become aware of the event of announcing, so he will never get a Christmas tree.*
- *"if I become aware [by a message] that some X announced to sell Christmas trees, go there". This rule is more complete, but, if C becomes aware too late (e.g., after New Year) he will also go there.*
- *The correct rule is thus "if I become aware [by a message] [before Christmas] that some X announced to sell Christmas trees, go there".*
- *Nevertheless, the real-world formulation of the rule will be of the style "if some X sells Christmas trees, I should go there" – which is formally based on an event that X sells a tree to some Y (which will eventually happen, but C will most probably not be informed about it).*

Thus, in a future stage of Semantic Web rules, the interpretation of rules should provide a reasoning-based rewriting of rules to get their intended semantics.

Composite Events. Composite events are subject of heterogeneity in that there are multiple formalisms and languages for describing them. As already mentioned, most of them use *event algebras*. The target framework for the Semantic Web must support this heterogeneity.

A.2.1.5 Types of Rules

In our ECA-based approach, the behavior of domains *described* in an axiomatic way by (OWL) ontologies is specified and implemented by ECA rules. In a first step, these descriptions have to allow to run applications by ECA rules. In the next step, the ontology has to be extended to preconditions and effects as a base for *reasoning*.

There are several types of rules that are used for actually specifying the ontology and the behavior of an application:

- Rules that axiomatize the *ontology*, i.e., mandatory relationships between actions, objects, and events that are inherent to the domain. The correctness of the rules must be proven against the ontology.
- Rules that specify a given *application* on this domain, e.g., business rules. Changing such rules result in a different behavior of the application.

ECA Rules. From the external user's point of view, *ECA-Business Rules* specify the actual behavior: "when something happens and some conditions are satisfied, something has to be done". Here, events and actions refer to a very high and abstract level of the ontology.

- Such rules are "actual", user-defined ECA rules since they trigger an action upon an event "to keep the application running". Such rules exist on different abstraction levels and granularity, designed to the notions of the application domain. Changing them changes the behavior of the application.
- Internally, such rules are also used for implementing mechanisms for detection of derived and composite events on the respective level.

ECE Event Derivation Rules: Providing High-Level Events. For implementing high-level rules, it is necessary that these high-level events are provided somehow: They must be *derived*.

- horizontal ECE rules: Here, the event is derived from another high-level event under certain conditions, e.g., *"when a booking for a flight is done, and this is the last seat, then the plane is completely booked"*. The rule is an E-C-E (event-condition-event) rule, e.g., the "action" consists in deriving/raising an event. The events are logically related and inherent in terms of the application. Changing such rules would invalidate the application wrt. its ontology.

- upward vertical ECE rules: an abstract event is derived from changes in the underlying database, e.g., *"when the arrival time in a database of a flight of today is changed, this is actually a delayed flight"*. The rule is again an E-C-E (event-condition-event) rule. The events are not logically related and inherent in terms of the application, but are related due to the physical implementation of the application (i.e., since an explicit message "flight F is delayed" is missing, and only visible due to a modification of the database". Changing such rules would invalidate the application wrt. its ontology.

  As another example, "professor_hired($object, $subject)" is (locally) derived at a node from an insertion of a fact into an SQL, XML, or RDF database.

These rules correspond to the *bottom-up* semantics of derivation rules: Given the body, do the head. While "classical" ECA rules are *active* rules, the above enumeration presented several kinds of *derivation* rules. The main difference of these wrt. classical *derivation* is that the latter define continuously existing *views* and are used for *querying*. In contrast, the *event derivation rules* "fire" only once when an event is detected and another event must be raised. Thus, these *ECE* rules are more similar to ECA rules than to derivation rules.
The derivation of events by such rules can be done similar to views:

- bottom-up style: they can be "materialized" by raising them explicitly when (and where) they occur (even if probably nobody is actually interested in them), or

- top-down style: when an application uses a derived event, it runs the rule locally.

High-level events can also be raised as side-effects of high-level actions, see below. When designing rules, it must be cared that such effects do not cause any behavior twice.

ACA/ACE Rules: Talking about High-Level Actions. High-level actions like "(at a travel agency) book a travel by plane from Hannover to Lisbon" cannot be executed directly, but there must be another rule that says how this is implemented (by searching for connections, possibly via Madrid). Such rules are *reduction* rules that reduce an abstract action to actions on a lower level.

On the other hand, there may be another business rule that should be executed whenever somebody does a plane travel from Germany to Portugal, putting this person on a list for sending them advertisements about (questionable) tax saving tricks by investing in resorts in the Algarve. The latter rule should not be defined on the basis of "if there are bookings for a person via some places that lead from Germany to Portugal" (which would e.g. also fire if a Tyrolian flies from Innsbruck to Munich and then to Lisbon; the german tax tricks do probably not apply to him), but could –most declaratively– directly use the *abstract action* "book a travel by plane from a german airport to a portuguese airport" for firing the rule.

Thus, there are rules that regard (abstract) actions as events – or in the above case more exactly, use the event of *committing an abstract action*. Such rules can be expressed based on transactional events, or on messages ("if we get the message that such an abstract action should be executed"), but in both cases this blurs the declarativity that the *action* actually is the reason to react.
Thus, there are several kinds of ACA rules:

- horizontal reduction ACA rules, e.g., *"the action of transferring 200E from account A to B is implemented by debiting 200E from account A and deposing 200E on account B"*. This rule is kind of an A-C-A rule that explains a composite action by its components, both still in terms of the application domain.
- vertical reduction ACA rules, e.g., *"the action of debiting 200E from account A is realized by reading the account value, adding 200 and writing it"*. This rule is also a kind of an A-C-A rule that reduces a composite action into its components on a lower level.
- horizontal non-reduction ACA rules see an action (that has to be executed for itself) as an event that should trigger another action, known also as *rule chaining*.

  This kind of ACA rules is more directly related to ECA rules. Changing such rules would not invalidate the application wrt. the ontology, but just change its behavior.

The above reduction ACA rules correspond to SQL's `INSTEAD` triggers: in SQL, `INSTEAD`-triggers are used for specifying what updates should actually be done instead of inserting something into a view (which is not possible), whereas here, simpler actions are specified instead of an abstract one. Such rules are closely related to the *top-down* semantics of derivation rules: To obtain the head, realize the body (cf. Transaction Logic Programming [14]). These rules describe actions that are logically related and inherent in terms of the application. Some of these rules are inherent to the ontology of the underlying domain, others specify only the behavior of a given application (including e.g. policies that are not inherent to the domain).

Since high-level events can also be seen as observations, it is also reasonable to raise them when an appropriate action is executed. In most cases, this amounts to a simple mapping from high-level actions to raise high-level events: the action "hire_professor($object, $subject)" directly raises the event "professor_hired($object, $subject)" at the same node (and is internally executed by inserting a fact into an SQL, XML, or RDF database; see downward ACA vertical rules above). The execution of both, ACA and ACE rules, must usually be located at the node where the action is executed (which makes completely sense because ACA is actually the algorithm to execute an action, and ACE is the communication of its effect on a high level).

Low-Level Rules. The base is provided by update actions on the database level (to which all abtract actions must eventually be reduced to actually change the state of any node) and low-level ECA rules, e.g., database triggers for referential integrity or bookkeeping.

Here, neither the event nor the action is part of the application ontology, but both exist and are related only due to the physical implementation of the application. Such rules guarantee –together with the RDF views on the database– the integrity of the model; thus, when verifying a process they also have to be taken into account.

Summary and Interference. The resulting information flow between events and actions is depicted in Figure A.2.4.

Example 2 (Actions and Events) *Consider the following scenario: A –rule-driven– review process leads to the acceptance of a paper. Here, "acceptance of a paper" is an action which is also an event on the Web level – "the paper P of A has been accepted for conference C". This event is communicated inside the program committee by mail (*push *communication). An internal rule of the conference of the form "when a paper is accepted, send a message to the author and list it on the conference Web page". By the latter, the event of "the paper P of A has been accepted" becomes actually accessible for the public. Communication in the Semantic Web then should lead to firing other business rules, e.g., at the DBLP publications server and at citeseer, where lots of "business rules" (later) react upon.*

Such rules in the Semantic Web are not formulated on the level of messaging, but assume the notification about events as given (which is, on a lower level, done by messaging).

The author of an accepted paper probably has a rule "when a paper is accepted (event), then book a travel to the conference". This action is submitted as a message "Person P wants to book a

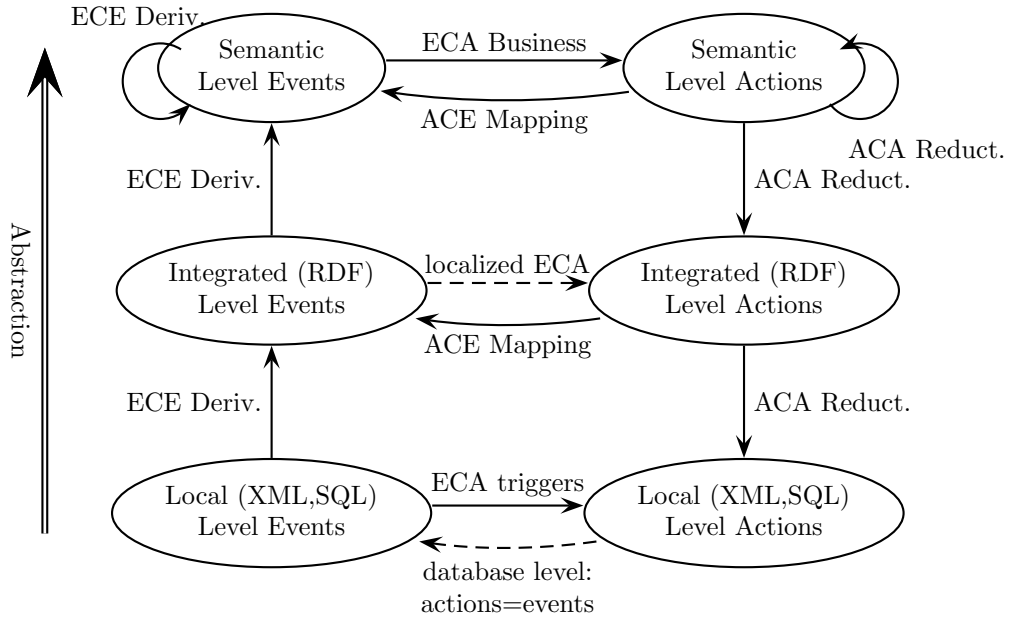


Figure A.2.4: Types of Rules

travel from X to Y on date D" to a travel agency. The travel agency reacts on incoming messages by searching for connections and book an available one, possibly by flights AA123 from X to Z and BB456 from Z to Y (horizontal rule for decomposing an action into its constituents). These actual bookings of flights are then submitted to the airline, seats are assigned, and the booking actually takes place by modifications of the database content (vertical rules). Assume that this booking reserves the last seat in AA123.

On the other hand, there are several other rules that should fire in this process, e.g., a rule that removes all completely booked flights from some list, and that raises the price for all flights between two destinations M and N in case that more than 50% of the total capacity on this connection for that day is booked.

Both can be done by vertical rules, reacting on database events for booking actual seats, or on the higher level, e.g. "on any booking between M and N (event), check all flights between two destinations M and N, and in case that more than 50% of the total capacity on this connection for that day is booked, raise the price for the remaining ones by 10E". In the latter case, the booking action is immediately also seen as an event "somebody books ...".

Moreover, there can be a business rule "whenever a person P books a travel from country C to country D, do ...". If X is located in C and Y is located in D, but Z is located in a different country, then, this event cannot be detected from the actual, independent bookings of the individual flights. Instead, the action "book a travel from X to Y on date D for person P" should be considered as a high-level event for immediately firing appropriate rules.

A.2.2 Simple ECA Rules: Data Model Triggers

A simple form of active rules that is often provided by database systems, are *triggers*. As shown in Figure A.2.4, these triggers on the database level form the lowest level of rules. Reacting directly to changes in the database, they provide the basic level of behavior. Triggers are simple rules on the *(database) programming language and data structure level*. They are associated with the logical level (i.e., not referring the the implementation of the logical data model, but acting on its notions). They follow a simple ECA pattern where the conditions are given in the database query language and the action component is given in a simple, operational programming language. In

SQL, triggers are of the form

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

In the Semantic Web, this base level is assumed to be in XML or RDF format. While the SQL triggers in relational databases are only able to react on changes of a given tuple or an attribute of a tuple, the XML and RDF models call for more expressive event specifications according to the (tree or graph) structure.

A.2.2.1 Triggers on XML Data

Work on triggers for XQuery has e.g. been described in [11] with *Active XQuery* (using the same syntax and switches as SQL, with XQuery in the action component) and in [4, 51], emulating the trigger definition and execution model of the SQL3 standard that specifies a syntax and execution model for ECA rules in relational databases. The following proposal refines our previous one developed in [3].

We propose to use *simple-expressions* that consist only of the downward axes child and descendant, and the final step is allowed to be an attribute step for basically locating events of interest .[1]

- `ON {DELETE|INSERT|UPDATE} OF` *simple-expr*: if a node matching the *simple-expr* is deleted, inserted, or updated,
- `ON MODIFICATION OF` *simple-expr*: if anything in the subtree rooted in a node matching the *simple-expr* is modified,
- `ON INSERT INTO` *simple-expr*: if a node is inserted (directly) into a node matching the *simple-expr*,
- `ON {DELETE|INSERT|UPDATE} [SIBLING [IMMEDIATELY]] {BEFORE|AFTER}` *simple-expr*: if a node (optionally: only sibling nodes) is modified (immediately) before or after a node matching the *simple-expr*.

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

Similar to the SQL `STATEMENT` and `ROW` triggers, the granularity has to be specified for each trigger:

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

The implementation of such triggers in XML repositories can e.g. be based on the *DOM Level 2/3 Events* or on the triggers of relational storage of XML data. Events/triggers on this logical level are local (and internal) to the database that provides an RDF view to the outside. Usually, the actions are local updates of the database (that then effect the RDF view indirectly), or they raise events on the RDF level (it is see also below), but it is also possible to send XUpdate or HTTP requests or SOAP messages to other nodes, or to state remote XML updates explicitly:

[1] Alternative possibilities are: allow arbitrary XPath expressions, or allow even for specifications like ON UPDATE OF *subexpr* IN *xpath-expr*.

```
ON INSERT OF department/professor
let $prof:= :NEW/@rdf-uri, $dept:= :NEW/parent::department/@rdf-uri
RAISE RDF_EVENT(INSERT OF has_professor OF department)
  with $subject:= $dept, $property:=has_professor, $object:=$prof;
RAISE RDF_EVENT(CREATE OF professor)
  with $class=professor, $resource:=$prof;
```

XML: Local and Global Rules in the "Conventional" XML Web. The above events occur always local in a node and can be detected at this node.

Rules on the XML level of the Web can either be local to a certain node, or they can include Web data, e.g., reacting on events on views that include remote data, or raising actions on the Web. For that reason, we call them *Web Level Triggers* (note that these can already be applied to the conventional non-semantic Web; whereas for integration reasons in the *Semantic Web*, the level of RDF events is preferable).

Actual rules on this level usually are not only based on atomic data-level events, but use own event languages that are based on a set of atomic events (that are not necessarily just simple update operations) and that usually also allow for composite events. Their event detection mechanism is not necessarily located in the database, but can be based on the above triggers.

Such rules require knowledge of the actual XML schema of the corresponding nodes. Provided a mapping between rules on the XML level and those of the RDF view level, the implementation can (more efficiently) be kept on the XML level, whereas reasoning about their behavior can be lifted to the Semantic Web level. A Semantic Web framework should also support this kind of rules.

From the *Semantic* Web point of view, events on the XML level should usually not be communicated to other nodes (except very close coupling with nodes using the same schema); instead semantic events should be derived from them.

A.2.2.2 Triggers on (Plain) RDF Data

RDF triples describe properties of a resource. In contrast to XML, there is no subtree structure (which makes it impossible to express "deep" modifications in a simple event), but there is some metadata[2]. A proposal for RDF events can be found in RDFTL [51, 52]. The following proposal refines our previous one developed in [3]:

- `ON {INSERT|UPDATE|DELETE} OF` *property* `[OF` *class*`]` is raised if a property is added to, updated, or deleted from a resource (optionally: of the specified class).
- `ON {CREATE|UPDATE|DELETE} OF` *type* is raised if a resource of a given class is created, updated or deleted.
- `ON NEW TYPE` is raised if a new type is introduced,
- `ON NEW INSTANCE OF [OF` *type*`]` is raised if a new instance of a type is introduced,
- `ON NEW PROPERTY OF INSTANCE [OF` *type*`]` is raised, if a new property is added to an instance (optionally: to a specified class). This extends `ON INSERT OF` *property* to properties that cannot be named (are unknown) during the rule design.

Besides the `OLD` and `NEW` values mentioned for XML, these events bind variables Subject, Property, Object, Class, Resource, referring to the modified items (as URIs), respectively. Trigger granularity is `FOR EACH STATEMENT` or `FOR EACH TRIPLE`.

Note that for plain RDF data, there is no *derived* information (subproperties, subclasses, domains, etc.). Thus, events *immediately* correspond to operations (inserting, deletion or modification of corresponding triples).

[2] we consider here only rdf:type as a special property. RDFS vocabulary, including RDFS reasoning is considered in the next section.

A.2.2.3 Triggers on RDFS and OWL Data

For the Semantic Web more interesting are triggers on RDFS and OWL data. In these cases, *reactivity* must be combined with *reasoning*. A proposal for triggers in such a scenario has been developed and implemented in [46, 64] based on the Jena Framework [31]; see also Section A.7.3.

Application-level events (that must be characterized appropriately in the application ontology) can then be raised by such rules, e.g.,

```
ON INSERT OF has_professor OF department
  % (comes with parameters $subject=dept, $property:=has_professor,
  %    and $object=prof)
  % $university is a constant defined in the (local) database
RAISE EVENT (professor_hired($object, $subject, $university))
```

which is then actually an event (e.g., professor_hired(*prof, dept, univ*)) of the application ontology on which a "business rule" of a publisher could react that says, if a new professor is hired at a university, then the appropriate list of textbooks should be sent to him. Note that in the above trigger, this event is only raised – the complete issues of communicating it and detecting it by the node that actually processes the business rule have to be dealt with separately.

A.2.2.4 Triggers vs. ECA Rules

As shown in Figure A.2.4, triggers on the database level form the lowest level of rules. They deal with the data model level instead of the application level. Thus, their "home" is inside the database, using notions of the database model, and their implementation often event depends an the availability of information from the physical level of the database. Especially, their "events" coincide directly with the update operations of the database, which are also the *actions* on that level. Triggers are not necessarily subject of the modularization of our model. In contrast, often triggers, although following the ECA paradigm, are only very restricted. They are usually closely intertwined with the database (e.g., in relational databases), although very efficient external implementations exist, cf. TriggerMan [27].
In case that triggers are implemented external to a database, the data items that are concerned by the triggering event occurs must usually be identifiable:

- SQL: OLD and NEW are the modified tuples; all related data can be identified externally by foreign keys. Furthermore, usually, the ROWID is used that identifies a tuple internally.
- XML: OLD and NEW should at least allow to access the subtrees, but also parent or ancestor nodes may be of interest. For this, an internal identifier must be accessible. Note that there may also be triggers that do a modification inside the database relative to the modified node, e.g., in the subtree.
- RDF: OLD and NEW are the current nodes. Since they in general have a URI, they can be identified without any problem.

These triggers are usually defined in a homogeneous way on the programming language level (i.e., data model events of the data model's update language, queries in the data model's query language, and actions are given in as data model updates or as a program segment in a programming language that includes the data model's query language). On the higher level, ECA rules make use of more abstract languages.

Rules on the Ontology Level. In rules on the level of an application, the events, conditions and actions refer to the domain ontology. Even more, often the local knowledge of the node is not sufficient, but in general, OWL events refer to the distributed scenario. Thus, it is not always appropriate to locate them in a database like triggers.

Local, active rules working on this level are e.g. available in the *Oracle Rule Manager*: Events are simlar to tuples of a view (but volatile) and can e.g. be raised by triggers. Then, active rules can be defined that react on such events, including a restricted form of composite events. We give here just an example to complete the "upward" transmission of events:

ON (professor_hired($prof, $dept, $univ))
WHEN $Books := *select relevant books for people at this dept*
BEGIN *do something* END

More complex rules also use composite events and queries against the Web. Composite events in general consist of subevents at different locations. Additionally, higher-level events are in general not explicitly raised. In the above example, both was simple: the source explicitly raised professor_hired($object, $subject, $university), and the publisher can e.g. register at all universities to be notified about such events. In general, events like "when a publication p becomes known that deals with ...") cannot be detected in this simple way, but must be derived and obtained from other, more general information. Here, Semantic Web reasoning comes is required even for detecting atomic events "somewhere in the Web". The timepoints of actual events and the event detection may differ.

A.2.3 ECA Language Structure

After having discussed and analyzed the requirements for specifying and implementing behavior in the Semantic Web by active, ECA-style rules, we develop now the structure of such rules and the required languages.

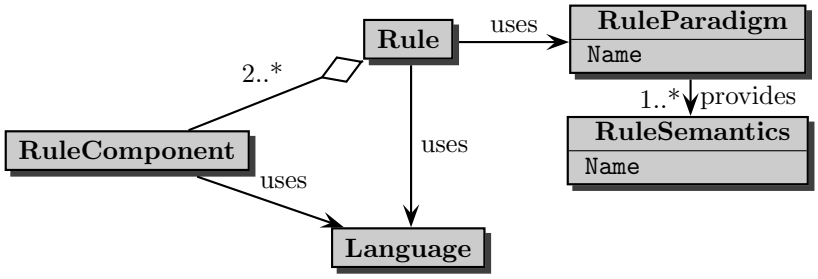
A.2.3.1 Language Heterogeneity and Structure: Rules, Rule Components and Languages

Rules in general consist of several components. For instance, deductive rules like in Prolog/Datalog consist of a *rule head* and a *rule body*; similar e.g. for F-Logic or Transaction Logic rules. These languages are *rule-based languages* – their head and body are both expressions in this language (but note that e.g. negation may in most cases only occur in rule bodies). In the ECA paradigm, rules are not homogeneous: they consist of an *event component*, a *condition component* (that together roughly correspond to the "body" since they describe the situations where the rule is applicable), and an *action component* (that roughly corresponds to the "head") – and all these components use different languages. For example, for database triggers, events are database events like "on update" (that are raised by update operations), where-clause conditions are expressed in SQL, and actions are SQL programs or updates (that again may raise events). Another example is the language XChange discussed in [15], where the event component consists of an event query, the condition component is a test by evaluating an Xcerpt [16] query, and the action component may contain update actions, transactions, or raising of an event. Yet another example is ruleCore [8], where the (different) languages for each component are described. The ECA languages mentioned in the introduction ([12, 11, 4, 52]) also use different languages in each of the components. In these cases, the event, condition, and action language are usually closely related, but this is not necessarily always the case. The more complex a scenario is, the more specialized are the used component languages.
Thus, the semantics of a rule is determined by two constituents:

- the rule semantics or "rule paradigm" that characterizes the interplay between the components, and the
- language(s) used in its components.

E.g., *deductive rules* have several common semantics, either top-down, bottom-up as fixpoints, or well-founded or stable semantics, independent what underlying language (first-order logic, F-Logic,

Transaction Logic etc.) is used. In the same way, ECA rules have a fixed semantics, independent what languages are used in the E, C, and A components. An important common feature here is that the communication both for derivation rules and for ECA rules is done by *logical variables*; we will discuss this in detail in Chapter A.3.



Constraints:
RuleParadigm determines number of RuleComponents
RuleComponents ordered or named; using appropriate languages

Figure A.2.5: Rules, Rule Components and Languages

An XML markup of rules must cover the *structure* of rules and rule component languages. Here, the RuleML language [57] provides *general* guidelines that must then be specialized for each paradigm.

In the following, we will investigate general ECA rules. The analysis of the languages will be continued in two aspects: semantics/ontology and syntax (i.e., algebraic, variables etc.).

A.2.3.2 Components and Languages of ECA Rules

In usual Active Databases in the 1990s, an ECA language consisted of an event language, a condition language, and an action language. For use in the Semantic Web, the ECA concept needs to be more flexible and adapted to the "global" environment of a world-wide living organism where nodes "speaking different languages" should be able to interoperate. So, different "local" languages, for expressing events, queries and conditions, and actions have to be integrated in a common framework.

The target of the development and definition of languages for (ECA) rules and their components should be a semantic approach, i.e., based on an (extendible) ontology for these notions that allows for *interoperability* and also turns the instances of these concepts into objects of the Semantic Web itself. The upper level of this ontology is shown as an UML diagram in Figure A.2.7, which will be explained below.

In contrast to previous ECA languages from the database area, we aim at a more succinct, conceptual separation between the event, condition, and action components, which are (i) possibly given in separate languages, and (ii) possibly evaluated/executed in different places. Each of the components is described in an appropriate language, and ECA rules can use and combine such languages flexibly.

Analysis of Rule Components. A basic form of ECA/active rules are the well-known *database triggers*, e.g., as already shown above, in SQL, of the form

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

For them, *condition* can only use very restricted information about the immediate *database update*. In case that an action should only be executed under certain conditions which involve a (local) database query, this is done in a procedural way in the *pl/sql-fragment*. This has the drawback of not being declarative: reasoning about the actual effects would require to analyze the program

code of the *pl/sql-fragment*. Additionally, in the distributed environment of the Web, the query is probably (i) not local, and (ii) heterogeneous in the language – queries against different nodes may be expressed in different languages. For our general framework, we prefer a *declarative* approach with a *clean, declarative* design as a "Normal Form": Detecting just the dynamic part of a situation (event), then check *if* something has to be done by probably obtaining additional information by a query and then evaluating a *boolean* test, and, if "yes", then actually *do* something – as shown in Figure A.2.6.

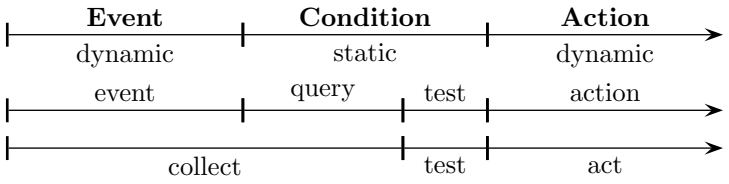


Figure A.2.6: Components and Phases of Evaluating an ECA Rule

With this further separation of tasks, we obtain the following structure:

- every rule uses an event language, one or more query languages, a test language, and one or more action languages for the respective components (for that, we allow several action components in different languages that have *all* to be executed,
- each of these languages and their constructs are described by metadata and an ontology, e.g., associating them with a processor,
- there is a well-defined *interface* for communication between the E, Q&T, and A components by variables (e.g., bound to XML or RDF fragments).

This model can be readily extended by adding a fifth optional component – the post-condition (another *test*) – resulting in a variation usually called ECAP rules. In most cases, this post-condition can be omitted by allowing the action language to test for conditions inside the action component. But it may have particular relevance when considered together with transactional rules, and for reasoning about the effects of sets of rules.

For applying such rules in the Semantic Web, a uniform handling of the event, query, test, and action sublanguages is required. For this, rules, their components, and the languages must be objects of the Semantic Web, i.e., described in XML or RDF/OWL in a generic *rule ontology* that contains all required information as shown in the UML model in Figure A.2.7.

A.2.3.3 Markup Proposal: ECA-ML

The model is accompanied by an XML ECA rule (markup) language, ECA-ML. The relationship between the rule components and languages is provided by identifying the languages with name-spaces (from the RDF point of view: resources), which in turn are associated with information about the specific language (e.g, an XML Schema, an ontology of its constructs, a URL where an interpreter is available). The latter issues are discussed in Section A.6; here we investigate the languages and the markup itself.

For an XML representation of ECA rules as shown in Figure A.2.7, we propose the following (basic) markup (ECA-ML):

```
<!ELEMENT rule (%variable-decl,event,query*,test?,action+)>
  <!-- %variable-decl is not yet specified in detail here-->

<eca:rule rule-specific attributes>
  rule-specific content, e.g., declaration of logical variables
  <eca:event identification of the language >
```

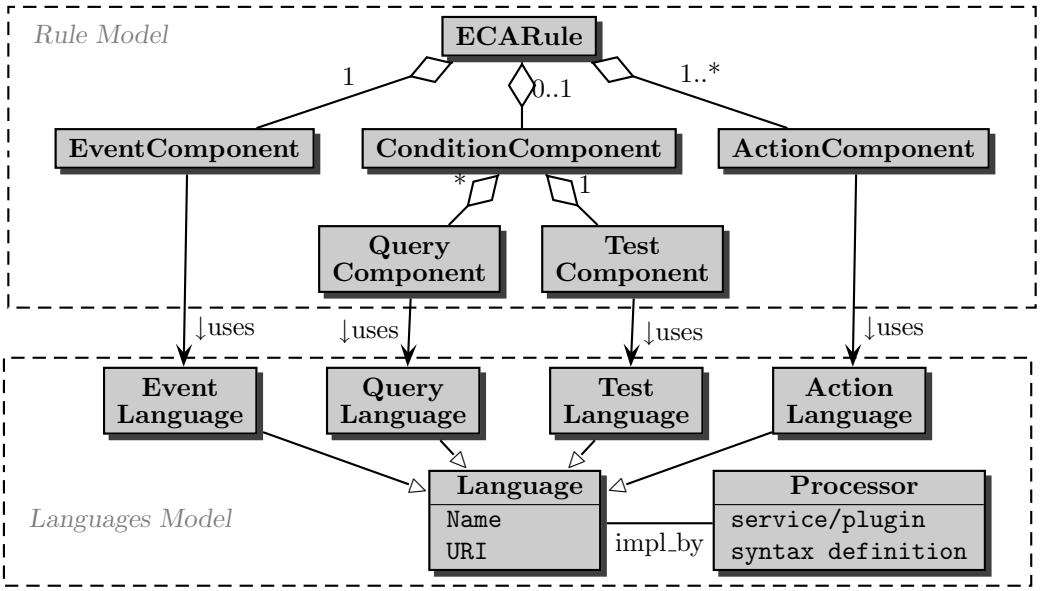


Figure A.2.7: ECA Rule Components and Corresponding Languages

event specification, probably binding variables; see Section A.3.4.1
</**eca:event**>
<**eca:query** *identification of the language* > <!-- there may be several queries -->
query specification; using variables, binding others; see Section A.3.5
</**eca:query**>
<**eca:test** *identification of the language* >
condition specification, using variables; see Section A.3.5.1
</**eca:test**>
<**eca:action** *identification of the language* > <!-- there may be several actions -->
action specification, using variables, probably binding local ones; see Section A.3.5.3
</**eca:action**>
</**eca:rule**>

The actual languages of the components (and on deeper nested levels) are usually identified by namespaces, sometimes also by explicit language attributes.

A similar markup for ECA rules (without separating the query and test components) has been used in [12] with *fixed* languages (a basic language for atomic events on XML data, XQuery as query+test language and SOAP in the action component). This fixed approach falls short wrt. the language heterogeneity, and especially the use and integration of languages for composite events. The same structure is also followed by the XChange transaction rules above, there again without any intention to deal with heterogeneity of languages: fixed languages are used for specifying the event, condition (without separating query/test), and action component. In contrast, here we generalise the approach to allow for using arbitrary languages. Thus, these other proposals are just possible configurations. Our approach even allows to mix components of both these proposals.

A.2.3.4 Hierarchical Structure of Languages

The approach defines a hierarchical structure of language families (wrt. the embedding of language expressions) which relates the different kinds of ontologies (application-dependent and application-independent) and their components as already described above in Section A.2.1.3 and Figure A.2.1 to the languages of rules and rule components as shown in Figure A.2.8 [here directly associating ontologies with the corresponding languages over the same alphabet]: As described until now, there

is an ECA language (that is already described by the above markup), and there are (heterogeneous) event, query, test, and action languages. Rules will combine one (or more) language of each of the families. In general, each such language consists of its own, application-independent syntax and semantics (e.g., event algebras, query languages, boolean tests, process algebras or programming languages) that is then applied to a domain (e.g. traveling, banking, universities, etc.). The domain ontologies define the static and dynamic notions of the application domain, i.e., predicates or literals (for queries and conditions), and events and actions (e.g. events of train schedule changes, actions of reserving tickets, ...). Additionally, there are domain-independent languages that provide primitives (with arguments), like general communication, e.g. received_message(M) (where M in turn contains domain-specific content).

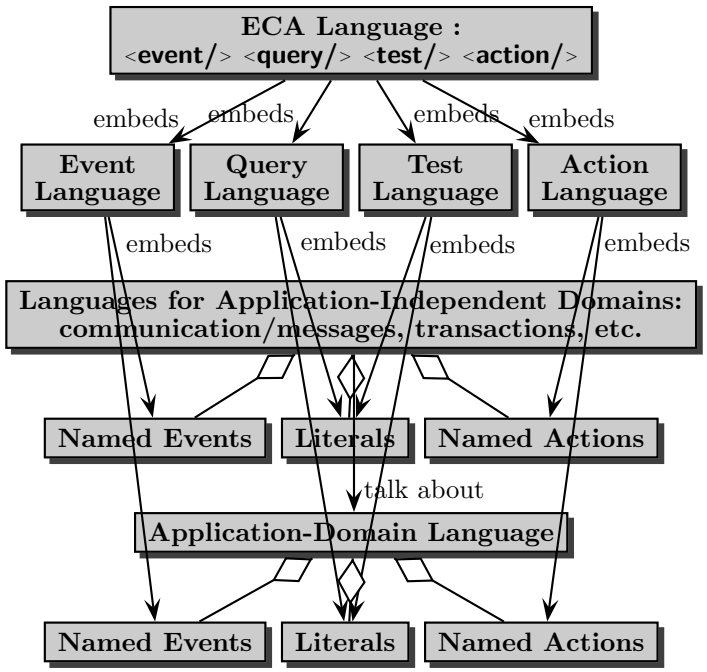


Figure A.2.8: Hierarchy of Languages

In the next section, we discuss common aspects of the languages on the "middle" level (that immediately lead to the tree-style markup of the respective components – thus, here the XML markup is straightforward). The semantics of rule execution and communication issues between the rule components are discussed in Chapter A.3. Samples of component languages will be discussed in Chapter A.4; a short account on domain languages including events and actions has been given in Section A.2.1.3.

A.2.3.5 Common Structure and Aspects of E, C, T and A Sublanguages

The four types of rule components use corresponding types of languages that share the same algebraic language structure, although dealing with different notions:

- event languages: every expression gives a description of a (possibly composite) event. Expressions are built by composers of an event algebra, and the leaves here are atomic events of the underlying application domain or an application-independent domain;
- query languages: expressions of an algebraic query language, embedding literals from the domains;

- test languages: they are in fact formulas of some logic over literals of that logic in the underlying domains (that determine the predicate and function symbols, or class symbols etc., depending on the logic);

- action languages: every expression describes an action. Here, algebraic languages (like process algebras) or "classical" programming languages (that nevertheless consist of expressions) can be used. Again, the atomic items are actions of the underlying domains.

Algebraic Languages.

As shown in Figure A.2.9, all components have in common that the component languages consist of an algebraic language defining a set of *composers*, and embedding *atomic* elements (events, literals, actions) that are contributed by the *domain languages*. Expressions of the language are then (i) atomic expressions, or (ii) composite expressions recursively obtained by applying composers to expressions. Due to their structure, these languages are called *algebraic languages*, e.g. used in *event algebras*, *algebraic query languages*, and *process algebras*. Each composer has a given *cardinality* that denotes the number of expressions (of the same type of language, e.g., events) it can compose, and (optionally) a sequence of parameters (that come from another ontology, e.g., time intervals) that determines its *arity* (see Figures A.2.9 and A.2.10).

For instance, "E_1 followed_by E_2 within t" is a binary composer to recognize the occurrence of two events (atomic or not) in a particular order within a time interval, where t is a parameter. Event languages define different sets of composers, such as XChange for its composite event queries [15], the ruleCore detectors [8], or the SNOOP event algebra of [17]. Similar composers are used in process algebras, or also –but in general syntactically covered– in algebra-based query languages. The boolean algebra with its composers is well-known.

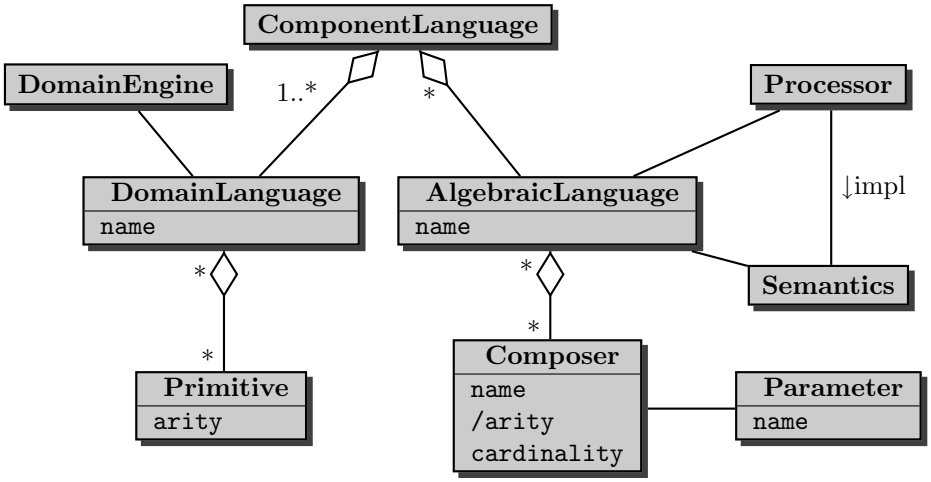


Figure A.2.9: Notions of an Algebraic Language

Semantics of Algebraic Languages.

Every algebra expression is assigned a semantics, i.e., from evaluating it (in case of queries: in a given state). Starting with the semantics of atomic expressions, the semantics of composite expressions is determined by the composer. The semantics of the different types of algebraic languages are as follows:

- event languages: in most cases, the event instance(or sequence) that matches the given expression pattern,

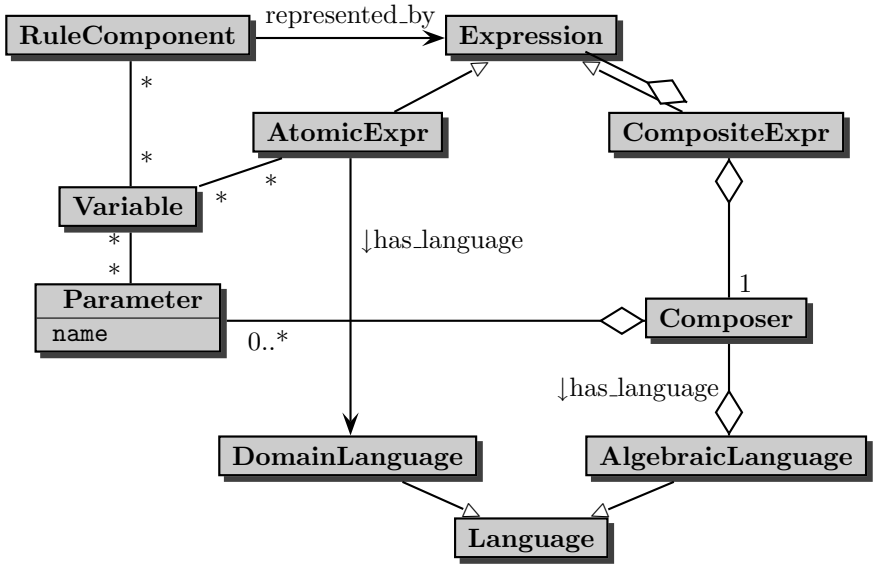


Figure A.2.10: Syntactical Structure of Expressions of an Algebraic Language

- query languages: a query result, e.g., a relation or an XML structure,
- test languages: a truth value of a logic (i.e., for classical logics, true or false)
- action languages: the formal semantics of terms of e.g. process algebras are denotational or operational semantics as state transformers; nevertheless, here we are only interested in their side effects on the underlying data.

Composition of Algebraic Languages.

For each type of such algebraic languages (i.e., event, query, test, and action languages), the expressions define and combine entities of the same kind, i.e., again, events, queries, tests, or actions, possibly with appropriate parameters and logical variables (see below) and using events, literals, or actions from the respective part of the domain language. Thus, from the ontology point of view, entities of the same kind described in different languages can be combined (e.g., a conjunction of a (sub)formula in first-order logic with one in description logic, or a sequence (or, more generic, any binary composer of any event language) of an event specified in language EL_1 and one specified in EL_2).

The global language concepts and the markup will support this, and we will also explain how evaluation also smoothly crosses these language borders. Note that Figure A.2.10 does not associate the whole rule components with a language, but each *expression* is associated to a language.

Tree Markup of Algebraic Languages.

Thus, language expressions are in fact trees which are marked up accordingly. The markup elements are provided by the definition of the individual languages, "residing" in and distinguished by the appropriate namespaces. As described above, it is also possible to nest composers and expressions from different languages of the same kind, distinguishing them in the markup by the namespaces they use. Thus, languages are in fact not only associated once on the *rule component* level, but this can also be done on the *expression* level.

Note that leaves of expressions can be either *atomic* expressions (events, predicates, or atomic expressions of whatever query language, atomic actions, etc.) that are defined in domain languages.

A.2.3.6 Language Information

As stated above, languages are associated on the *expression* level. We first investigate this from the point of view of the ECA engine (which is the first module to be developed), where the expressions of interest are the rule components. Here, the functionality of selecting services according to the language information is located (having nested subexpressions in different languages requires to have this functionality also in the processors of component languages, e.g., for having a subevent specification in a different event algebra).

As shown in Figure A.2.10, every expression (i.e., the rule components and their subexpressions) can be associated with a language: an expression is either

- an atomic one (atomic event, literal, action) that belongs to a domain language (either application-dependent or application-independent), or
- a composite expression that consists of a composer (that belongs to a language) and several subexpressions (where each recursively also belongs to a language – in many cases, the same as the composer), or

Language Binding for Components and Expressions. The language binding is made explicit by the namespace that is used in the root node of an expression (and declared there or in one of its ancestor elements; e.g. in the <eca:rule> element). The namespace declaration always yields a URI.
The markup of an ECA rule with language information has the following form:

```
<eca:rule>
  <eca:event xmlns:evns="ev-lang-uri">
    <evns:element-name> ... </evns:element-name>
  </eca:event>
  <eca:query xmlns:qns="q-lang-uri">
    <qns:element-name> ... </qns:element-name>
  </eca:query>
    :
</eca:rule>
```

Here, *ev-lang-uri* and *q-lang-uri* are URIs associated with the namespaces of an event language and a query language.

The meaning of the URIs will be discussed in Chapter A.6 (since there is not yet a standardization what is "behind" the namespace URI, we propose an intermediate solution that is sufficient for the infrastructure in our approach). Each of these languages (i.e., their URIs) has an associated engine that captures the semantics of the (composers of its) language. The engines provide the (expected) interfaces for communication, must keep their own state information, including at least the current variable bindings. Specific tasks of the engines then include e.g. the evaluation of composite events (for the event languages), or the execution of transactions (for the action engines). Thus, the framework itself does not have to deal with actual event detection or transaction execution, but only with employing suitable services (provided by the "owners" of these sublanguages) on the Web.

The leaves of the markup trees are then atomic events, literals, or actions, contributed by the underlying domains (and residing in the domain's ontology and namespace).

Special markup elements are provided for using and binding *variables* inside the expressions and on the rule level (e.g., results of the event detection or of functional queries); see Section A.3.3.1.

A.2.3.7 Opaque Rules and Opaque Components

Rules and components can also be given in an "opaque" form. This means that are given as program code in some already existing language and have an operational semantics as an ECA rule or component. Opaque rules and components include e.g. the following:

- SQL Triggers (opaque rules)
- SQL/SQLX, XPath or XQuery queries
- matching regular expressions
- program code for actions
- Web Service calls via HTTP

Especially during the development of a prototype, such functionality will be used. Wrappers are employed to integrate them into the framework (cf. Section A.7.1.2).

A.2.3.7.1 Opaque Rules

There are the existing trigger-style languages that handle specific, simple database events, simple conditions and actions, with their own syntax as discussed in Section A.2.2 above. Since these triggers work on the logical level and are in general (very efficiently) implemented based directly on the physical database level, they are not necessarily marked up in ECA-ML. Often, they are even not subject to the "Semantic Web", since they are just used to locally *implement* something that is *specified* in a completely different way (e.g., integrity constraints), or for raising RDF-level events based on an SQL or XML storage. In our ontology, we embed this as *opaque* rules as shown in Figure A.2.11.

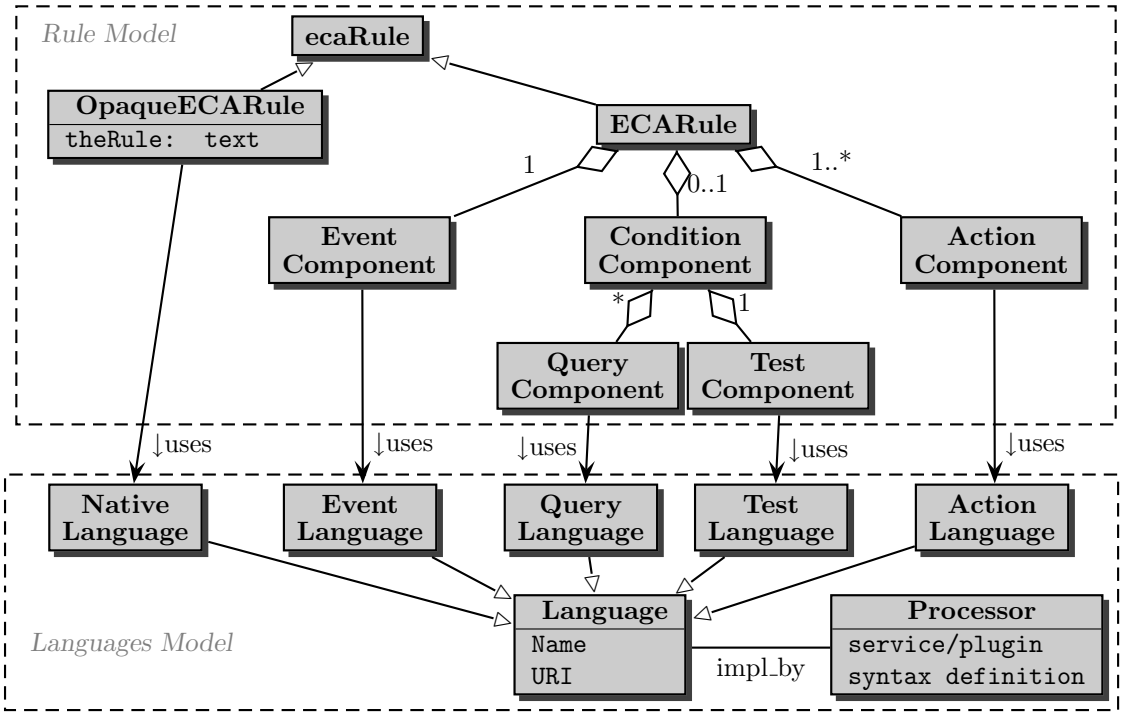


Figure A.2.11: ECA Rule Components and Corresponding Languages II

The ECA-ML language integrates opaque rules by the following markup: An <eca:opaque> element with text content (program code of some rule language) and an attribute language that indicates the name of a language or the URI where an interpreter is found, similar to the namespace. Names are can also be allowed for language if is designed similar to XML's NOTATION concept that associates resources with a program URI.

```
<eca:rule>
  <eca:opaque language="uri of the trigger language">
    ON database-update WHEN condition BEGIN action END
  </eca:opaque>
</eca:rule>
```

Since opaque rules are ontologically "atomic" objects, their event, condition, and action components cannot be accessed by Semantic Web concepts.

Note that there are canonic mappings between such triggers and their components and the general ECA ontology.

A.2.3.7.2 Opaque Components

Analogous to *opaque* rules, we allow for opaque rule components. An opaque component consists of a code fragment (not in XML markup) of some event/query/logic/action language, e.g., conditions that are not expressed in a markup language, but in XQuery, or actions expressed in Java or Perl. Opaque components extend the possible syntax of expressions as shown in Figure A.2.12. Opaque components play an important role especially during the development of the framework when not yet any dedicated sublangages are available. In such cases, again <eca:opaque> elements are used that refer to a URI for that language.

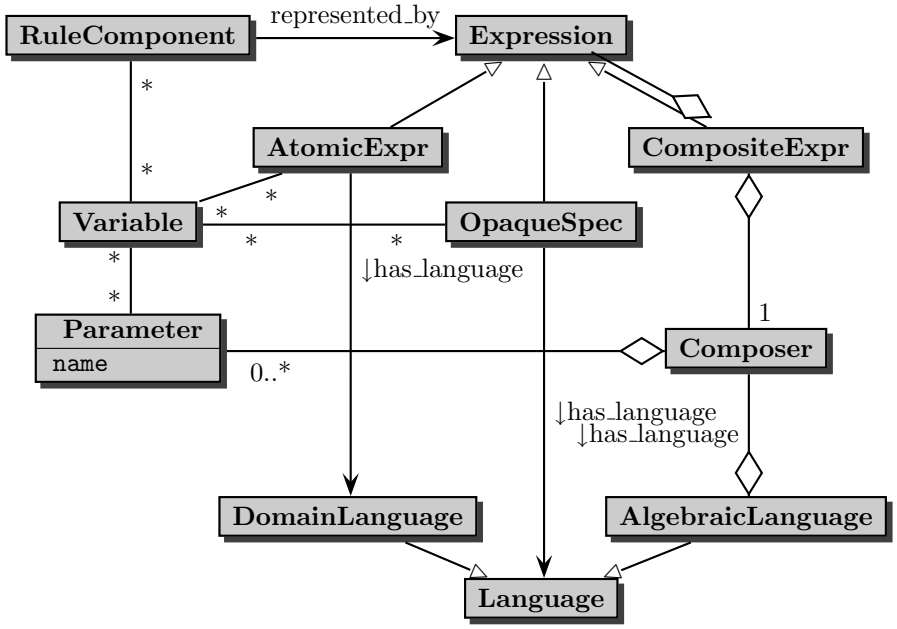


Figure A.2.12: Syntactical Structure of Expressions (Algebraic and Opaque)

Opaque fragments are marked up as shown below and do no reside in a namespace:

```
<eca:{query|condition|test|action}>
  <eca:opaque attributes>
    opaque code
  </eca:opaque>
</eca:{query|condition|test|action}>
```

For being able to execute such opaque fragments, the following cases must be considered (note that the examples do not deal with the handling of the result; this will be discussed in the next section):

- Opaque component language expressions against generic (language) services: *opaque code* is e.g. an XQuery query against the Web or a Web source given as document("...") that can be executed by any XQuery service (e.g., based on saxon).
  In this case, the eca:opaque element has a language attribute whose value may either be a URI, or a namespace prefix that is declared before (and must be resolvd to to URI by the respective application).

```
<eca:query>
  <eca:opaque language="http://www.w3.org/XQuery">
    for $c in document("http://dbis.uni-goe.de/mondial/mondial.xml")//country
      return $c/name
  </eca:opaque>
</eca:query>
```

  Note that an important instance of such a generic service is a *local* service for evaluating XPath expressions against (small) instances bound to variables (e.g., for analyzing XML fragments returned by event or query components; see Section A.3.3.1).

- Specific Domain Services: e.g., XML repositories, SQL databases, or Web Services can be queried by HTTP GET. In all cases, there is a URL to which a certain query is submitted. In this case, the eca:opaque element has a uri attribute that indicates the service. Additional attributes can be used to give further specifications.

  For HTTP GET there are two possible syntaxes:

  – append the query part to the base URI and having an empty content:

```
<eca:query>
  <eca:opaque uri="http://exist.dbis.uni-goe.de/exist/servlet/db/mondial.xml
                   ?_query=//country[name='Germany']&_wrap=no"
    method="get"/>
</eca:query>
```

  – having the base URI as the uri attribute, and the local part as content:

```
<eca:query>
  <eca:opaque uri="http://exist.dbis.uni-goe.de/exist/servlet/db/mondial.xml">
    method="get">
    ?_query=//country[name='Germany']&_wrap=no
  </eca:opaque>
</eca:query>
```

    The latter has the advantage that it separates the service part from the query part:

    * having the service part separate allows to use information about the service,
    * the query part can contain variables whose values must be inserted,
    * and it is easier accessible to analysis and reasoning.

    This holds especially when the service is not a database query service, but a form-like "lookup" Web service:

```
<eca:query>
  <eca:opaque uri="http://simple-service.dbis.uni-goe.de" method="get">
    ?name='Germany'
  </eca:opaque>
</eca:query>
```

  Discussion (including definitions of Section A.3.1.5):

  - when input variables are declared, the GET syntax can be derived when the URI is known: $uri?var\_name_1=value_1\&\ldots\&var\_name_n=value_n$
  - in case that a service with multiple "methods" is used, the service should be given in the URI part (since the service description according to Chapter A.6 is associated with it) with this part, all other stuff in the contents.

- More advanced HTTP methods (POST) and HTTP-based protocols (like SOAP) that require pre-processing before sending the request are also indicated as attributes of the eca:opaque element. The content of the opaque element is then sent as message content; possibly modified for communication variable bindings; cf. Section A.3.2.2).

```
<eca:query>
  <eca:opaque uri="http://soap-service.dbis.uni-goe.de/xpath-servlet">
    for $c in doc("/db/mondial.xml")//country
    return $c/name
  </eca:opaque>
</eca:query>
```

  In case that service descriptions according to Section A.6.1.2 are used, it should *not* be necessary to indicate the used communication protocol explicitly.

If a component is given as opaque code, the communication is chosen according to the language indicators:

service URI	protocol	method	used protocol
http://...	–	–	native (framework)
http://...	–	get/post	http with indicated method
ip-address	tcp	–	tcp
ip-address	–	–	native (framework)

Note that it is possible and useful to use opaque code with framework-aware services: This is the exact characterization of wrappers around non-framework-aware services and provides a good means for rapid prototyping of an ECA engine.

Note that so far we did not deal with submitting values of variables to the services (either as values for comparison in predicates, or as "documents" that have to be queried); this will be discussed in Sections A.3.1.2, A.3.2.2, and A.3.2.3.

Chapter (Appendix A: ECA Framework) A.3

Abstract Semantics: Rule Level

This chapter develops the framework from the point of view of the rule level. For this, a rough intuitive idea what the event, query, test and action components do is sufficient. They will be considered in more detail in Chapter A.4.

A.3.1 Abstract Declarative Semantics of Rule Execution

This section deals with the overall semantics of ECA rules and the abstract semantics for each of the components, and with the actual communication. Although the languages are heterogeneous wrt. the components, there is a common requirement: to support language heterogeneity at the rule component level, there must be a precise convention between all languages how the different components of a rule can exchange information and interact with each other.
There are two kinds of communication in the rules:

- Horizontal communication, following the data flow in the rule according to Figure A.2.6 from the event component to the query component and so on.

  For this, we propose to use *logical variables* in the same way as in Logic Programming (cf. Section A.3.1.2).

- Vertical communication between the ECA engine and the engines that are responsible for processing the component languages.

  We align this way of communication with the concepts used for logical variables by regarding every component as a mapping that takes as input a set of tuples of variable bindings and returns another set of tuples of variable bindings.

We introduce the notion of variable bindings that are used for both kinds of communication and provide a logic-based declarative semantics on the rule level and for data exchange with component services.

A.3.1.1 Rule Semantics

Comparison: Firing Deductive Rules. For deductive rules (that do not have an event component) in *bottom-up* evaluation, the body is evaluated and produces a set of tuples of variable bindings (Datalog, F-Logic, Transaction Logic, Statelog, XML-QL, XPathLog [42], Xcerpt [16]; in some sense also the basic form of XQuery). Then, the rule head is "executed" by *iterating* over all bindings, for *each* binding instantiating the structure described in the head (in some languages also executing actions in the head).

The semantics of ECA rules should be as close as possible to this semantics, adapted to the temporal aspect of an event:

ON event AND additional knowledge, IF condition then DO something.

We transfer the concept of variable bindings to ECA rules in the following examples:

Example 3 (Cancelled Flights) *Consider a rule that should do the following: Whenever a flight is canceled, send a message to the destination airport that the flight will not take place.*

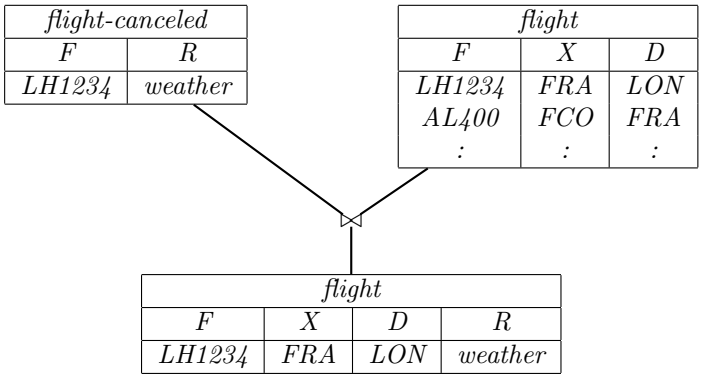
Assume that events are of the following form

<travel:canceled-flight code="LH1234" reason="weather" />

indicating the code F of the canceled flight and the reason R of the cancellation. Then, a query against a database is evaluated that yields the destination airport D. Finally, a mail is sent to the target airport D, telling that and why the flight F is canceled.
A logic programming-style rule (using a relation flight(flight_no, from, to)) would look like this:

travel:canceled-flight(F, R) $\wedge$ flight(F, _, D) $\rightarrow$ send_mail(D,F,"canceled",R).

In LP terminology, the event component travel:canceled-flight(F, R) binds *variables F and R. The query component flight(F, _, D) alone binds variables F and D. Both results are joined and the result is then propagated to the action component (which corresponds to the rule head in LP) where they are then used.*
Formally, the rule semantics is join-based:



An alternative, algebraically equivalent, evaluation strategy (which corresponds to evaluating a join based on an index) is to use the (only) binding of F from the event part to lookup *the destination and to just to* extend *the variable bindings.*

A.3.1.2 Logical Variables

We propose to use *logical variables* in the same way as in Logic Programming that can be bound to several things: values/literals, references (URIs), XML or RDF fragments, or events. The resp-resentation of the bound items must be in ASCII, e.g., URIs, serialized XML, or XML-serialized RDF. The binding of a variable to an event (or a sequence of events) e.g. occurs in the SNOOP language [17] in *cumulative aperiodic events*; such variables can then be used for extracting values from these events. Variables can be bound by the rule (as constants upon registration) or by the components and used in later components.

For logical variables used in LP rules, there are several definitions that carry over to EQTA rules:

- Similar to deductive rules, variables used for communication occur *free* in the components, their scope is the rule,

- While in deductive rules, variables must be bound by a positive literal in the body[1] to serve as join variables in the body (adhering to safety requirements!) and to be used in the head, in ECA rules we have four components that induce an information flow according to Figures A.2.6 and A.2.7; see Section A.3.1 for details of the execution semantics.
- Positive occurrences are defined analogously to deductive rules based on the term/formula structure (must be done with the semantics of each individual such language).
- Positive occurrences can be used to bind variables to a value. In case that a variable occurs positively several times, it acts then as a join variable, i.e., the values must coincide; this e.g. allows for an event component that in some cases binds a variable which is then used as a join variable in the condition, and in other cases is only bound by the latter.
- Negative occurrences of a variable *use* the value the variable has been bound *before*.
- Thus, during execution of a rule, any variable occurring negatively must be bound to a value earlier on the rule level (e.g., with the rule's initialization, or by deriving its value from another variable) or in an "earlier" (E<Q<T<A) or at least "earlier" in the same component as where the negative occurrence is. This leads to the usual definition of safety of rules.
- Expressions can also use local variables, e.g., in first-order logic conditions. In this case, the scope of a variable is local, e.g., by a quantifier.
- Variables in the action component: Using variables as parameters to an action in the action component counts as negative occurrences (in case that a language used in the action component does not define positive or negative occurrences). Thus, for languages used in the action component hat do not define the notions of positive/negative occurrences, all occurrences are negative. Note that this allows for binding a variable in the action component, e.g., by allowing for evaluation of queries in that component, like in Transaction Logic [13] that defines its own notions of positive occurrences.

The relationships between rules, rule components and variables on this abstract level are shown in Figure A.3.1:

- for each rule R, $R.\mathsf{scopes}$ (the set of all logical variables whose scope is the rule) is the union of $R.\mathsf{occurs\_positive}$ (before or between evaluating components) and $C.\mathsf{free}$ for all its components C;
- for each component C of a rule R, and also for each expression inside any component, $C.\mathsf{positive}$, $C.\mathsf{negative}$, $C.\mathsf{free}$, $C.\mathsf{bound}$, denote the sets of variables that occur positively, negatively, free or bound.

  Free variables occur either positively or negatively:
  $C.\mathsf{free} = C.\mathsf{positive} \cup C.\mathsf{negative}$.

A.3.1.3 Horizontal Communication: Logical Semantics

So far, seeing components as abstract predicates yields a preliminary declarative semantics of ECA rules which is based on the LP join semantics. The *horizontal* communication during processing ECA rules is actually based on propagating sets of tuples of variable bindings. Several issues are dealt with in the subsequent sections:

[1] note that there are different terminologies in the literature about "positive" and "negative" literals: In Logic Programming, these notions are defined wrt. the rule body, whereas in works based on resolution of disjunctive clauses, they are defined wrt. those literals (as in Xcerpt/XChange). Since a (Horn) clause $p(x) \vee \neg q(x)$ corresponds to a LP rule $p(x) \leftarrow q(x)$, in the first case, $q(X)$ is a negative binding whereas in the second case, it is a positive binding.

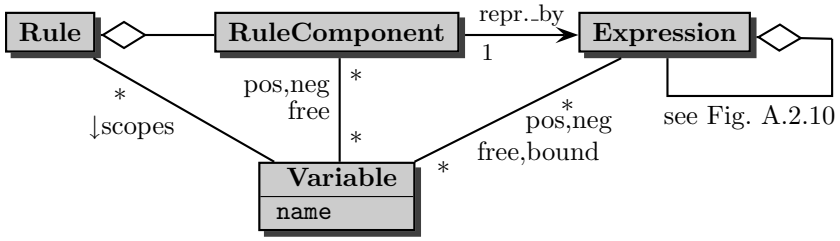


Figure A.3.1: Logical Rules and Variables

- replacing the abstract predicates by actual event, query, and action components that are specified by sublanguages and evaluated by appropriate services. This requires vertical communication;

- special semantics of events since they are not like usual predicates; expecially, the semantics of event algebras (event sequences, cumulative events) must be integrated accordingly;

- special semantics of Web Services since they use specific notions of input and output variables;

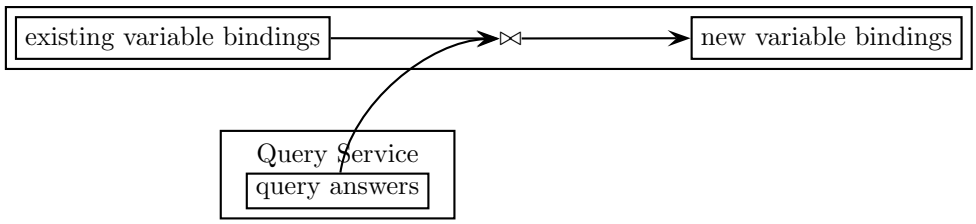
- appropriate adaptations of the notions of positive and negative occurrences of variables and their influence on the evaluation of joins;

- procedural aspects like event detection and at least basic transactional functionality.

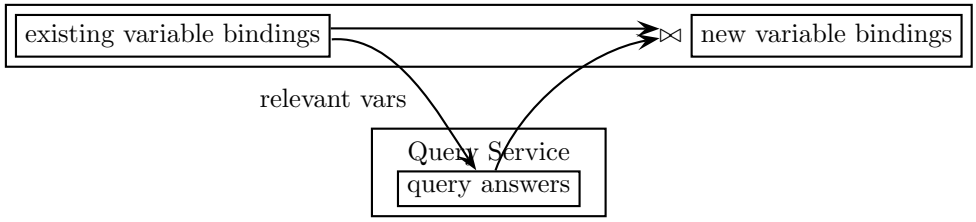
A.3.1.4 Vertical Communication

In the same way, the communication with the engines that process the sublanguages is done by sets of tuples of variable bindings. The abstract predicates are now replaced by the actual evaluation. Here, the interplay between the variable bindings on the ECA level and the component services and their results can be designed in different ways:

- Join-Style: at a given state of evaluation, the next query is evaluated independent from the already obtained variable bindings, and the result is joined with the already existing variable bindings. Here, the functionality of the ECA engine consists of invoking a service and joining results:



- Similar to *sideways information passing strategies* in algebraic evaluation that use variables that are already bound as constraint when evaluating the next predicate or subquery for minimizing results as soon as possible (cf. internal evaluation of Florid), existing variable bindings can be communicated downwards when calling a component language engine. In this case, the join semantics is realized as a restriction in the component language engine:



- The actual evaluation is a mixture for several reasons:
  - only actual Datalog services would implement the pure LP-style strategy;
  - negated variables must be bound before; here the join acts as a set difference,
  - variables that are not used at all do not need to be communicated downwards and upwards again (this also reduces the number of actual tuples of variable that have to be communicated);
  - some services (especially, many query languages like SQL and XQuery) show a functional behavior that does not bind variables at all;
  - functionality of the services (e.g. Web Services that require a single input value and return a single output value);

In the next sections, downward and upward communication are investigated.

A.3.1.5 Communication Modes and Declaration of Variables

Usually, languages based on logical variables do not use explicit variable declarations. Nevertheless, for controlling variable exchange, is must be known to the ECA engine which variable must or should be exchanged with the component services (especially during the development phase where opaque components and non-semantic services are employed). This also allows to extend the binding mechanism with a type system.

Consider the abstract structure of a rule over a set $x_1, \ldots, x_n$ of variables (without loss of generality, we consider only one query and one action component):

$$\mathsf{event}(x_{e_1}, \ldots, x_{e_{n_e}})\ \ \mathsf{query}(x_{q_1}, \ldots, x_{q_{n_q}})\ \ \mathsf{test}(x_{t_1}, \ldots, x_{t_{n_t}})\ \ \mathsf{action}(x_{a_1}, \ldots, x_{a_{n_a}})$$

where $1 \leq n_e, n_q, n_t, n_a \leq n$ and $1 \leq e_1, \ldots, e_{n_e} \leq n$ are pairwise disjoint, same for $q_1, \ldots, q_{n_q}$, $t_1, \ldots, t_{n_t}$, and $a_1, \ldots, a_{n_a}$.

From a logical point of view, these variables are the *free* ones in the event, query, test and action part. Actually, the components may have an operational semantics that uses some of the x_i as input, and generates some others as output or "result". From an abstract view, they all can be considered as predicates.
In the following, we will clarify four kinds of variables. Considering the following example query (event, test, and action components are specialized cases):

Example 4 (Variables) *Consider a car rental company that holds a database that holds for each location a list of all models with a classification and prices available. Seen as a* predicate*, the service has the following* extension *as a predicate* available*:*

answer("Frankfurt","Golf", "B", "40")
answer("Hannover","Golf", "B", "50")
answer("Hannover","Passat", "C", "120")
answer("Hannover","S500", "D", "250")
answer("Paris","C4", "B", "50")
answer("Paris","Golf", "B", "65")
answer("Paris","C6", "D", "150")
answer("Bucharest","Logan", "B", "25")
:

Consider now a travel agent service that has some personalized knowledge about its clients, e.g., what car they drive at home. The service has now a rule that states

> *"when a client books a flight to* place *with an airline* airline*, offer him a list of cars equivalent to his own to rent".*

The variable place *is bound by the "flight booking" event, the second relevant variable is* class *that can be bound by a query against a local car database. The third step is then to collect offers at the given place, which is the part under discussion now. Logically, the rule reads as*

```
offer(Model,Price) :-booking(Person,_From,To), owns(Person,OwnCar),
                     class(OwnCar,Class), available(To,Model,Class,Price).
```

Consider the owner of a VW Golf traveling to Paris, i.e., To*="Paris" and* OwnCar*="Golf" are known from the first line.*

- *Bottom-up evaluation: take the extensions of* class *and* available*, join them with the known values for* To *and* OwnCar *and return the result.*
- *Service (1): call the service for "Paris", look up the own car's class in a local database, and join the results,*
- *Service (2): look up the own car's class in a local database, call the service for ("Paris","B"), and return the results.*

The actual choice in the rule (i) depends on the functionality of the data source, and (ii) results in different costs of query evaluation and size of the transferred result. While (i) depends on the design (and the rule designer must adhere to it), (ii) can be dealt with by an optimizer [to be developed].
Consider now a service that requires the location as an input; giving the class is optional.

- *used variables: the service only uses the variables* To *and* Class*. The value of* Airline *is irrelevant.*
- *input variables:* To *is an input variable that* must *be given.*
- *result variables: variables that are only returned; if they are submitted to the service, they are ignored.*

As said above, it is then possible to call the service only by submitting the To *location and doing the join of the answer tuples with the known* Class *afterwards, or call the service with both* To *and* Class*, i.e, the service already applies the join condition. Probably, the second case would be more efficient.*

Consider an extension where a traveler who owns several cars is offered all models that are equivalent to some of his own ones. Now, for people owning several cars, the first case would be more efficient in most cases. Here, it would be useful to have a service that can be called with multiple tuples.

Thus, having an environment where variables are not only logical ones, but also are used as arguments and results, the augmented relationships between rules, rule components and variables are shown in Figures A.3.2 (adapting Figure A.3.1) and Figure A.3.3. The *abstract, declarative* semantics is still logical, only the *operational* semantics needs a more detailed look.

Used variables; free variables. In logical languages, these are the variables that occur free. In languages that have a term markup, they can be derived by analyzing the term structure (simply by //variable/@name). Inside an algebraic evaluation, it is usually preferable to communicate all used variables that are already bound to use them as constraints and to minimize results as soon as possible (cf. internal evaluation of Florid).

Input variables. In logical languages, these are the variables that occur free, but negatively, e.g., the z in $p(x, y) \wedge \neg q(y, z)$: when stating this query against any source, values for z must be provided to be *safe*.

In logic-based languages, they are defined inductively on the structure of the query. In the Semantic Web environment, they must either be indicated by the rule designer (based on knowledge about the services), or they must be distinguished by the service description.

Output variables. Output variables do not exist in logical languages. Nevertheless, in the Semantic Web, they do exist. Output variables are variables that are bound by a service, independent if they have been bound before or not. To provide join variable semantics, they *must* be considered by equi-joining the result with the values bound before.

Returned variables. When considering to join the answer set of a predicate with a set of already existing variable bindings, it must be taken into consideration whether an answer actually contains all variables that are relevant for a join:

- Datalog answer-style: a query, e.g., ?-capital('Germany',X) results only in the variable binding X/'Berlin' .
- When considering a given set of variable bindings as a constraint, a query, e.g.,

  ?-capital(C,X)[{(C/'Germany'),(C/'France')}]

  results in extended variable bindings

  {(C/'Germany',X/'Berlin'), (C/'France',X/'Paris')} .
- functional "lookup" services: the input variables are only communicated downwards, and only the result is communicated upwards. Such services are e.g. provided by HTTP forms:

  http://capitalsoftheworld.com?country='Germany'

  would result only in the value 'Berlin', which must then be bound to a variable. Such lookup services can also return frame-like data; e.g.

  http://countriesoftheworld.com?country='Germany'

  could return a structure like

  <code>D</code>
  <area>356910</area>
  <population>83536115</population>

  which could be interpreted as binding three variables.

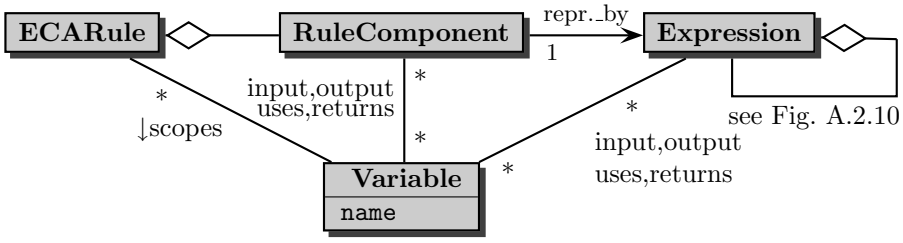


Figure A.3.2: Use of Variables in Components of ECA Rules

The evaluation of the event component (i.e., the successful detection of a (composite) event) binds variables to values that are then extended in the query component, possibly constrained in the test component, and propagated to the action component.

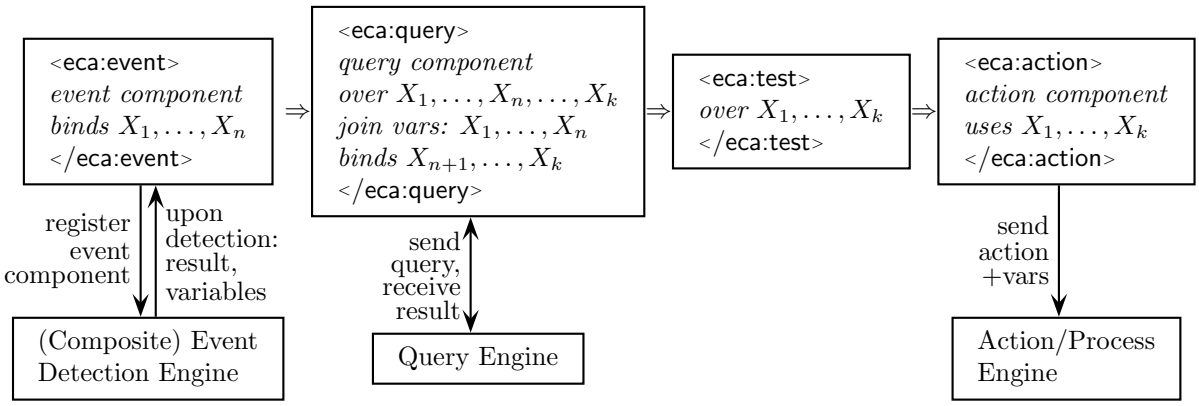


Figure A.3.3: Use of Variables in an ECA Rule

Variables and kinds of components. Depending on the type of the component (event, query, test, and action), the following kinds of variables can occur:

	input	used and either input nor output	output	returns
event	(X)	-	X	X
query	X	X	X	X
test	(neg)	(pos)	-	X
action	X	-	-	-

(X) means that this can be only variables bound to constants at the time of registering the rule. Since a test is a logical *condition*, the usual definitions of positive and negative occurrences of variables apply. The above table shows that the query part, by *obtaining* information and correlating it to already present information needs the most complex communication interface.

Declarations of Variable Use. For a rule component C (and an expression in general, since the usage of variable is defined recursively), used-vars(C), input-vars(C), output-vars(C) and returned-vars(C) denote the above sets of variables.

For that reason, at least when evaluating an ECA rule according to the specification that will be given in Section A.3.1, knowledge of the above-mentioned sets of variables is useful:

- Downward communication:
  - If for a component C, used-vars(C), input-vars(C) and output-vars(C) are known, check if a value for each of the input-vars(C) is contained in each tuple (otherwise, abort). Then, any set input-vars(C) $\subseteq V \subseteq$ used-vars(C) $-$ output-vars(C) is useful to be communicated. Note that used-vars(C) $- V$ must be considered afterwards in a join.
  - In case that the set used-vars(*component*) of variables that are used in a component is known (which, as mentioned can be done relatively simple if the component is marked up), it is safe (i) to communicate all existing bindings downwards *and* perform a join afterwards.
  - Otherwise, all existing bindings must be communicated downwards.
- Upward communication:
  - if used-vars(C) = returned-vars(C), then the result can be joined immediately.
  - if used-vars(C) $\supsetneq$ returned-vars(C), then the component must be evaluated separately for each bindings of used-vars(C) $-$ returned-vars(C) and the missing variabled must be taken into consideration.

– In reality, there are mainly two kinds of services:
  * Logic Programming semantics: outputvars(C) = ∅ and returned-vars(C) = used-vars(C), and
  * lookup services: outputvars(C) ≠ ∅ and returned-vars(C) = output-vars(C).

  Default assumptions: if no returned-vars, then the above cases are assumed.

The markup of components may thus *optionally* contain explicit declarations of the usage of variables.
However, the goal is that this information can be derived from the markup and the semantic information about the component languages. For this, every service that "offers" a language should provide the following functionality (see also Chapter A.6).

- Algebraic component languages: given an XML or RDF fragment of an instance of the language: Validation of the fragment, list of all variables that are used, and all variables that occur positively (i.e., can be bound by this fragment).
- For every Domain Language and Domain Web Service the input and result variables should be defined in a service description.

Explicit declarations are mainly important for handling opaque fragments and non-Semantic Web services. For the ECA prototype, such services are frequently used.

Example 5 (Rental Cars (Revisited)) *Consider again the car rental example from above. Since we focus on the handling of variables, we give the components as (atomic) Datalog predicates with variable communication mode declarations. Assume that the Datalog service is local and wrapped approriately with a framework-aware wrapper (see Section A.7.1.2.2).*

```
<eca:rule xmlns:eca="http://www.semwebtech.org/eca/2006/eca-ml"
    xmlns:xpath="http://www.w3.org/XPath"
    xmlns:pseudocode="http://www.pseudocode-actions.nop">
    <eca:event>
      <eca:opaque language="datalog-match" output-variables="Person To">
        booking(Person,_From,To)
      </eca:opaque>
    </eca:event>
    <eca:query>
      <eca:opaque language="datalog-match" used-variables="Person OwnCar">
        owns(Person,OwnCar)
      </eca:opaque>
    </eca:query>
    <eca:query>
      <eca:opaque language="datalog-match" used-variables="OwnCar Class">
        class(OwnCar,Class)
      </eca:opaque>
    </eca:query>
    <eca:query>
      <eca:opaque uri='http://localhost/lookup-cars' input-variables="To" output-variables="Class Model Price ">
        ?_place=To
      </eca:opaque>
    </eca:query>
    <eca:action>
      <eca:opaque uri='localhost' input-variables="Model Price">
        show all model-price tuples
      </eca:opaque>
    </eca:action>
</eca:rule>
```

A.3.2 Logical Variables: Markup for Communication

A.3.2.1 Basic Interchange of Variable Bindings

We propose the following representation for variable bindings that will be used for interchange (see also Section A.3.2.3):

```
<!ELEMENT variable-bindings (tuple+)>
<!ELEMENT tuple (variable+)>
<!ELEMENT variable ANY>
<!ATTLIST variable name CDATA #REQUIRED
          ref URI #IMPLIED>    <!-- variable has either ref or content-->
```

```
<logvars:variable-bindings>
  <logvars:tuple>
    <logvars:variable name="name" ref="URI" />
    <logvars:variable name="name">
      any value
    </logvars:variable>
      :
  </logvars:tuple>
  <logvars:tuple> ... </logvars:tuple>
  <logvars:tuple> ... </logvars:tuple>
     :
  <logvars:tuple> ... </logvars:tuple>
</logvars:variable-bindings>
```

Note that such data exchange is required not only for ECA rules, but for all kinds of services that are based on logical variables (rules, queries, reasoners). For this, we propose to use a separate namespace, here called logvars, referring to the URI http://www.semwebtech.org/lang/2006/logic. The complete logvars DTD can be found in Appendix A.10.

In the next sections, the exchange of variable bindings with the component services is discussed, extending this basic structure and markup.

A.3.2.2 Downward Communication: Variable Bindings

The actual handling of downward communication depends on the interface provided by the respective services.

Downward communication occurs in the following cases (note that the handling of the event component differs sigificantly from that of queries, tests, and actions:

- When a rule is *registered*, the event component is submitted to an event detection engine (that understands the event language). Optionally, variables that are already bound on the rule level (when handling *rule patterns*) can also be contained in the message. The message must contain the following information (see Section A.4.1 for the actual structure and syntax):

  1. administrative
     - where the answer should go, and
     - an identification to be used in the answer.
  2. contents:
     - the (event) component or a reference to it, and
     - optional: the current variable bindings (in the format described in Section A.3.1.2). Note that only those variables need to be communicated that are actually used in the component according to the above considerations.

For events, there is necessarily an asynchronous communication of requests and answers:

  - downward communication by register at *registration time*,
  - upward communication (see Section A.3.2.3) by logvars:answers or logvars:answer at *rule evaluation time* when the event is actually detected (see Section A.4.3).

- Queries, tests, and actions are submitted at rule evaluation time for certain variable bindings. The information contained in the message is similar to that for events.

  Note that it is also possible to *register* a query or an action with free variables at registration time, and to *request* answers or execution (for given variable bindings) at *rule evaluation time*.

A.3.2.3 Upward Communication: Results and Variable Bindings

Upward communication is concerned with the exchange of results and variable bindings, i.e., returning results and bound variables from evaluating an expression. For this, the structure and markup shown in Section A.3.2.1 is extended.

Upward Communication: (Functional) Results and (Logical) Variables. There are several possibilities what the "result" of evaluating a rule component can be:

- Logic-Programming-style languages that bind variables by matching free variables (e.g. query languages like Datalog, F-Logic [33], XPathLog [42]). Here, the matches can be *literals* (Datalog) or literals and structures (e.g., in F-Logic, XPathLog, Xcerpt). Similar techniques can also be applied to design languages for the event component.
- Functional-style languages that are designed as functions over a database or an event stream and a set of input/environment variables:
  - query languages that return a set of data items (e.g., SQL, OQL) that can be interpreted as producing a set of variable bindings (attribute names as variables; probably obtained by the "renaming" operator of the relational algebra),
  - query languages that return a data fragment (e.g. XQuery, Xcerpt [16] – this is only possible since "schema-free" data like XML exists). Here, the result is not bound to an obvious variable name. Note that this should result in a set/sequence of variable bindings if a set/sequence of nodes is created.
  - for event languages, the "result" of an expression can be considered the sequence of detected events that "matched" the event expression in an event stream (e.g., XChange).

  In this case, the languages can also *use* variables that are bound before. Thus, "downward" communication is explicit, whereas the upward communication is implicit (and the result must be bound to a variable by the *surrounding* language). Note that the answer can even be empty which *sometimes* is interpreted as "false".
- Both forms can be mixed (F-Logic, cumulative operators in event languages).

To cover all of them, we propose a structure as e.g. used in the Florid system [24] where with *each* result, a set of variable bindings is associated (cf. [38, Section 2.2.1]). We propose the following representation for interchange of results and variable bindings that extends the markup already used in the logvars namespace (the complete logvars DTD can be found in Appendix A.10).

```
<!ELEMENT answers (answer*)>
<!ELEMENT answer (result—variable-bindings—(result,variable-bindings))>
<!ELEMENT result ANY>
<!ELEMENT variable-bindings (tuple+)>
```

```
<!ELEMENT tuple (variable+)>
<!ELEMENT variable ANY>
<!ATTLIST variable name CDATA #REQUIRED
          ref URI #IMPLIED>    <!-- variable has either ref or content-->
<logvars:answers>
  <logvars:answer>
    <logvars:result>
      any result structure
    </logvars:result>
    <logvars:variable-bindings>
      <logvars:tuple>
        <logvars:variable name="name" ref="URI" />
        <logvars:variable name="name">
          any value
        </logvars:variable>
          :
      </logvars:tuple>
    </logvars:variable-bindings>
  </logvars:answer>
  <logvars:answer>
    :
  </logvars:answer>
</logvars:answers>
```

A set of answers consists of multiple answers, where each answer consists of a result value and/or a set of tuples of variable bindings. A variable binding can either be given inline as serialized XML, or as a URI reference (e.g., to a Web page, or an RDF URI). The answers can optionally contain a reference to an ID that in case of asynchrounous communication indicate to what they are an answer (this is needed for event detection; query handling can be synchronous or asynchronous). Note the following:

- In cases where only one single answer is produced (which is often the case for event detection, or when calling a "functional" Web Service), the outer <logvars:answers> may be omitted, returning only one <logvars:answer> structure.
- for services that return no functional result (e.g., a Datalog query service) or no variable bindings (e.g., an XQuery service), each <logvars:answer> structure contains only the relevant subelement.
- for services that return only a single functional result (an event sequence, or an answer to an XQuery or SQLX query), it is allowed not to mark it up at all. It is then treated as <result> element of a single answer and can be bound to an ECA-level variable as described below.

Upward communication occurs in the following cases (see Section A.4.1 for the actual structure and syntax):

- Event detection: an event has been detected. Usually, the result consists of the sequence of relevant events (as functional result) and variable bindings. The message must contain the following information:
  1. administrative: the identification of the event that has been detected
  2. contents: the result (in the format described above for <answers>).
- Query answering. Here the result also consists of the a set of answers as above. There is the possibility that a query answer is sent in several parts.

1. administrative: the identification of the query that is answered, and whether the answer is complete (default: yes);
2. contents: the result (in the format described above for <answers>).

A.3.3 Markup: Binding and Using Variables

Variables can be bound to many kinds of data: XML nodes like elements and attribute nodes, text contents, strings, numbers, and –in an RDF setting– also RDF URIs. Thus, the *mechanisms* for dealing with variables must be generic. On the other hand, since variable binding and using happens inside the scope of individual languages, also the in general depends on the individual language.

The framework only defines how <ecans:variable> can be used on the ECA level (and immediately below it), and in the next paragraphs we discuss some design alternatives how to integrate the handling of variables in to component languages.

A.3.3.1 Alternative Syntaxes

While the semantics of the ECA rules provides the infrastructure for these variables, the markup of specific languages must provide the actual handling of variables (mainly: binding variables) in its expressions. We propose to use a uniform handling of variables in the ECA language (see also Section A.3.3.3), and in the E, Q, T, and A component languages.

Variables: Syntax. We propose to use the following constructs for binding variables, where we borrow from several language designs (note that in contrast to some other languages, binding and using variables is here the same):

1. (syntax for use borrowed from XQuery, XSLT and XML-QL; syntax and semantics for binding borrowed from XML-QL [18]): (use) variables by {$*var-name*}:

   ```
   <travel:canceled-flight number=$flight/>
   ```

   or

   ```
   <travel:canceled-flight number="{$flight}"/>
   ```

   matches an event (e.g., <travel:canceled-flight number="LH0815"/>) and binds $flight to the number of the flight. Note that the second variant is valid XML where the "{$*name*}" is embraced in quotes and parentheses. This *atomic event specification language (AESL)* will be considered again in Section A.4.2.2.1.

2. (borrowed from XSLT): (bind) variables by <variable name="..."> elements:

   ```
   <foo:variable name="name">
     content
   </foo:variable>
   ```

   where *content* is any expression (e.g., an XML fragment in a match-style query language, an event specification or a query) that returns some value. The variable is then bound to this value (see also Examples 14 and 9 below). Note that the foo:variable elements reside in the component language's namespace (the variable is part of the component language tree).

3. (reminiscent of F-Logic and XPathLog): alternative way of binding results of subexpressions to variables:

```
<foo:bla foo:variable="name">
  content
</foo:bla>
```

where <foo:bla> is any expression (e.g., an XML fragment in a match-style query language, an event specification or a query) that returns some value. The variable *name* is then bound to this value.

This syntax corresponds to o[m→ V[a→ b;...]] (V bound to the results of the property m of o) in F-Logic and even more e[b→ V[c→ d;...]] in XPathLog (V bound to the b subelements of e). Xcerpt uses a similar construct for bindings variables in its query terms.

4. XLink-out-of-line or RDF style: Variable bindings can be seen as references into a term (XLink) or as edges between a variable name and a term. Then, they can be expressed by elements of the form

   ```
   <foo:variable name="name" select="$component/xpath-expression" />
   ```

   where *xpath-expression* references into the term structure of a component expression (should be relative to the rule, the component, or self). This can mainly be applied for *matching* purposes e.g. for (atomic) events or (XML) pattern queries.

We propose to use similar constructs also in the component languages, but the actual decision is up to the language designers.

A.3.3.2 Discussion: Variable Syntax

The above syntaxes have their advantages and disadvantages:

- all three are "used" to people from certain communities.
- case (1) is the most simple and "clear", both for people from Logic Programming (matching of variables) and from programming languages like XQuery. We currently do not see any problems with it. But it is only usable for simple cases (attributes or whole element contents).
- case (2): this interferes with the term structure. This is not relevant for *executing* a rule, but when rules and their components are seen as resources and have to be addressed. The navigation expressions for addressing subterms in a pattern depends on where such variables are mentioned.
- case (3): here, the use of variables is indicated *inside* of elements (which violates their DTD). This is problematic when submitting components to services (in this case, the attribute must be removed; it is intended to be used by the service that processes the *outside*).
- case (4): here, the variables do not interfere with the term structure. On the other hand, since they are decoupled from the structure, reasoning requires to parse the language used in the select attribute.

  The path expressions should be restricted to the same level or component where the <variable> is located (e.g., on rule level binding the result of a query, but not binding the second subexpression of an event expression) since otherwise communication is required that is not provided by the loosely coupled language modules.

It is obvious that this must be considered carefully. When using RDF instead of XML markup, things become easier since these things can be dealt with in separate triples (in any namespace). Variables are then like "annotations" to a structure. Note that the above case (4) is already in RDF style.

Next, we discuss this issue for ECA-ML. Another discussion can be found in Section A.4.2.6 when a markup for a composite event language closely related to the SNOOP event algebra [17] is presented.

A.3.3.3 Variable Bindings by ECA Rules

For making the functional result part of a component (e.g., i.e., the of the event component, or of a query component) accessible in the ECA rule, it must immediately be bound to a variable on the rule level. For this ECA-ML provides several equivalent mechanisms according to the above mechanisms:

- eca:variable elements may occur on the ECA rule level that contain an expression:

  ```
  <eca:rule. . . >
    <eca:variable name="result-var">
      <eca:event . . . >
        :
      </eca:event
    </eca:variable>
      :
  </eca:rule>
  ```

  (analogous for queries)

- eca:variable elements may occur on the upper level *inside* a component. The following is equivalent to the above:

  ```
  <eca:rule. . . >
    <eca:event. . . >
      <eca:variable name="result-var" >
        contents
      </eca:variable>
    </eca:event>
      :
  </eca:rule>
  ```

  Here, the use of a variable is indicated *inside* of the event element – although actually intended to be used by the service that processes the *outside*. Only the contents of the eca:variable element must be sent to the event component service. (Note that if the component language, say, foo also provides a variable syntax as above, also <foo:variable would have the same semantics.)

- the following syntax according to (3) is also equivalent:

  ```
  <eca:rule. . . >
    <eca:event [eca:]variable="result-var" >
      contents
    </eca:event>
      :
  </eca:rule>
  ```

  Again, the use of a variable is indicated *in* the event element – although actually intended to be used by the service that processes the *outside*. The eca:variable attribute must not be sent to the event component service.

- <eca:variable> elements of the form (4)

  ```
  <eca:variable name="name" language="xpath" select="expr" />
  ```

  can be used for binding a new variable based on already bound ones in *expr*, e.g.,

```
<eca:rule...>
   something that binds variable x
  <eca:variable name="y" language= "xpath" select="$x/foo/@bar"/>
   :
</eca:rule>
```

These expressions can be in any language (e.g. XPath; the language attribute can be omitted if the service uses a default language) that an ECA service implements for such simple *local* evaluations. The above is a shorthand for

```
<eca:variable name="name">
  <eca:query>
    <eca:opaque language="xpath"> expr </eca:opaque>
  </eca:query>
</eca:variable>
```

- a similar style is used for "forward" declarations for binding a variable to a functional result of a rule component by

  ```
  <eca:variable name="name" language="xpath" bind-to="expr"/>
  ```

  (note: bind-to instead of select) where *expr* begins with $rule, e.g.,

```
<eca:rule...>
  <eca:variable name="queryresult2" language= "xpath" bind-to="$rule/query[2]"/>
   :
</eca:rule>
```

Such variables are bound as follows:

- If the result from evaluating a component is one or more <logvars:answer>, then for each <logvars:answer>, every <logvars:tuple> in the <logvars:variable-bindings> part is extended with the variable *result-var* which is bound to the <logvars:result> part of the <logvars:answer> (see Examples 14 and 9 and below).
- If the result is other XML (e.g., from evaluating an XPath query or analogously for RDF), *result-var* is simply bound to it in the same way as in XSLT (this often saves writing wrappers to the above exchange format).

Note that most currently existing tools (e.g. query interfaces) do not return their data in such a format. In such cases, wrappers (that can be provided locally by the ECA service) can be used. Note that the XQuery return clause (and similar constructs like XML generating functions in SQLX) can be used to return this format directly (this functionality is especially used in the prototype for rapid prototyping using existing XQuery services; see Section A.7.1.2).

Usually, event detection returns the relevant event sequence in this way, we illustrate this situation in Section A.3.4.1.

Requirements.

- if a request contains multiple tuples over variables $X_1, \ldots, X_n$, then *each* resulting tuple must bind a superset of $X_1, \ldots, X_n$ (to allow an unambiguous correlation). Such services incorporate a full join semantics (cf. declaration of returned-vars in Section A.3.1.5).

- if a service allows only for requests that contain a single tuple of variable bindings $X_1, \ldots, X_n$, the ECA engine must invoke it for each tuple of bindings $X_1, \ldots, X_n$ that occurs in the current variable bindings. It must then correlate each of the results to the correct original tuple(s). Such services are in general only lookup services.

$\Rightarrow$ it should be part of the service description, to indicate whether the resulting variable bindings are a superset of the input/used variables (cf. Section A.3.1.5 and Section A.6.1.2).

A.3.4 Operational Aspects of Rule Execution

The previous sections presented logical variables used for the declarative semantics of ECA rules, together with explicit markup for representation and communication of variable bindings
We did not yet discuss the component languages and their engines, or the domain languages that provide the atomic notions. So far, it is sufficient to be able to express them in an illustrative way by opaque expressions, and assume the actual component evaluation to be done by "demons" that understand an agreed format for downward communication and that return answers in an agreed format for upward communication. Component services will be discussed later in Chapter A.4. We first clarify the canonic operational semantics of the ECA engine.

The operational semantics of ECA rules differs then from that of logical rules in that the rule body is not just a query, but consists of a triggering event, and then of the evaluation of queries.

Example 6 *Consider again the situation from Example 3: For* each *"canceled flight" event, the rule is "fired". "Fired" means that the event component produces* one *"answer". The next "canceled flight" message fires another rule instance that will be completely independent.*

For that "answer" to the event component, the (one and only) destination is selected, bound to the Destination *variable, and for this pair* (Flight, Destination)*, the action is triggered.*

A different (and more complex) situation occurs, if the query component produces several answers, e.g. when a query selects all customers who have a reservation for this flight and sends each of them an e-mail.

A.3.4.1 Firing ECA Rules: the Event Component

An event is something that occurs (or, that is detected – in contrast to local databases that represent a closed world, the occurrence of an event somewhere in the Web does not necessarily mean that it is actually detected anywhere where it is relevant). Formally, detection of an event results in an occurrence indication, together with information that has been collected (consider here the data exchange format discussed in Section A.3.1.4). The ECA engine must then execute the rule accordingly, using the obtained variable bindings. Note that the cardinality of answers must be considered in this case:

Example 7 (Exam Registration) *Consider the following scenario: for an exam, first the (online) registration is opened, then students register, and at a given timepoint, the registration closes. Assume the events to be marked up as e.g.*

```
<uni:reg_open subject="Databases" />
<uni:register subject="Databases" name="John Doe" />
<uni:reg_close subject="Databases" />
```

A can e.g. describe an action to be taken "if registration for an exam E is opened, students $X_1, \ldots, X_n$ register, and registration for E closes". The composite event is then formulated as "registration to an exam is closed after (it had been opened and) students s_1, ..., s_n registered" and is reported at the timepoint when the registration closes. With its occurrence indication, information about the subject E and registered students $X_1, \ldots, X_n$ is given. Such a cumulative *semantics is provided by appropriate event operators, e.g. by SNOOP [17].*

The rule must then be fired for this one event.

Example 8 *Consider the following rule: "If flight F is delayed for more than 25 minutes, do ...". Information about delayed flights is available all 10 minutes as a message including a report of the form*

```
<msg:receive-message sender="service@fraport.com">
  <msg:content xmlns:travel="http://www.semwebtech.org/domains/2006/travel">
    <travel:delayed-flight flight="LH1234" minutes="30"/>
    <travel:delayed-flight flight="AF0815" minutes="90"/>
    <travel:delayed-flight flight="CY42" minutes="60"/>
      :
    <travel:canceled-flight flight="AL4711"/>
    :
  </msg:content>
</msg:receive-message>
```

There are two ways how to maintain this rule:

- *XML level: the rule designer knows that such messages come in, and formulates a suitable atomic event pattern (cf. Section A.4.2 as*

  ```
  <eca:rule>
    <eca:event xmlns:travel="http://www.semwebtech.org/domains/2006/travel" xmlns:mail="....">
      <eca:atomic-event>
        <mail:receive-message sender="service@fraport.com">
          <mail:content>
            <travel:delayed flight="$flight" minutes= "$minutes"/>
          </mail:content>
        </mail:receive-message sender="service@fraport.com">
      </eca:atomic-event>
    </eca:event>
      :
  </eca:rule>
  ```

  which binds variables flight and minutes.

- *Formulating the rule purely in the application domain as*

  ```
  <eca:rule>
    <eca:event xmlns:travel="http://www.semwebtech.org/domains/2006/travel">
      <eca:atomic-event>
        <travel:delayed flight="$flight" minutes="$minutes"/>
      </eca:atomic-event>
    </eca:event>
  </eca:rule>
  ```

  and using an ECE (see Section A.2.1.5) event derivation rule "if there is a message whose content matches event E, then consider E to be detected".

  Note that use of <eca:atomic-event> is optional, and everything inside is here a non-normative example. Details of embedding atomic events into surrounding languages are discussed in Section A.4.2.5.

The answer from the event detection module can result in

- *one answer, containing several tuples, or*

- *several answers, containing one tuple.*

Note that later, the same event pattern will be detected again, which again fires the rule.

In case that an occurrence indication contains *multiple* tuples of variable bindings, the semantics must be carefully considered: The tuples must be regarded as semantically independent since they –although "just by chance" detected at the same time– represent independent events. For that reason, a correct (but not always most efficient) semantics would be that the ECA engine immediately separates them and fires independent instances of the rule (see Section A.3.5.2).

Thus, if a final event should report about something set-like, this must be contained in the semantics of the event language, not of the ECA language – then it is practically too late. As stated above, an explicitely cumulative semantics is e.g. supported by the SNOOP event algebra.

In many approaches, the "result" of event detection is the sequence of the events that "materialized" the event pattern to be detected. In this case, an appropriate way is to bind this result to a variable as shown above and afterwards the values of other variables can be extracted from this one.

Example 9 *Consider the following situation from Example 7 and an event specification (using XPathLog and regular expression syntax in an obvious way):*

```
<eca:rule ... >
  <eca:variable name="Subj" />
  <eca:variable name="regseq">
    <eca:event>
      <eca:opaque language='xpathlog-events'>
        <eca:bind-variable name="Subj" />
        reg_open[@subject→Subj], register[@subject→Subj]*, reg_close[@subject→Subj]
      </eca:opaque>
    </eca:event>
  </eca:variable>
    :
</eca:rule>
```

The returned information from the event detection service in the markup proposed in Section A.3.2.3 looks as follows, returning the relevant event sequence. The variable Subj has also been bound in the event component:

```
<logvars:answer component="event" ref="identifier">
  <logvars:result>
    <uni:reg_open subject="Databases" />
    <uni:register subject="Databases" name="John Doe" />
    <uni:register subject="Databases" name="Scott Tiger" />
      :
    <uni:reg_close subject="Databases" />
  </logvars:result>
  <logvars:variable-bindings>
    <logvars:tuple>
      <logvars:variable name="Subj">Databases</logvars:variable>
    </logvars:tuple>
  </logvars:variable-bindings>
</logvars:answer>
```

Next, the variable regseq is bound to the <result> part. The variable bindings after completely evaluating the event component look as follows:

```
<logvars:variable-bindings>
  <logvars:tuple>
    <logvars:variable name="regseq">
      <uni:reg_open subject="Databases"/>
      <uni:register subject="Databases" name="John Doe"/>
      <uni:register subject="Databases" name="Scott Tiger"/>
        :
      <uni:reg_close subject="Databases"/>
    </logvars:variable>
    <logvars:variable name="Subj">Databases</logvars:variable>
  </logvars:tuple>
</logvars:variable-bindings>
```

A.3.5 The Query Component

This second component is concerned with *static* information that is obtained and restructured from two areas:

- analyzing the data that has been collected by the event component (in the variable bindings), and
- based on this data, stating queries against databases and the Web.

Whereas the event component of a rule may be "answered" by detecting occurrences of the event pattern several times, the query component returns all answers at the same time. The query component is very similar to the evaluation of database queries and rule bodies in Logic Programming: in general, it results in a set of tuples of variable bindings (that are possible answers to a query).

Grouping: Set-Valued vs. Multi-Valued. An important issue here is to deal with sets (e.g., in the above examples, all customers who booked a flight that has been canceled, or all students that registered for an exam):

- bind a variable to a collection, e.g.,
  $\beta = \{\mathsf{Subj} \rightarrow \text{'Databases'}, \mathsf{student} \rightarrow \{\text{'John Doe'}, \text{'Scott Tiger'}, \ldots\}\}$, or
- produce separate tuples of variable bindings:
  $\beta_1 = \{\mathsf{Subj} \rightarrow \text{'Databases'}, \mathsf{student} \rightarrow \text{'John Doe'}\}$,
  $\beta_2 = \{\mathsf{Subj} \rightarrow \text{'Databases'}, \mathsf{student} \rightarrow \text{'Scott Tiger'}\}$.

We follow again the Logic Programming specification that every answer produces a variable binding. For variable binding by matching (as in Datalog, F-Logic, XPathLog, Xcerpt etc.), this is obvious. Since we also allow variable bindings in the functional XSLT style, the semantics is adapted accordingly:

- each answer node of an XPath expression yields a variable binding;
- each node that is *return*ed by an XQuery query yields a variable binding; if the XQuery query is of the form
  <*name*>{ for ... where ... return ...} </*name*> ,
  then the whole result yields a single variable binding.

Example 10 *Consider again Example 9 where the resulting event contained several registrations of students. For doing anything useful, their names have to be extracted.*

1. *as multiple string-valued variables:*

```
<eca:rule ... >
    :
  same as above, binding variables "Subj" and "regseq"
    :
  <eca:variable name="Student">
    <eca:query>
      <eca:opaque language='xpath'>
        $regseq//uni:register[@subject=$Subj]/@name/string()
      </eca:opaque>
    </eca:query>
  </eca:variable>
    :
</eca:rule>
```

The above query generates the extended variable bindings
$\beta_1 = \{Subj \rightarrow$ *'Databases'*, $regseq \rightarrow$ *(as above)*, $Student \rightarrow$ *'John Doe'*$\}$,
$\beta_2 = \{Subj \rightarrow$ *'Databases'*, $regseq \rightarrow$ *(as above)*, $Student \rightarrow$ *'Scott Tiger'*$\}$.

2. *or as a single variable:*

```
<eca:rule ... >
    :
  same as above, binding variables "Subj" and "regseq"
    :
  <eca:variable name="Students">
    <eca:query>
      <eca:opaque language='xquery'>
        <students>
          for $s in $regseq//uni:register[@subject=$Subj]/@name/string()
          return <name> { $s } </name>
        </students>
      </eca:opaque>
    </eca:query>
  </eca:variable>
    :
</eca:rule>
```

This query generates the extended variable binding
$\beta = \{$ *Subj* $\rightarrow$ *'Databases'*, *regseq* $\rightarrow$ *(as above)*,
Students $\rightarrow$ *<students> <name>John Doe</name>*
<name>Scott Tiger</name> </students>$\}$

The above query showed how data from the variable bindings obtained from the event detection is extracted. Note that this query is very similar to the *event queries* mentioned in XChange (instead of the above opaque query, also an Xcerpt/XChange query could have been used).

The next example shows a query against an XML repository in the Web. Again, the answer of the query component is represented in a single XML variable. This example then leads immediately to questions about handling the action component.

Example 11 *Consider an ECA rule with opaque components (using different languages) that, whenever a flight is canceled, notifies* every customer *who has a reservation for this flight (e.g., by SMS), and sends a message to the airport hotel with the names of* all customers *to make a pre-reservation for this night.*

```
<eca:rule xmlns:eca="http://www.semwebtech.org/eca/2006/eca-ml"
```

```
    xmlns:xpath="http://www.w3.org/XPath"
  <eca:variable name="Bookings"> http://localhost/schedule.xml </eca:variable>
  <eca:variable name="Flight" />
  <eca:event>
    <eca:opaque language='datalog-match'>
      <eca:bind-variable name="Flight" />
      flight_cancellation(Flight)   <!-- matches Flight against received message -->
    </eca:opaque>
  </eca:event>
  <eca:variable name="Customers">
    <eca:query>
      <eca:opaque language="xquery">
        <eca:use-variable name="Flight" />
        <eca:use-variable name="Bookings" />
        return
          <customers>
          { for $c in document($Bookings)//flight[@id=$Flight]/reservation/customer
            return $c }
          </customers>
      </eca:opaque>
    </eca:query>
  </eca:variable>
  <!-- evaluates XPath expression and binds each of the results to the variable 'Customers' -->
  <eca:test>
    <eca:opaque language='xpath'>
      $Customers/customer
    </eca:opaque>
  </eca:test>
  <eca:action>
    <eca:opaque language='pseudocode'>
      <eca:use-variable name="Customers" />
      <eca:use-variable name="Flight" />
      send one message with all Customers/customer/name to the hotel,
      then
      for each N in Customers/customer/@phonenr do
        notify_cancellation(Flight, sms:N)
    </eca:opaque>
  </eca:action>
</eca:rule>
```

In the first case, the action has a multi-valued semantics, whereas in the second case it has a set-valued semantics. This simple case could be solved by binding the *set* of customers to a variable. The above "solution" moves the solution inside the query component, and is sufficient in this case, but not in general (and less declarative).

Note that there can be query services that return the query answer stepwise (which makes sense if the receiver is then able to continue already for some of the answers). This should be considered in the service descriptions (of the ECA service and of the query service) and in the message answer (default: complete).

A.3.5.1 The Test Component

The test component is still concerned with the information obtained so far. It evaluates a condition which is a mapping from a knowledge base to true/false. In general, the evaluation of conditions is based on a logic over literals with boolean combinators and quantifiers. A Markup Language exists with FOL-RuleML [10]. Since first-order logic is in general undecidable, it is recommended to use suitable fragments. Instead of first-order atoms, also "atoms" of other data models can be used. Additionaly, we envisage to allow to use simple expressions like XPath of a language that

is locally supported in the ECA engine. Note that XPath expressions are also literals that result in a true/false (true if the result set is non-empty) value (as in the above example). Variables are communicated to the test in the same way as above. The test component returns then the set of tuples that satisfy the condition (for further propagation to the action component).

A.3.5.2 Summary of Event, Query and Test Semantics

Given the variable bindings resulting from the event component, the semantics of evaluating queries is based on the usual answer & join semantics of Logic Programming: for the cases where queries only contain positive occurrences of variables, the resulting variable bindings of the event and query components are just the join of the individual sets of variable bindings. As long as only positive queries are used, the semantics of the query component is commutative. In case of negative occurrences, the usual safety constraints apply, variables must be bound positively before they can be used in a negated occurrence. Then, negation is interpreted as set difference. The test component acts as a selection that removes all variable bindings that do not satisfy the test.

The normal form induces a sequential operational semantics; other evaluation and execution strategies are possible based on equivalence transformations .

In each stage, the variable bindings are considered as a set of tuples that is represented in XML as above, and communicated with the event, query and test services. Note that this allows for aggregation and grouping constructs in queries and tests.

Multiple Occurrences. The "plain" semantics of an ECA rule assumes that a rule is fired for each individual occurence of the event pattern given in its event component. As shown above, that multiple independent instances of an event pattern are detected at the same time, and reported as a whole. In such cases, it is possible to fire the rule for all these occurrences together (since we do not bind program variables, but use the notion of logical variables and sets of tuples of variable bindings, the underlying declarative semantics allows for this). Semantically, there is a only a difference if transactional issues in the action component are considered. In this case it is enough to separate the instances when executing the action component.

A.3.5.3 The Action Component

The action component is the one where *actually* something is done in the ECA rule: for each variable binding, the action component is executed. Actions do not have a result. Thus, there is only a set of input variables (usually called parameters in operational semantics) that is submitted in the same way as above.

The action component may consist of several <eca:action> elements which can use different action languages. The semantics is that all actions are executed. Note that actions are not allowed to bind variables, thus they are independent on this level. (Note that sequential and conjunctive execution of actions can also be specified on the level of action languages *inside* the <eca:action> element.)

Grouping and de-grouping execution steps. Here, the possibility of grouping and de-grouping is required: it makes a strong difference if a set is represented by one tuple of variables, or the whole set is bound as a set in one tuple.

Example 12 *Consider again Example 9 where the names of students that have registered for an exam have been collected:*

1. *In the first case, we had the following variable bindings:*
   $\beta_1 = \{Subj \rightarrow$ *'Databases'*$, regseq \rightarrow$ *(as above)*$, Student \rightarrow$ *'John Doe'*$\}$,
   $\beta_2 = \{Subj \rightarrow$ *'Databases'*$, regseq \rightarrow$ *(as above)*$, Student \rightarrow$ *'Scott Tiger'*$\}$,
   i.e., a set of two tuples.

If the action now is "(for each tuple), send the lecturer a mail with the value of the variable Student*", the lecturer will get two mails, each one with one student.*

Instead, we want to send the lecturer one *mail with the names of all registered students.*

2. *For such cases, a functionality for "group by Subject" (which results in only one tuple) would be useful, e.g. a way to group by* Subj*, collecting all names in a list (and forgetting about* regseq*, resulting in*
   $\beta_{grp} = \{$Subj $\rightarrow$ 'Databases', Student $\rightarrow$ {'John Doe','Scott Tiger'}$\}$,
   for which the action then can be called.

3. *In the second case, we have only one variable binding (very similar to β_{grp} above,*

$$\beta = \{\ \{\quad \text{Subj} \rightarrow \text{'Databases'}, \quad \text{regseq} \rightarrow \text{(as above)},\ \text{Students} \rightarrow \langle\text{students}\rangle\ \langle\text{name}\rangle\text{John Doe}\langle/\text{name}\rangle\ \langle\text{name}\rangle\text{Scott Tiger}\langle/\text{name}\rangle\ \langle/\text{students}\rangle\}$$

   for which the action can be called immediately (and e.g. submit the XML structure of Students *as the message content).*

On the other hand, if the task is "to send a message to each of the registered students", the first case is immediately suitable, whereas the second one needs either an iteration inside the action component ("for each name in Students *do ...") or, – more declaratively on the ECA level – an* ungrouping*, resulting in the variable bindings of β_1 and β_2.*

Such cases are very frequent. A rule can even contain actions of both kinds, e.g., in

> For the above E&Q components, "send the lecturer an e-mail with the names of all students *and* send a confirmation message to each of the registered students".

In such cases, the number of variable bindings must be changed by grouping or de-grouping, dependent on what the result of the E&Q components are. Grouping and ungrouping on the ECA level is allowed *before* each <eca:action> element. We propose to inherit these elements also to the action languages:

- all tuples that coincide in *all* named variables are grouped together; the other variables are aggregated as lists, sum, avg etc., or omitted (a default can be specified; *aggr-op* = list|sum|avg|omit|...):

```
<eca:group-by aggr="aggr-op">
  <eca:group-variable name="name" />
    :
  <eca:group-variable name="name" />
  <eca:aggr-variable op="aggr-op" name="name" aggr-name="name" />
    :
  <eca:aggr-variable op="aggr-op" name="name" aggr-name="name" />
</eca:group-by>
```

- Flattening a list or sequence: The new variable binding is obtained by splitting a list, or applying a query to the variable (XPath, or any default language configured by the ECA service or the rule). Note that this can also be done by a sequence of <eca:query> elements.

```
<eca:ungroup-by>
  <eca:ungroup-variable>variable-name
    [<eca:ungroup-query> ...</eca:ungroup-query>]
  </eca:ungroup-variable>
    :
  <eca:ungroup-variable>variable-name
    [<eca:ungroup-query> ...</eca:ungroup-query>]
  </eca:ungroup-variable>
</eca:ungroup-by>
```

A.3.5.4 Transactions

Although the issue of transactions does not directly have to do with the semantics of ECA rules, some issues should be raised here. Transactional issues are only concerned with the action component (events can neither fail nor be rolled back, queries and tests can also not "fail" and there is nothing to be rolled back). Transactional functionality can be offered independently by the action languages *inside* the <eca:action> elements.

Since we stated above that the semantics for excution of the action component is the same as for executing the head of a deductive rule, i.e., handling it separately for each tuple of variable bindings, transactions that should cover the whole group must explicitly be expressed on the ECA level.
For this, we propose an

<**eca:transaction** *attributes*> ... </**eca:transaction**>

element that can occur around or anywhere in an <eca:action> element.
With this, e.g., actions for *all* tuples can be grouped as a transaction by

```
<eca:rule ...>
  <eca:event> ...</eca:event>
  <eca:query> ...</eca:query>
  <eca:test> ... </eca:test>
  <eca:transaction>
    <eca:action> ... </eca:action>
  </eca:transaction>
</eca:rule>
```

A further element allows to take a group of tuples together for execution of a transaction (note that in contrast to <eca:group-by>, the number of tuples does not change):

```
<eca:transaction-group-by/>
  <eca:group-variable name="name" />
    :
  <eca:group-variable name="name" />
</eca:transaction-group-by>
```

(e.g., if an event contains a list of delayed flights, the query component returns a pair of variable bindings for each customer, and the appropriate actions should be separate transactions for each flight).

A.3.5.5 Examples

Example 13 (Rental Cars (Revisited)) *Execution is then as follows: after the event part, the variable bindings look as follows, containing only one tuple:*

```
<logvars:variable-bindings>
  <logvars:tuple>
    <logvars:variable name="Person">John Doe</logvars:variable>
    <logvars:variable name="To">Paris</logvars:variable>
  </logvars:tuple>
</logvars:variable-bindings>
```

The first query for looking up what car(s) the person owns declares Person and OwnCar to be used variables. Since up to now, only Person is bound, owns(Person,OwnCar)[{(Person/"John Doe")}] is evaluated by the local Datalog database. Consider now the case that John Doe owns two cars. It results in the bindings

```
<logvars:variable-bindings>
  <logvars:tuple>
    <logvars:variable name="Person">John Doe</logvars:variable>
    <logvars:variable name="OwnCar">Golf</logvars:variable>
  </logvars:tuple>
  <logvars:tuple>
    <logvars:variable name="Person">John Doe</logvars:variable>
    <logvars:variable name="OwnCar">Passat</logvars:variable>
  </logvars:tuple>
</logvars:variable-bindings>
```

These bindings are joined with the first ones, resulting in

```
<logvars:variable-bindings>
  <logvars:tuple>
    <logvars:variable name="Person">John Doe</logvars:variable>
    <logvars:variable name="To">Paris</logvars:variable>
    <logvars:variable name="OwnCar">Golf</logvars:variable>
  </logvars:tuple>
  <logvars:tuple>
    <logvars:variable name="Person">John Doe</logvars:variable>
    <logvars:variable name="To">Paris</logvars:variable>
    <logvars:variable name="OwnCar">Passat</logvars:variable>
  </logvars:tuple>
</logvars:variable-bindings>
```

The next query against the local Datalog database for obtaining the class of the respective car(s) declares OwnCar *and* Class *to be used. Up to now, only* OwnCar *is bound, thus variable, i.e., the query* class(OwnCar,Class) [{(OwnCar="Golf"),(OwnCar="Passat")}] *is evaluated against the local Datalog database. It results in the bindings*

```
<logvars:variable-bindings>
  <logvars:tuple>
    <logvars:variable name="OwnCar">Golf</logvars:variable>
    <logvars:variable name="Class">B</logvars:variable>
  </logvars:tuple>
  <logvars:tuple>
    <logvars:variable name="OwnCar">Passat</logvars:variable>
    <logvars:variable name="Class">C</logvars:variable>
  </logvars:tuple>
</logvars:variable-bindings>
```

which is again joined with the existing ones:

```
<logvars:variable-bindings>
  <logvars:tuple>
    <logvars:variable name="Person">John Doe</logvars:variable>
    <logvars:variable name="To">Paris</logvars:variable>
    <logvars:variable name="OwnCar">Golf</logvars:variable>
    <logvars:variable name="Class">B</logvars:variable>
  </logvars:tuple>
  <logvars:tuple>
    <logvars:variable name="Person">John Doe</logvars:variable>
    <logvars:variable name="To">Paris</logvars:variable>
    <logvars:variable name="OwnCar">Passat</logvars:variable>
```

```
      <logvars:variable name="Class">C</logvars:variable>
    </logvars:tuple>
  </logvars:variable-bindings>
```

The next query for looking up the available cars at the destination declares To *to be an input variable, i.e., the query* http://localhost/lookup-cars?_place='Paris' *is evaluated, resulting in the bindings*

```
  <logvars:variable-bindings>
    <logvars:tuple>
      <logvars:variable name="Class">B</logvars:variable>
      <logvars:variable name="Model">C4</logvars:variable>
      <logvars:variable name="Price">50</logvars:variable>
    </logvars:tuple>
    <logvars:tuple>
      <logvars:variable name="Class">B</logvars:variable>
      <logvars:variable name="Model">Golf</logvars:variable>
      <logvars:variable name="Price">65</logvars:variable>
    </logvars:tuple>
    <logvars:tuple>
      <logvars:variable name="Class">D</logvars:variable>
      <logvars:variable name="Model">C6</logvars:variable>
      <logvars:variable name="Price">150</logvars:variable>
    </logvars:tuple>
  </logvars:variable-bindings>
```

Note that the ECA engine must keep the knowledge that all these tuples refer to To/"Paris"*. Joining the result removes the tuple dealing with the class "C" since no such cars are available in Paris and results in*

```
  <logvars:variable-bindings>
    <logvars:tuple>
      <logvars:variable name="Person">John Doe</logvars:variable>
      <logvars:variable name="To">Paris</logvars:variable>
      <logvars:variable name="OwnCar">Golf</logvars:variable>
      <logvars:variable name="Class">B</logvars:variable>
      <logvars:variable name="Model">C4</logvars:variable>
      <logvars:variable name="Price">50</logvars:variable>
    </logvars:tuple>
    <logvars:tuple>
      <logvars:variable name="Person">John Doe</logvars:variable>
      <logvars:variable name="To">Paris</logvars:variable>
      <logvars:variable name="OwnCar">Golf</logvars:variable>
      <logvars:variable name="Class">B</logvars:variable>
      <logvars:variable name="Model">Golf</logvars:variable>
      <logvars:variable name="Price">65</logvars:variable>
    </logvars:tuple>
  </logvars:variable-bindings>
```

The final action declares Model *and* Price *as input. Thus, only the bindings*

```
  <logvars:variable-bindings>
    <logvars:tuple>
      <logvars:variable name="Model">C4</logvars:variable>
      <logvars:variable name="Price">50</logvars:variable>
```

```
    </logvars:tuple>
    <logvars:tuple>
      <logvars:variable name="Model">Golf</logvars:variable>
      <logvars:variable name="Price">65</logvars:variable>
    </logvars:tuple>
  </logvars:variable-bindings>
```

are submitted to it.

Example 14 *Consider an ECA rule with opaque components (using different languages) that, whenever a flight is canceled, sends a message to the destination airport that the flight will not take place:*

```
<eca:rule xmlns:eca="http://www.semwebtech.org/eca/2006/eca-ml"
    xmlns:xpath="http://www.w3.org/XPath"
  <eca:variable name="Schedule">http://localhost/schedule.xml</eca:variable>
  <eca:variable name="Flight" />
  <eca:variable name="Destination" />
  <eca:variable name="event">
    <eca:event>
      <eca:atomic-event>
        <travel:canceled-flight/>
        <!-- matches any travel:canceled-flight event, e.g. <travel:canceled-flight code="LH1234" /> -->
      </eca:atomic-event>
    </eca:event>
  </eca:variable>
      <!-- the matched event is now bound to the variable $event -->
  <eca:variable name="Flight" language="xpath" select="$event/canceled-flight/string(@code)" />
  <eca:variable name="Destination">
    <eca:query>
      <eca:opaque language='xpath'>
        <eca:input-variable name="Flight" use="$Flight" />
        <eca:input-variable name="Schedule" use="$Schedule" />
        string(document($Schedule)//flight[@id=$Flight]/@to)
      </eca:opaque>
    </eca:query>
  </eca:variable>
  <!-- evaluates XPath expression and binds the result to the variable 'Destination' -->
  <eca:test>
    <eca:opaque language='boolean'>true</eca:opaque>
  </eca:test>
  <eca:action>
    <eca:opaque language='pseudocode'>
      <eca:input-variable name="Flight" use="$Flight" />
      <eca:input-variable name="Destination" />
      send "Flight $Flight has been canceled today" to the destination airport ...
    </eca:opaque>
  </eca:action>
</eca:rule>
```

The ECA engine proceeds as follows: It binds the variable Schedule *as a constant to the given value and allocates variables* Flight *and* Destination*. The event component consists only of an atomic event. If such an event, e.g.* <travel:canceled-flight code="LH1234" /> *is detected (by matching), it is bound to the variable* event *(this semantics coincides with most event detection semantics that return the relevant event sequence as the result). In the next step, the variable* Flight *is bound by evaluating an XPath expression against the value of* event*, yielding the binding* Flight/"LH1234"*.*

Next, the ECA engine submits the query where ($Schedule *is replaced with the constant URL, and* $Flight *replaced with the actual binding "LH1234") to the XPath engine, that evaluates the*

expression and returns its result, i.e., the identifier of the destination airport (e.g., "FRA"). The ECA engine binds the returned result to its variable Destination*. The condition is then empty (every flight has a destination). Next, the pseudocode fragment in the action component is equipped with the flight number and the destination airport and a message is sent.*

Chapter (Appendix A: ECA Framework) A.4

Abstract Semantics and Communication: Component Services

Component languages/services and domain languages/services can be provided by everybody somewhere in the (Semantic) Web. We do not propose a central registry of languages. A language becomes known to an ECA engine by being referenced in some rule (with its namespace URI). The algebraic components provide the glue between atomic notions and the ECA level. Usually, the "result" of an algebra expression is of the same type as the elements of its (formal) domain. Thus, different algebras of the same type can usually be nested (in some sense forming a bigger algebra that provides the union of the operators).

Communication issues from the point of view of the *rule semantics*, i.e., transmission of variable bindings has already been discussed in Chapter A.3; expecially the formats for downward and upward communication in Section A.3.2. In this chapter, this is extended from the point of view of the component services.

We first discuss the general communication patterns. Then, we continue in detail with the structure, syntax, and semantics of the event component and its communication issues. The abstract semantics and markup of the query and test components are simpler since they are in general an embedding of query langauges in the framework. The discussion of the action component follows completes the discussion of the components. A short summary comcludes the chapter.

A.4.1 General Communication Patterns

For the communication between services, messages (in XML) have to be exchanged. Several communication patterns are used:

- request-response in asynchronous and synchronous way,
- some requests do not have a direct response (e.g., submitting composite event specification) but later no, one or more "answers" referring to the request will be send,
- answers always refer to some request,
- additionally, events are communicated that do not refer to a request, but are (implicitly) "answers" on a registration.

Service URLs. The framework is flexible wrt. the actual URLs where the services expect the requests. A service can e.g. receive all for all tasks requests at a single URL, or provide a separate method URL for each method. This information is managed by *Language and Service Registries (LSRs)*, see Section A.6.5.1.

Communication Protocols. The communication can be implemented by HTTP or SOAP. The prototype uses the HTTP POST method Nevertheless, it is recommended to plan to allow for both.

Message Contents.

- component expressions (i.e., composite events, atomic event specifications, queries, actions): in plain XML representation as elements in the respective namespaces (note: without the eca:event etc. elements around them – they must only contain markup of the service's namespace),
- answers to requests: as <logvars:answers> or <logvars:answer> XML elements.

The design of messages follows the ideas of the simple SMTP protocol. Any message that awaits an answer contains the following components:

Messages/Tasks that will be answered:	
sender	by url of the sending process
Reply-To	where the answer should go (URL)
Subject	an identification that allows for uniquely identifying the answer (e.g., this can be the URI of an event component which is submitted for detection - the URI will then be used in the answer).
contents	(application-dependent)

Tasks given as XML Fragments. A request uses the same parameters as a request in real life, or an e-mail. Most tasks are specified by XML fragments (registering a rule, registering a composite event, registering an atomic event description, registering or answering a query, registering or executing an action). In these cases, the XML fragment is submitted to a suitable service. Optionally, variables that are already bound on the rule level (when handling *rule patterns*) can also be contained in the message. The message must contain the following information:

Messages/Tasks on XML Fragments:	
Sender, Reply-To	as above
Subject	URI of the component. The answer will refer to this subject (in case of asynchronous communication).
content(1)	the component or a reference to it
content(2)	optionally: variable bindings (cf. Section A.3.2.2).

Answers from Evaluation. Answers are expressed as <logvars:answers> or <logvars:answer> XML elements. They go to the url that has been specified as Reply-To by the requester. The message must contain the following information:

Messages/Tasks on XML Fragments:	
Subject	the URI sent as Subject by the requester for identifying the answer.
Sender, Reply-To	empty, optional.
content(1)	in the <logvars:answers> format described in Section A.3.2.3
content(2)	optional "comments" like "answer is complete" or "more answers follow", or error messages.

Communication of Events. As already discussed, events are plain XML fragments:

- Reply-To and Subject: none
- contents/task: event in XML markup.

Actual Markup. The Sender is always submitted in the HTTP header. The Reply-To and Subject can alternatively be submitted in the HTTP header (according to the convention that private/non-standard properties start with "X-", they are called X-Reply-To and X-Subject then). If no variable bindings are submitted. Optionally, all information sent in the body can be wrapped into an XML element hull to be not only a sequence of elements, but a single element node (Such things have to be specified in the Service Descriptions; see Section A.6.1.2.) Additional comments can also be added to answers, e.g., for incomplete query answers in a stream-like strategy.

First examples will be given in Section A.4.3.1.

A.4.2 The Event Component: Structure and Languages

The semantics of the event component and its services can be distinguished into two levels. The event component language is embedded in the language hierarchy as shown in Figure A.2.8. It is based on the event ontology as discussed in Section A.2.1.4. Composite events consist of certain combination of atomic events. Thus, the event component of a rule, which is a *specification* of a composite event (using a *Composite Event Language (CEL)*), consists of the specification of the combination, and of specifications of the contributing atomic events. Thus, languages of two types are needed:

Event Algebras: They are used to define composite events, including their specification as algebra terms as shown in Figure A.2.10 and their detection algorithms. Since the usual semantics of evaluating an event algebra expression is to return the matching event sequence, subexpressions from different algebras can be combined easily.

Atomic Event Description Formalisms: They are used to specify which atomic events are considered relevant and how these are matched (at runtime), and what is the result of detecting them. They have to allow to deal with events according to the ontology of atomic events given in Figure A.2.2

Thus, event component uses a combination of one ore more event algebras, using atomic events of one or more applications, and possibly atomic data-level events from several data models, and atomic events from application-independent services.

A.4.2.1 Atomic Events

Events occur as atomic events on the Web. For atomic events, two issues have to be considered:

- the atomic event as an occurrence on the Web (which is communicated somehow in some representation), and
- specification of atomic events to react upon in the event components of ECA rules.

A.4.2.1.1 XML Representation of Atomic Events

We assume that events are available as XML or RDF fragments.

Example 15 (Atomic Events) *Events are data fragments that are available in XML markup or as RDF fragments.*

- *The event*

```
<travel:canceled-flight flight="LH1234" />
    <travel:reason>bad weather</travel:reason>
</travel:canceled-flight>
```

is an event in the traveling domain that mean that the flight "LH1234" is canceled and also the reason is given. Note that more information (e.g., that this concerns today's flight that should depart in one hour) must then be accessible from the context. Complete information would be available if the event is of the following form <travel:canceled-flight flight="LH1234" date="10062005" />.
An example for a delayed flight is

```
<travel:delayed-flight flight="LH1234" minutes="30" />
```

- *The following events have already been used above in an example:*

```
<uni:reg_open subject="Databases" />
<uni:register subject="Databases" name="John Doe" />
```

- *in RDF, events are resources of a type "event" that also have a name and are connected to other resources as parameters.*

A.4.2.2 Atomic Event Descriptions and Formalisms

The atomic event specifications (AESs) form the leaves of the event component tree. In general, the AES specifies which occurring events are relevant to take a reaction, e.g., the name of the event, or also its contents (i.e., its parameters). Thus, there are always *two* languages involved as shown in Figure A.4.1:

- a domain language (associated with the namespace of the event),
- and an atomic event description/matching/query language/formalism for *describing* what events should actually be matched. Since the event is seen as an XML or RDF fragment, these conditions can be stated as patterns, or as queries against this fragment.

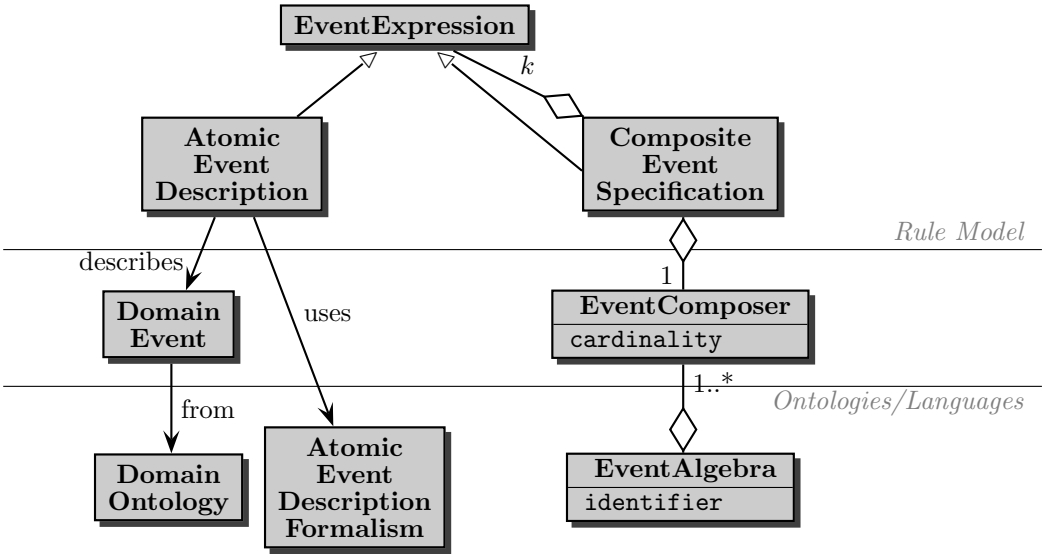


Figure A.4.1: Event Expressions: Languages

Since the event is seen as an XML or RDF fragment, the AES formalism is actually a query language that states these conditions again an XML or RDF fragment. The result of the evaluation

should be yes/no (as "the specified event occurred"), optionally (but recommended) the formalisms also should support for using and binding variables. The implementation of AES formalisms is provided by *Atomic Event Matcher (AEM)* services. An *Atomic Event Matcher* that implements such a formalism should usually return the event as functional result (since usually expected by the semantics of the surrounding event algebra), together with the variable bindings.
From the point of view of the abstract semantics, the AES must provide the following information:

- Mandatory: information about the URI of the service that is responsible for processing the AES must be accessible to the processor of the surrounding language.
  (Note that this is not available in the above example.)
- Information about the URI of the service(s)/domains that provide the actual events is required to be accessible for the service that then actually processes the AES. This information may be accessible for the surrounding service.
- The content must state (expressed by the AESL) the requirements on the events on which a reaction should be taken, e.g., the name of the event, or also its contents (i.e., its parameters). It is only required that the content is understandable for the service that then actually processes the AES, not for the surrounding service.

Some sample AES formalisms are discussed below. The actual embedding in the rule markup is dicussed later in Section A.4.2.5.

A.4.2.2.1 Event Specification by XML-QL Style Matching

Pattern-based (e.g. like XML-QL [18]) event specification continues the tree-like style of event algebras. In this case, the AES specifies the actual atomic event by a pattern and variables can be bound to fragments.

```
<travel:delayed-flight xmlns:travel="http://www.semwebtech.org/domains/2006/travel"
    foo:language="xmlqlmatch" flight="{$flight}" time="{$minutes}">
  $content
</travel:delayed-flight/>
```

specifies that an event is relevant if it is a travel:delayed event. In case that flight is already bound, this acts as a (join) condition on the code of the canceled flight, otherwise flight is bound by the matching. $content is bound to the complete content of the element.

The simple XML-QL matching style does not allow for binding specific elements to variables (only the whole contents as above). As an extension, variable elements (in the namespace of the matching formalism!) or variable references of the form $*var-name* can be used inside the pattern to express that a fragment of the event is bound or must match a variable:

```
<travel:canceled-flight xmlns:travel="http://www.semwebtech.org/domains/2006/travel"
    xmlns:aes-xmlql="http://www.semwebtech.org/eca/2006/aes-xmlql" flight="{$flight}">
  <aes-xmlql:variable name="reason">
    <travel:reason/>
  </aes-xmlql:variable>
</travel:canceled-flight>
```

matches any travel:canceled-flight event concerning a given flight with and binds the variable reason to the travel:reason element of the flight. If flight is already bound, it acts as a (join) condition on the code of the canceled flight.

Note that the above syntax is valid XML: variables as attribute values are enclosed into quotes; for distinguishing them from strings (e.g., an attribute price="$30", they are put inside braces. A similar syntax has been implemented in [61].

A.4.2.2.2 Navigation-Based Event Specification

Another alternative uses XPath style matching. Consider that for the matching, the event itself (as an XML fragment) is available as $event. Then,

```
<aed-navig:variable name="var-name" select="$event/relative-expr..." />
```

can be used to access data within the event, and elements of the form

```
<aed-navig:test condition="xpath-expr" />
```

can be used for tests (note that this is the same as eca:test for the test component). Variables can also be addressed by $*var-name* as in XQuery (for using them as join variable or for binding them to the matched value).
This formalism can be designed with a surrounding *domain-ns*:*name* element, or with a *aed-navig*:*name* element.
Here, the variants look as follows:

```
<travel:delayed-flight xmlns:travel="http://www.semwebtech.org/domains/2006/travel"
    xmlns:aes-xpath="http://www.semwebtech.org/eca/2006/aes-xpath">
  <aes-xpath:test condition="$event/@flight=$flight" />
  <aes-xpath:variable name="minutes" select="$event/@minutes" />
</travel:delayed-flight>
```

or

```
<aes-xpath:match xmlns:aes-xpath="http://www.semwebtech.org/eca/2006/aes-xpath"
    xmlns:travel="http://www.semwebtech.org/domains/2006/travel">
  <aes-xpath:test condition="$event/name()='travel:delayed-flight'" />
  <aes-xpath:variable name="flight" select="$event/@travel:flight" />
  <aes-xpath:variable name="minutes" select="$event/@travel:minutes" />
</aes-xpath:match>
```

A.4.2.2.3 Event Specification by Opaque XQuery

Going one step further regarding the event as a small XML document, XQuery can be used to check its properties. For every event that is known to the service, the query is evaluated.

```
<eca:opaque language="xquery-eventchecker" [name="name"]>
  xquery expression over $event
</eca:opaque>
```

- the check is e.g. expressed as *xquery expression over $event*,

  ```
  let $event := /
  where $event/name()="travel:flight-canceled"
  and $event/@flight= "LH123"
  return $event
  ```

- optional: indicate *event-element-name* to allow early filtering. The XQuery evaluation is only done if the name matches.

- The XQuery syntax can also be used in an experimental service to generate the upwards communication format with bound variables immediately.

Event Specification by Xcerpt. Another possibility is to use the language Xcerpt [16] developed in REWERSE WG I4. Xcerpt query terms can use and bind variables in a declarative way. For "standard" Xcerpt, the opaque mechanism has to be used. Xcerpt's XML markup would be an interesting example of an XML embedding of a query language.

A.4.2.3 Event Algebras

Event algebras, well-known from the Active Databases area, serve for specifying *composite* events by defining *terms* formed by nested application of composers over *atomic* events. There are several proposals for event algebras, defining different composers. Each composer has a semantics that specifies what the composite event means.

For dealing with composite events in the context of the ECA rules proposed here, we propose at least the following composers: "E_1 `OR` E_2", "E_1 `AND` E_2" (in arbitrary order), and "E_1 `AND THEN` E_2 `[AFTER PERIOD` $\{< \mid >\}$ *time*`]`" the latter one composing two events and using an additional parameter *time*, indicating the time that has passed between the occurrence of E_1 and E_2. Detection of a composite event means that its "final" atomic subevent is detected:

$$\begin{array}{lll} (1) & (E_1 \nabla E_2)(t) & :\Leftrightarrow\ E_1(t) \vee E_2(t)\ , \\ (2) & (E_1 \triangle E_2)(t) & :\Leftrightarrow\ \exists t_1 \leq t \leq t_1 : (E_1(t_1) \wedge E_2(t)) \vee (E_2(t_1) \wedge E_1(t)). \\ (3) & (E_1 ;_{\Delta t} E_2)(t) & :\Leftrightarrow\ \exists t_1 \leq t \leq t_1 + \Delta t : E_1(t_1) \wedge E_2(t). \end{array}$$

Event algebras contain not only the aforementioned straightforward basic connectives, but also additional operators. A bunch of event algebras have been defined that provide also e.g. "negated events" in the style that "when E_1 happened, and then E_3 but not E_2 in between", "periodic" and "cumulative" events, e.g., in the SNOOP event algebra [17] of the "Sentinel" active database system.

Example 16 (Cumulative Event, [17]) *A "cumulative aperiodic event"*

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

occurs with E_3 and reports the collected occurrences of E_2 in the meantime. Thus, its detection is defined as "if E_1 occurs, then for each occurrence of an instance of E_2, collect it, and when E_3 occurs, report all collected occurrences (in order to do something)".

A cumulative periodic event can be used for "after the end of a month, send an account statement with all entries of this month":

$$\begin{array}{l} E(Acct) := \\ A^*(first\_of\_month(m), (debit(Acct,Am) \nabla deposit(Acct,Am)), first\_of\_month(m+1)) \end{array}$$

where the event occurs with first_of_next_month. The "result" of the expression is the list of all contributing events.

XML Markup for the Event Component. The <eca:event> elements contain elements according to event algebra languages. Every subexpression is associated by its namespace with the appropriate components of the language – i.e., an event algebra or atomic expressions from underlying domains. In general, if an event algebra supports an XML markup, it will define its own ways for dealing with atomic events and variables.

Example 17 *The cumulative event from Example 9 (there given as a regular expression) can be given in SNOOP as*

$$\mathcal{A}^*(reg\_open(Subj), register(Subj, Stud), reg\_close(Subj)) \ .$$

The following markup binds the complete sequence to regseq *and the subject to* Subj*:*

```
<eca:rule ... >
  <eca:variable name="Subj" />
  <eca:variable name="regseq">
    <eca:event xmlns:snoopy="http://www.semwebtech.org/eca/2006/snoopy" xmlns:uni="...">
      <snoopy:cumulative>
        <!-- ignoring the identification of the AES formalism -->
```

```
        <uni:reg_open subject="{$Subj}"/>
        <uni:register subject="{$Subj}"/>
        <uni:reg_close subject="{$Subj}"/>
      </snoopy:cumulative>
    </eca:event>
  </eca:variable>
    :
</eca:rule>
```

(Note that the language could e.g. also provide redundant <snoopy:atomic> elements for being more explicit and providing means for identifying the AES formalism; see Section A.4.2.5.)

A.4.2.4 Embedding Algebraic Languages

The embedding of the event, query, test and action components is straightforwardly represented by a namespace change: outside, there is the eca: namespace, inside there is e.g. the SNOOP namespace, or an <eca:opaque> element that indicates the embedded language. In the same way, algebraic languages can be embedded into each other (e.g., embedding an expression of one event algebra as subexpression of another).

Example 18 (Embedding of Algebraic Languages) *Consider two event algebras, e.g., SNOOP and rCML (RuleCore Markup Language) [8]. An rCML expression can be embedded in a SNOOP expression as follows:*

```
<eca:rule ... >
    <eca:event xmlns:snoopy="http://www.semwebtech.org/eca/2006/snoopy">
      <snoopy:cumulative>
        <snoopy:or> ...</snoopy:or>
        <rcml:times xmlns:rcml="..."> ... </rcml:times>
        <snoopy:sequence> ...</snoopy:sequence>
      </snoopy:cumulative>
    </eca:event>
  </eca:variable>
    :
</eca:rule>
```

From the point of view of the ECA engine, the event part resides in the snoopy namespace. Thus, when registering this rule, the ECA engine will register the event component at a SNOOP engine. The SNOOP engine prepares the usual algorithm, in this case, for detecting a cumulative event pattern. The <snoopy:or> and <snoopy:sequence> subexpressions are also mapped to the local algorithm. The <rcml:times> subevent cannot be detected locally since the semantics is unknown to the SNOOP engine. Instead, it is registered at an rCML engine. Whenever the latter detects this event, it sends an appropriate <answers> message as described in Section A.3.2.3 to the SNOOP engine. This once more illustrates the use of generic communication schemata.

For embedding atomic events specifications, things are a bit more difficult since an atomic event specification involves a domain language/namespace and an AES formalism.

A.4.2.5 Embedding Atomic Events in Composite Events and Rules

For the surrounding language (which can be an event algebra of also the ECA-ML language in case of rules that react upon atomic events), an *atomic event specification (AES)* is a leaf element which does *not* belong to the surrounding namespace. The namespace border indicates the language border and initiates the handover between the "responsible" processors.

Since an atomic event specification involves a domain language/namespace and an AES formalism, one of these languages can be represented by the namespace of the root of the AEM XML

structure. In the above examples in Section A.4.2.2, this was always the domain namespace, but taking e.g. an XQueryX or Xcerpt XML markup of the subtree, the outer expression would be in the formalism's namespace.

The actual design of the markup is up to the composite event specification language: its interpreter must know where to submit the leaf atomic event description. This can be done either if the AES's namespace identifies the AES formalism, or if an intermediate element contains the necessary information. Additionally, the AES interpreter must be able to detect the domain namespace from the AES.
Note that in an RDF setting, the assignment of languages to an AES is simply done by two triples

(*aes*, has_domain, *domain*)
(*aes*, uses_formalism, *formalism*).

Explicit Markup Borders. The surrounding (i.e., composite event) language can specifiy that the leaves are surrounded by e.g. an explicit <atomic-event> element:

```
<cel:atomic-event cel:language="atomic event specification language AESL"
                  cel:domain= "domain of the event" >
    atomic event specification in formalism AESL
</cel:atomic-event>
```

Note that depending on the namespace of the inner element, either domain or language can be omitted.

Example 19 *A specification of an event that reacts upon a delayed-flight event and extracts some data can be embedded as follows:*

```
<eca:rule>
  <eca:event>
      :
      <cel:atomic-event
        cel:language="http://www.semwebtech.org/eca/2006/aes-xmlql"
        cel:domain="http://www.semwebtech.org/domains/2006/travel" >
        <travel:delayed-flight xmlns:travel="http://www.semwebtech.org/domains/2006/travel"
          flight="{$flight}" time="{$time}" />
      </cel:atomic-event>
      :
  </eca:event>
    :
</eca:rule>
```

In the same way as the surrounding language can provide redundant explicit elements that carry the language information, the markup of the *embedded* (AESL) formalism can provide this, e.g. by

```
<eca:rule>
  <eca:event>
      :
      <aes-xmlql:atomic-event
        xmlns:aes-xmlql="http://www.semwebtech.org/eca/2006/aes-xmlql"
        aes-xmlql:domain="http://www.semwebtech.org/domains/2006/travel" >
        <travel:delayed-flight xmlns:travel="http://www.semwebtech.org/domains/2006/travel"
          flight="{$flight}" time="{$time}" />
      </aes-xmlql:atomic-event>
      :
  </eca:event>
    :
</eca:rule>
```

Note that if the domain namespace can be derived from the inner markup it is not necessary to indicate it explicity (the Languages-and-Services Registry knows which namespaces are languages and which are domains).

```
<eca:rule>
  <eca:event>
    <aes-xmlql:atomic-event domain="http://www.semwebtech.org/domains/2006/travel"
      xmlns:xmlqlmatchns="http://www.semwebtech.org/eca/2006/aes-xmlql"
      xmlns:travel="http://www.semwebtech.org/domains/2006/travel">
      <travel:delayed-flight flight="{$flight}" time="{$time}"/>
    </aes-xmlql:atomic-event>
  </eca:event>
    :
</eca:rule>
```

We recommend the designers of such languages to support such explicit elements as *optional* markup.

No Explicit Element. Such an explicit surrounding element is often not necessary since there is a namespace change, and the language information can be given as an attribute in the subelement.

A.4.2.6 Example: SNOOP

As an example, a CED service based on the SNOOP [17] event algebra, extended with logical variables has been implemented in the prototype.
A DTD-style syntax description is as described below. It contains the usual simple combinators, additionally "any n out of *alternatives*", different kinds of periodic and aperiodic, optionally cumulative events. Operands can also be atomic events, or variables (whose contents are again algebra trees).

```
<!ENTITY % operand "(and | or | sequence | not |
   any | multi-occurrences |
   aperiodic | cumulative-aperiodic | periodic | cumulative-periodic |
   atomic-event | ANY-from-other-namespace |
   opaque |
   variable )">
<!ELEMENT and (%operand;, %operand;)>
<!ELEMENT or (%operand;, %operand;)>
<!ELEMENT sequence (%operand;, %operand;)>
<!ELEMENT any (%operand;)*>
<!ATTLIST any
    number-of-occurrences CDATA #REQUIRED>
<!ELEMENT multi-occurrences %operand;>
<!ATTLIST multi-occurrences
    number-of-occurrences CDATA #REQUIRED>
<!ELEMENT aperiodic (%operand;, %operand;, %operand;)>
<!ELEMENT cumulative-aperiodic (%operand;, %operand;, %operand;)>
<!ELEMENT periodic (%operand;, %operand;, %operand;)>
<!ELEMENT cumulative-periodic (%operand;, %operand;, %operand;)>
<!ELEMENT not (%operand;, %operand;, %operand;)>
<!ELEMENT atomic-event ANY>
<!ELEMENT variable (%operand;)>
<!ATTLIST variable name CDATA #REQUIRED>
```

```
<!ATTLIST extension for atomic-event and non-snoop subelements:
          xmlns:other-namespace CDATA #IMPLIED
          language CDATA #IMPLIED
          domain CDATA #IMPLIED>

<!ATTLIST extension for opaque:
          language CDATA #REQUIRED
          domain CDATA #REQUIRED>

<!ATTLIST extension for all elements except variable:
          variable CDATA #IMPLIED>
```

Language Identification for Nested Subexpressions. As discussed above, the SNOOP interpreter must be able to identify the languages of non-snoop subexpressions (atomic events or nested expressions from other CELs) accordingly. By default, this is done by the namespace used by these elements. Additionally, snoop:language (usually, expecting the namespace; but it can be allowed to use short names as "aliases") and snoop:domain can be given if needed. Note that both are required in case of opaque AESs (an XQuery AES in fact is an opaque AES).

Variable Bindings in Algebra Expressions. Variables can be bound to literal values or subtrees as discussed above in Section A.4.2.2. Furthermore, variables can be bound to algebraic subexpressions. Our SNOOP-style proposal provides different ways to handle variables on this level.

Example 20 *Consider the following event specification that should react if a b-event occurs after an a-event (by joined nr). The shortest form here does not require to use explicit <snoop:atomic-event> elements:*

```
<eca:rule...>
  <eca:event xmlns:dom="anydomain" ...>
    <snoop:sequence>
      <dom:a snoopy:language="http://www.semwebtech.org/eca/2006/aes-xmlql" nr="{$id}"/>
      <dom:b snoopy:language="http://www.semwebtech.org/eca/2006/aes-xmlql" nr="{$id}"/>
    <snoop:sequence>
  </eca:event>
    :
</eca:rule>
```

Assume now that for every such pair, the complete second event, i.e., the complete structure

```
<dom:b nr="x123" attributes> contents </dom:b>
```

should be bound to a variable var-x. This can be achieved by putting it inside a <snoopy:variable> element:

```
<eca:rule...>
  <eca:event xmlns:dom="anydomain" ...>
    <snoopy:sequence>
      <dom:a snoopy:language="xmlqlmatch" nr="{$id}"/>
      <snoopy:variable name="var-x">
        <dom:b snoopy:language="xmlqlmatch" nr="{$id}"/>
      </snoopy:variable>
    <snoopy:sequence>
  </eca:event>
    :
</eca:rule>
```

Then, /rule/event/seq/[1] addresses the first event specification, but the second one is not addressed by /rule/event/seq/*[1] but by /rule/event/seq/*[2]/*.*
Instead, adding the relationship to a variable as an attribute in the snoopy: markup would be equivalent and favorable:

```
<eca:rule. . . >
  <eca:event xmlns:dom="anydomain" . . . >
    <snoopy:sequence>
      <dom:a snoopy:language="xmlq|match" nr="{$id}" />
      <dom:b snoopy:language="xmlq|match" nr="{$id}" snoopy:variable name="var-x" />
    <snoopy:sequence>
  </eca:event>
    :
</eca:rule>
```

Then, /rule/event/seq/[1] and /rule/event/seq/*[2] address the atomic event specifications. Both atomic events specifications also include the necessary information about the used formalism (by attribute) and the domain (implicit by namespace).*

A.4.2.7 Related Work and Existing (Sub)languages

Detection of Composite Events.

The internals of the event detection engine are then concerned with implementing the semantics of the event combinators, which can be done in different ways:

- Operators as Classes – (cf. RuleCore [8]),
- Tree and event queries – (cf. SNOOP [17], XChange [15]),
- Automata and Petri Nets – ODE and SAMOS
  (here, also a representation of automata states in XML and transformations by XSLT can be used),
- RDF: describe how an event description transforms into another upon an atomic event. This semantic solution would require an ontology of event combinators, but allows then for a very high-level specification of rules.

Existing (sub)languages.

Especially, existing tools can be employed in a service-oriented architecture:

- XChange's event query mechanism [15]. Then, the event component is e.g. marked up by

  ```
  <eca:rule xmlns:banking="http://www.banking.nop"
       xmlns:temporal="http://www.some.webservice" >
    <eca:event xmlns:xchange="http://xcerpt.org/xchange" >
      <!-- xchange fragment in opaque or xml-markup form -->
    </eca:event>
      :
  </eca:rule>
  ```

- RuleCore [8] is an ECA system that provides an event detection component where new operators can be added by via appropriate classes. Then, RuleCore can be used to implement and experiment with arbitrary event algebras.
- Xcerpt [16] can be used as an AESL.

Composability: Embedding Different Event Algebras

Note that with the above design, it is also possible to embed terms (events) of an event algebra into terms of another one: the namespace identifies the subterm as an algebra expression, and the whole subterm is sent to the responsible processor that in turn answers with a "detected event" message as usual.

A.4.3 Architecture and Communication: ECA, CED, and AEM

Event processing is done in cooperation of an ECA engine, one or more *Composite Event Detection Engine (CED)* that implement the event algebras, and one or more *Atomic Event Matchers (AEM)* that implement the *Atomic Event Specification Languages (AESL)*. The architecture part that is relevant for handling events is shown in Figure A.4.2.
The event *detection* then happens when actual events occur. Here, an *independent* and an *independent* variant are considered:

Independent: here, the owner of the rule does not supply the events (e.g., when a user defines a rule for travel planning). In this case, the AEM is responsible for being informed about relevant events.

Dependent: here, the owner of the rule also supplies all relevant events. This is e.g. the case when an application service (e.g. an airline) defines rules for their own business that use only the events where the airline is aware of.

The independent version works as follows:

- The ECA engine registers the event component at an appropriate CED (composite component) [or AEM (only atomic event)] service,
- the CED registers the AESs at appropriate AEMs that implement the AESLs,
- the AEMS are assumed to be informed about all relevant events in the format given in Section A.4.2.1. Details about how this happens are described in Section A.5.6.1.
- The domain nodes or event brokers forward relevant atomic events to the AEMs,
- the AEMs match them against the (more detailed) specification and inform the CEDs,
- the CEDs process them and inform the ECA engine in case that a composite event has been detected.

A.4.3.1 Abstract Semantics and Markup of Event Detection Communication

The above considerations show that the communication with CED services and AEM services uses the same patterns. The all communication ECA→CED, ECA→AEM, CED→CED (nested) and CED→AEM follows patterns described in Section A.4.1:

- Downwards: Reply-To, Subject/identifier of request, fragment, optional: variable bindings (depends whether the service can handle multiple bindings, see Section A.6.1.2).
- Upwards: Subject/identifier, functional result (matched event (sequence)) with optional tuples of variable bindings.

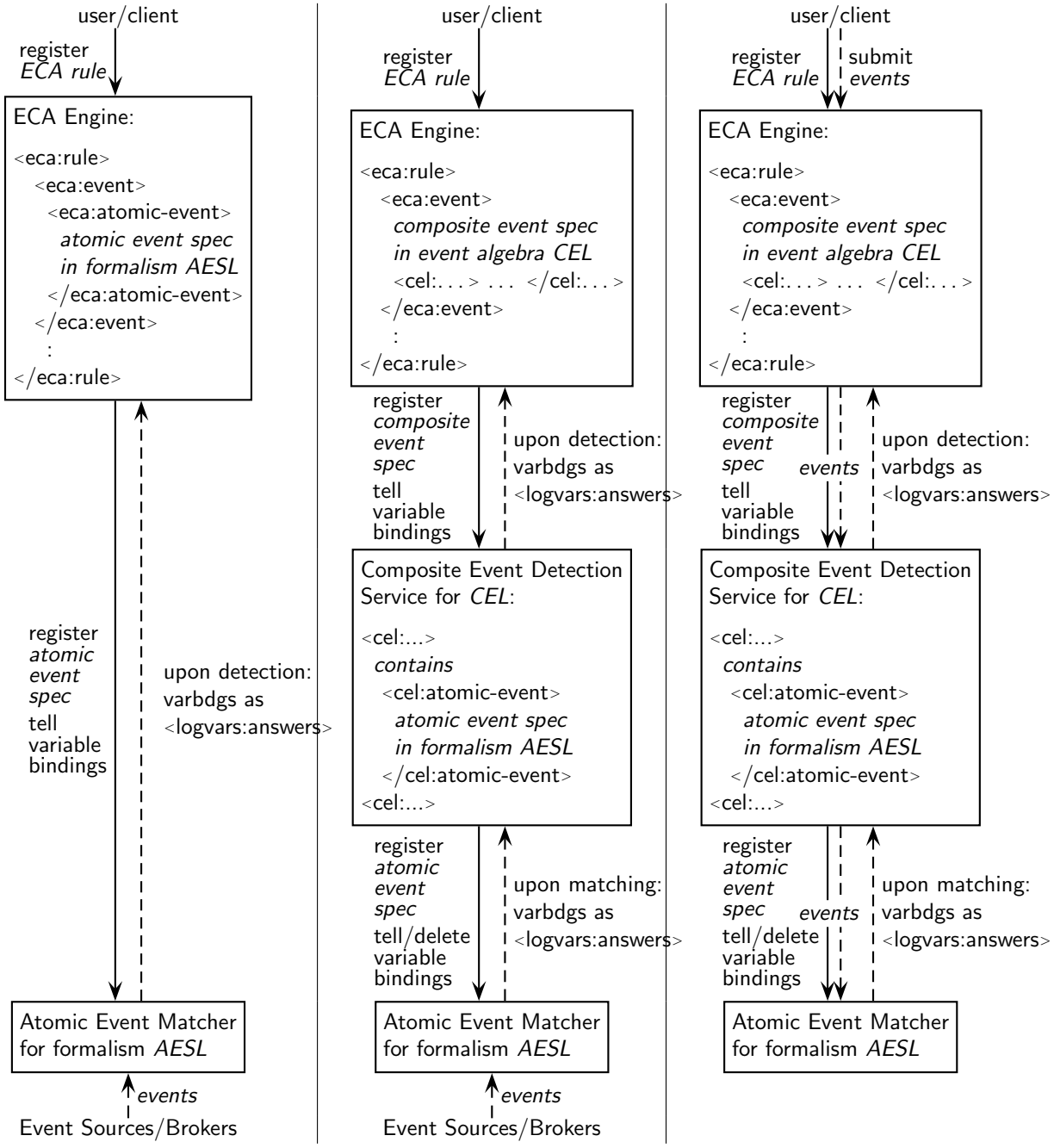


Figure A.4.2: Architecture: Processing Event Components and Events (left and middle: independent; right:dependent)

The responses of the event component services must match the format for variable bindings given in Section A.3.2.3. Moreover, the answer from detecting an atomic event should have the same format as when detecting a composite event for the following reasons:

- conceptual cleanliness: both are "detected events",
- architecture: the event component of a rule may be either a composite event or an atomic one. Having different answer formats would require more work. The same holds for allowing

embedded event algebra expressions into another event algebra expression.

As usual in the literature about Active Databases, the "semantics" of an event expression is usually not only "yes/no", but the sequence of atomic events that contributed to the detection. Event algebras usually did not use free logical variables with join semantics, but we do it. Thus, here we have a case where the semantics consists of

- a functional result, and
- for each such result (in this case, it is only one), a set of tuples of variable bindings (which actually can be several ones if the sequence allows for several matches).

A.4.3.2 Communication for Event Components between ECA and CED

When a rule is registered at the ECA engine, the ECA engine registers the event component at an appropriate service. Here we consider only composite event components (atomic ones are dealt with in the following section since their communication is analogous to the one between CED and AEM). Consider the following generic example:

```
<eca:rule. . . >
  <eca:event>
    <evt:foo xmlns:evt="URI of the 'evt' event language" >
      specification of the event component
    </evt:foo>
  </eca:event>
    :
</eca:rule>
```

The language used the event component is identified by the namespace of the event subexpression. How this service identification throughout the Web is done exactly is described in Chapter A.6 – the result is a URL where the request has to be sent to and some minor directives how it should be sent (HTTP, SOAP) and wrapped (header, body, hull element).
The above abstract semantics is e.g. communicated in XML as

```
to: service-URL of the CED
<register>
  <Reply-To>URL where the answer is expected (at the ECA engine) </Reply-To>
  <Subject>identifier of the request –
      in most cases the RDF URI of the event component</Subject>
  <!-- next the event component -->
  <evt:foo xmlns:evt="URI of the 'evt' event language" >
    specification of the event component
  </evt:foo>
  <!-- and optionally the existing variable bindings -->
  <logvars:variable-bindings xmlns:logvars="http://www.semwebtech.org/lang/2006/logic" >
    variable bindings
  </logvars:variable-bindings>
</register>
```

The upwards communication is even simpler: the result is sent to the address originally given as Reply-To. The agreed subject is used, and the <answers> or a single <answer> element is submitted:

```
to: Reply-To-address from the request that is answered
<anyname>
  <Subject>identifier of the request</Subject>
  <!-- next the event component -->
```

```
  <logvars:answers xmlns:logvars="http://www.semwebtech.org/lang/2006/logic">
    answers
  </logvars:answers>
</anyname>
```

If the subject is sent in the HTTP header, also the surrounding <anyname> element can be omitted.

A.4.3.3 Communication for Event Matching with AEMs

Similar to the ECA-CED communication, it is asynchronous and consists of *registering* relevant CESs (downwards) and reporting occurrences (upwards). In case that the event component of a rule ist just atomic, there is a direct communication between ECA and AEM; otherwise the CED communicates with the AEM (note that this induces that the upward communication from the AEM to CED or ECA should be the same as from the CED to the ECA, using the answers format described in Section A.3.2.3). In the following we assume a communication from the CED with the AEM, which subsumes the ECA-AEM case.

The proceeding is the following: The CED selects all AESs (=leaf expressions) inside the CES and registers each of them at an appropriate AEM. We here assume that the AEM is aware of all (relevant) events (this will be described in more detail in Section A.5.6.1) and notifies the CED upon occurrence of an event matching a registered AES.

Registration of an AES. If the ECA engine or any composite event detection engine is interested in being informed about occurrences of atomic events wrt. some AES, it sends a message with the following contents to an appropriate atomic event detection service:

- Reply-To, Subject/identifier of request, the AES (as XML fragment or URI reference), optional: variable bindings.

The overall format is the same as when registering an CES at a CED:

```
to: service-URL of the AEM
<register>
  <Reply-To>URL where the answer is expected (at the ECA or CED engine) </Reply-To>
  <Subject>identifier of the request –
      in most cases the RDF URI of the AES wrt. event component</Subject>
  <!-- next the AES -->
  the AES fragment
  <!-- and optionally the existing variable bindings -->
  <logvars:variable-bindings xmlns:logvars="http://www.semwebtech.org/lang/2006/logic">
    variable bindings
  </logvars:variable-bindings>
</request>
```

Occurrence Indication of Relevant Events. When such an event is actually detected, it is immediately reported in an answer by the AEM that uses the format described in Section A.3.2.3 by logvars:answers, logvars:answer, logvars:result and logvars:variable-bindings.

Note that if multiple occurrences are detected together, more than one logvars:answer elements can be sent in one logvars:answers element (although event notifications should always be sent as soon as possible, it is possible that one event leads to multiple occurrences of a specified atomic event).

Example 21 *Consider that notifications of delayed flights are published all 10 minutes as a list. The domain broker for travel: gets this information and processes it.*

```
<msg:receive-message sender="service@fraport.com">
  <msg:content>
    <travel:delayed-flight flight="LH1234" minutes="30"/>
    <travel:delayed-flight flight="AF0815" minutes="90"/>
      :
    <travel:canceled-flight flight="AL4711"/>
    :
  </msg:content>
</msg:receive-message>
```

Consider the following rule at an ECA engine:

```
  <eca:rule>
    <eca:event>
      <eca:atomic-event language="http://www.semwebtech.org/eca/2006/aes-xmlql">
        <travel:delayed-flight flight="{$flight}" minutes="{$minutes}"/>
      </eca:atomic-event>
    </eca:event>
      :
  </eca:rule>
```

The ECA engine registers the atomic event at an appropriate atomic event matcher (that in turn will contact the travel service to be informed about relevant events), see Section A.4.2.

Later, an answer upon arrival of the above message from the airport will be of the following form:

```
<logvars:answers subject="rule-id/event">
  <logvars:answer>
    <logvars:result>
      <travel:delayed-flight flight="LH1234" minutes="30"/>
    </logvars:result>
    <logvars:variable-bindings>
      <logvars:tuple>
        <logvars:variable name="flight">LH1234</variable>
        <logvars:variable name="minutes">30</variable>
      </logvars:tuple>
    </logvars:variable-bindings>
  </logvars:answer>
  <logvars:answer>
    <logvars:result>
      <travel:delayed-flight flight="LH1234" minutes="30"/>
    </logvars:result>
    <logvars:variable-bindings>
      <logvars:tuple>
        <logvars:variable name="flight">AF0815</variable>
        <logvars:variable name="minutes">90</variable>
      </logvars:tuple>
    </logvars:variable-bindings>
  </logvars:answer>
</logvars:answers>
```

Note that in an RDF environment, there would be a URI reference for LH1234 and AF0815.

Alternative: Sideways Information Passing for Atomic Event Detection

In case that composite events consist of multiple atomic events that share logical *join* variables during evaluation of a composite event, in the same way as applying a *sideways information passing strategy* in query evaluation nand rule evaluation, variable bindings obtained by "earlier" atomic events can be used for constraining the relevant event occurrences.

Example 22 (Join Variables in Composite Events) *Consider again Example 17 which uses a cumulative SNOOP event with a join variable $Subj:*

```
<snoopy:cumulative xmlns:snoopy="http://www.semwebtech.org/eca/2006/snoopy"
    xmlns:uni="...">
  <uni:reg_open snoopy:language="xml-ql-match" subject="{$Subj}"/>
  <uni:register snoopy:language="xml-ql-match" subject="{$Subj}"/>
  <uni:reg_close snoopy:language="xml-ql-match" subject="{$Subj}"/>
</snoopy:cumulative>
```

Here, when the event component is registered at Snoop, it can do the following:

- *it can immediately register all three atomic event patterns at the XML-QL-Matcher service. Assume that then reg_open("Databases") is detected. Later, the AEM will report for register("Scott Tiger", "Databases"), but also for register("John Doe", "Algorithmics") for that rule. The Snoop engine must then apply the join semantics.*
- *it can register only the first event pattern reg_open($Subject). If then, reg_open("Databases") is detected and reported, Snoop registers next register(_,"Databases") which will only report registrations for databases, not for other courses. (The same can be achieved by registering once and disabling/enabling event patterns at the AEM.)*

A.4.3.4 Identification of an AEM for an Atomic Event Specification

As stated above, at the language border to the AES it must be possible for the processor of the surrounding language to identify which service is responsible for processing the AES. Having an element that is suspected to be an atomic event description (i.e., it leaves the namespace from the surrounding language, or it is an opaque element), the processor determines the relevant service as follows:

- Check if the namespace of the first element "behind" the language border is an AESL (which can be found out in the language&services dictionary, see Section A.6.5.1). If yes, use it.
- Check if the first element "behind" the language border has a language attribute (which is not in the domain namespace of the element). If this language is an AESL, use it. (note that this covers explicit wrapping elements and also allows to add this information just as a shortcut to e.g. the element to be matched).
- Check if the last element "before" the language border (note that this is "the" language of the currently processing service - so it is directly understood) is a wrapping element for atomic events and contains information about the embedded language (e.g. in a language attribute). If yes, use it.
- Determine the domain of the first element "behind" the language border. If the domain broker for that domain supports an AED mechanism, then submit the AED directly to the domain broker.
- If the element is an <opaque language="*language*"> element, then identify the responsible service (which can be the language service or a domain broker which is automatically aware of relevant events). Note that the opaque can either belong to the surrounding namespace or to the namespace of the embedded language.

- Otherwise apply suitable heuristics or return an error message.

Note that a language border in a fragment of an event algebra language does not necessarily lead into an AESL, but can also lead to an embedded event algebra language.

A.4.3.4.1 Example

Example 23 *The following specifies, in an illustrative, non-normative (XML) markup, an event for (very simplified) detection of a late train. It is a composite event specification in the SNOOP (algebraic) language, and uses atomic events from messaging and the domain of train travels. The detection of late trains is made either by being warned by mail the travel agency, or by the occurrence of a domain-specific event signaling changes in a given (pre-defined) source with expected arrival times:*

```
<eca:rule xmlns:msg="http://www.messages.msg/messages"
      xmlns:travel="http://www.semwebtech.org/domains/2006/travel">
  <eca:event>
    <snoopy:disjunctive xmlns:snoopy="http://www.semwebtech.org/eca/2006/snoopy">
      <snoopy:atomic language="xml-ql-match">
        <msg:receive-message sender=$myTravelAgent">
          <msg:content>
            <travel:delayed-train train=$myTrain arrivalTime=$newArrival/>
          </msg:content>
        </msg:receive-message>
      </snoopy:atomic>
      <snoopy:atomic>
        <aes-xpath:event xmlnsaes-xpath:"http://www.semwebtech.org/eca/2006/aes-xpath">
          <aes-xpath:test cond="$event/name()='travel:delayed-train'"/>
          <aes-xpath:test cond="$event/@train=$myTrain"/>
          <aes-xpath:variable name="newArrival" select="$event/@arrivalTime"/>
        </aes-xpath:event>
      </snoopy:atomic>
    </snoopy:disjunctive>
  </eca:event>
  <eca:action> an action specification in any markup
  </eca:action>
</eca:rule>
```

When a customer registers the rule, the values for the variables myTrain *and* myTravelAgent *have to be supplied. The ECA rule engine registers the whole* event *component at the SNOOP service (identified by the URL of the* snoopy *namespace):*

- *Reply-To: ECA engine (exact URL where the reponse should arrive)*
- *Subject/task identifier:* the-rule-id/event
- *the whole event component, i.e., the* snoopy:disjunctive *element.*
- *a* logvars:variable-bindings *element that contains the bindings for* myTrain *and* myTravelAgent.

Snoop parses the tree as far as it belongs to its namespace. The composite event is an "or" of two atomic events which are not "understandable" to Snoop.

For the first one, language="xml-ql-match" *is given which can be looked up in the Languages&Services Registry for identifying a service. The whole AES is then sent to* xml-ql-match-service, *together with the bindings of the variables* myTrain *and* myTravelAgent:

- *Reply-To: me*

- *Subject/task identifier:* the-rule-id/event/disj/*[1]
- *the whole event component, i.e., a* snoopy:disjunctive *element.*
- *optional (if the* xml-ql-match-service *supports variables): a* logvars:variable-bindings *element that contains the binding for* myTrain *and* myTravelAgent*; otherwise the value must be inserted as string into the code fragment.*

The AES describes the receipt of a message (marked-up in XML) with an attribute sender *which is equal to the value of the variable* myTravelAgent*, and with a* content *with a* delayed *element with an attribute* train *coinciding with that of* myTrain*. After detecttion, the variable* newArrival *will be bound to the value of the attribute* arrivalTime *of the* delayed *element.*

*The second AES is identified by its namespace "*navigation-based-formalism-uri*" where also a service URL can be looked up. The whole AES is sent there, this time, only the bindings of the variable* myTrain *are required. The AES specifies a domain-specific event* travel:delayed-train *(that occurs "somewhere in the Web" and has to be detected by Semantic Web mechanisms). The event is implicitly bound to* $event*. The details are then checked by XPath expressions against* $event*: If its attribute* train *equals the value of the variable* myTrain*, then* newArrival *is bound to the value of the* newTime *attribute of the event.*

Both AEM services will scan all events that they are aware of for matching, and in case of success they will return an answer (according to the format given in Section A.3.2.3).
Assume that the second AEM becomes aware of an event

```
<travel:delayed-train xmlns:travel="http://www.semwebtech.org/domains/2006/travel"
      train="ICE 773" plannedArrival="11:30" arrivalTime="13:00" /> .
```

Then, it responds to the Snoop service with a message

- *task identifier:* the-rule-id/event/disj/*[2]
- *contents:*

```
<logvars:answer>
  <logvars:result>
    <travel:delayed-train xmlns:travel="http://www.semwebtech.org/domains/2006/travel"
        train="ICE 773" plannedArrival="11:30" arrivalTime="13:00" />
  </logvars:result>
  <logvars:variable-bindings>
    <logvars:tuple>
      <logvars:variable name="myTrain">ICE 773</variable>
      <logvars:variable name="newArrival">13:00</variable>
    </logvars:tuple>
  </logvars:variable-bindings>
</logvars:answer>
```

Snoop will then evaluate the semantics of the discjunctive event, which is detected if one alternative has been detected and sends a message to the ECA engine:

- *task identifier:* the-rule-id/event
- *contents:*

```
<logvars:answer>
  <logvars:result>
    <travel:delayed-train xmlns:travel="http://www.semwebtech.org/domains/2006/travel"
      train="ICE 773" plannedArrival="11:30" arrivalTime="13:00" />
```

```
    </logvars:result>
    <logvars:variable-bindings>
      <logvars:tuple>
        <logvars:variable name="myTrain">ICE 773</variable>
        <logvars:variable name="myTravelAgent">...</variable>
        <logvars:variable name="newArrival">13:00</variable>
      </logvars:tuple>
    </logvars:variable-bindings>
  </logvars:answer>
```

Note that upon receipt of the message at the ECA service, the event sequence itself is not bound to a variable (since it is not indicated) – all relevant information can be extracted without this.

Example 24 *In Example 17, we illustrated SNOOP's cumulative event: registration for an exam begins, students register and the registration is closed. The returned message in this case from the Snoop service could e.g. contain the following answer:*

```
<logvars:answer>
  <logvars:result>
    <uni:reg_open subject= "Databases"/>
    <uni:register subject= "Databases" name= "John Lennon"/>
    <uni:register subject= "Databases" name= "Paul McCartney"/>
    <uni:register subject= "Databases" name= "George Harrison"/>
    <uni:register subject= "Databases" name= "Ringo Starr"/>
    <uni:reg_close subject= "Databases"/>
  </logvars:result>
  <logvars:variable-bindings>
    <logvars:tuple>
      <logvars:variable name="Subj">Databases</variable>
    </logvars:tuple>
  </logvars:variable-bindings>
</logvars:answer>
```

For collecting all names, the event component of the rule is contained in an <eca:variable name= "regseq"> element that assigns the returned event sequence to that variable. The names of the registered students can then be extracted by a query.

A.4.3.5 The Query Component

There are a lot of query languags for XML or RDF data around, such as XQuery (where an XML markup proposal has been presented in [65]), F-Logic, RDFQL, XPathLog, or Xcerpt. Note that queries that bind (join) variables can already be used for restricting the results.

Queries that are only concerned with the contents a certain database node can use the query language of this node, often in an opaque way. In all cases, the result of evaluating queries is communicated in the usual form described above.

A.4.3.6 Opaque Queries

For the development of a prototype of the framework we rely on *opaque* queries; see Section A.7.1.2.

A.4.3.7 Atomic Queries

Analogously to atomic events or actions, atomic queries refer to atomic notions of the domain ontologies. In algebra-based query languages like SQL or Description Logic queries, the atomic queries are the leaf expressions (table names, concept names, property names) which are combined by operators.

Relational Model/SQL/Datalog Concerning the relational model and the relational algebra or calculus, atomic queries are the relation names (i.e., the leaves in algebra trees), or predicates (in conjunctive queries in the relational calculus and Datalog). The "algebraic" language is then the relational algebra, SQL, or simple conjunctive queries as in Datalog.

RDF. For the RDF model, atomic queries are triples. The "algebraic" language is then provided by SPARQL etc.

XML. For XML, this notion does not exist so clearly. XQuery FLWR expressions are obviously not atomic. Are XPath expressions atomic queries? According to the algebra evaluation, they are not. Atomic here corresponds to decomposing XPath expressions in single steps and expressing an XPath expression as a conjunctive query (as e.g. done when mapping it to F-Logic).

Since there is not yet a favorite Semantic Web query language, there are numerous proposals and interpretations.

A.4.3.8 Composite Queries

Composite queries can be expressed in two ways:

- by query algebras over atomic query expressions, or
- logic-based.

The result of the evaluation of an expression is returned in the format described in Section A.3.2.3.

GG(M)QL: Generic Graph Query (Markup) Language. An XML Markup can be defined based on algebraic operations. For instance, a tree notation of the relational algebra (providing domain-independent operations) with (domain-dependent) relations as atomic expressions provides an example of a tree-structured query component.

Dealing with graph-shaped data, an extended algebra is recommended, e.g., in the style of the one used for the implementation of F-Logic [33, 34] in the Florid system [24] or with some adaptations for XPathLog [42, 43] in the LoPiX system [41].

In addition to the operations known from the relational algebra, i.e., "union", "intersection", "difference" (as derived operation), "selection", "projection", "join" and "division" (derived), a "navigation" operation is common for graph or tree data. Additionally, "grouping/aggregates" (leading to denormalized data) should be provided.

We follow here the definition of the formal semantics of F-Logic as e.g. given in [38, Section 2.2.1]. The (preliminary) DTD is as follows:

```
<!ENTITY % operand "(... |  ANY-from-other-namespace | opaque)">
<!ELEMENT union (%operand;, %operand;)>
<!ELEMENT intersect (%operand;, %operand;)>
<!ELEMENT difference (%operand;, %operand;)>
<!ELEMENT select (%operand; condition)>
<!ELEMENT project (%operand;)>
  <!ATTLIST project variables NMTOKENS #REQUIRED>
<!ELEMENT join (%operand;, %operand;)>
<!ELEMENT navigation (%operand;, %operand;?)>
  <!ATTLIST navigation
            axis (xml-axes) #IMPLIED
            method-type (f-logic method types) #IMPLIED
            nodetest (xml nodetest) #IMPLIED
            property (nmtoken) #IMPLIED
            variable (variable-name) #IMPLIED >
```

```
<!ELEMENT condition (any-test-language-expression)>
```

- navigation covers F-Logic, XML and RDF. The optional parameters correspond to the XML "axis", the F-Logic "method type", the XML nodetest and the RDF/F-Logic property name. Optionally, the nodetest/property can be replaced by a second operand for *computed* names (as in F-Logic, XPathLog, or SPARQL). In this case, the answer part of the result of the second operand acts as property name, the variable bindings are joined. The variable attribute indicates that the result of the navigation should be bound to a new variable or joined with an existing one.
- condition: use an "external" test language (i.e., a subexpression of another namespace), or an opaque statement. It is recommended that a query language provides a set of predicates such as equal, <, >, substring etc. as in XPath locally.

Markup of an RDF-based Query Language. For *reasoning* about rules on the RDF level, it is preferable that the markup of queries is as close as possible to the RDF model itself.

- RDF patterns (XML markup: in the style of RDF/XML). Note that the normal form of this are tripels. RDF patterns are sufficient for positive queries; for negative ones, additional syntax like in F-Logic, XPathLog, or Xcerpt has to be defined.
- a logic based on tripels like SPARQL with the RDFS/OWL built-in semantics.

A.4.3.9 Result Semantics

Query languages can have a result of the following types:

- functional: just a set of things
- logical: a set of tuples of variable-bindings
- functional+logical: format as described in Section A.3.2.3.

When dealing with the ontology of languages (see Section A.6), the type of result semantics must be given.

A.4.4 The Test Component

According to the distinction between the query component (obtaining information) and the test component (which is only allowed to use existing information), evaluating the test means evaluating a condition. A condition is a formula, its composers are the boolean operators, quantifiers etc. *Atomic* tests are thus only domain-independent tests, i.e., equality, comparisons like "<" and ">" etc. Often, the test part can already be included into the query part if the query language allows for conditions, filters, or even in form of join conditions in conjunctive queries.

Tests can be expressed formulas in any logic, e.g., First-Order logic (where an XML markup is given in [10]), F-Logic, XPath-Logic, or XQuery or Xcerpt (note that in these languages, expressions that have an empty result also yield the truth value "false").

For tests, similar considerations as for queries apply. In contrast to the query part, tests cannot bind variables and their result is only true or false.

- Join/restriction semantics: tuples of variable bindings are communicated downwards, and the test service returns only those tuples that pass the test. These tuples are then joined with the bindings on the ECA level.
- functional semantics: tuples of variable bindings are communicated downwards, and the test service labels each tuple with true/false.

Test expressions are thus composite expressions over atomic predicates (that can be "simple" classical predicates, or pattern terms as in F-Logic or Xcerpt).

Simple tests (comparison predicates etc.) can usually be evaluated locally at the ECA engine. For this, also the simple boolean algebra operators "and", "or" and "not" are used.

A.4.4.1 Test Component Languages

In contrast to queries, tests work on a set of tuples of (input) variable bindings. They cannot bind additional (non-local) variables (local variables may be used in subqueries).

Boolean Logic. The simplest tests only use atomic tests and combine them by boolean operators, i.e., "and", "or" and "not". For the markup, see below. The (preliminary) DTD is as follows:

```
<!ENTITY % operand "(... |  ANY-from-other-namespace | opaque)">
<!ELEMENT and (%operand;, %operand;)>
<!ELEMENT or (%operand;, %operand;)>
<!ELEMENT not (%operand;)>
<!ELEMENT condition (any-test-language-expression)>
```

The operators can be evaluated according to their usual boolean semantics by verifying them (truth-table style), or relationally: and is actually a join or cartesian product, or is union (care for same set of bound variables), and not is actually a set difference. Quantifiers are not allowed.

Quantifiers and Grouping and de-grouping. A test can be evaluated to each tuple individually, or for a group of tuples. In the latter case, quantifiers can be applied to each group.

Example 25 *Consider the following situation: the input consists of a set of pairs (X, Y) where for each X, multiple pairs are allowed. Those X where all corresponding Ys are > 100, or at least one Y is > 1000, should be continued, the other ones are discarded.*

```
<group-by variable=‘‘X’’>
  <or>
    <all variable=‘‘Y’’>
      <fn:greater-than>
        <variable>Y</variable>
        100
      </fn:greater-than>
    </for-each>
    <exists variable=‘‘Y’’>
      <fn:greater-than>
        <variable>Y</variable>
        1000
      </fn:greater-than>
    </exists>
  </or>
</group-by>
```

(Note that similar behavior can be expressed also by the action part with alternative branches, grouped by X)

Note that, given a variable X, any test of the form $\forall X : (p(X, Y) \rightarrow q(Y)$ actually involves a query, namely to obtain all Y such that $p(X, Y)$ holds. Thus, this *should* be done in a previous query step. Moreover, "for all" is critical when an open world is assumed.

Often, it is preferable to express "typical" predicates in the domain ontologies as derived properties.

A.4.4.2 Atomic Tests: Predicates

A test language can support built-in predicates that are evaluated locally. Additionally, domain-specific predicates and terms do not can be used in the test. The latter have to be evaluated by calling a corresponding domain broker. To save this additional communication overhead, it is recommended to state already the qury as strict as possible.

Example 26 *Consider the following simple rule: a person P books a travel to city C. The person has a contract with a very attentive car rental company* WorldWidePersonalizedMobility (WWPM) *that provides a highly personalized service. Customers can register for different profiles, e.g., "H" (habit) customers are offered a car of the same size as the person owns at home. For this, the car rental company maintains a database about which cars the person owns, and how much the person is willing to pay. If an appropriate car is available at the target city, the most expensive one is reserved (WWPM wants to make profit, and to offer the customers the "highest" choice).*

- *Rule "H":*
- *event: person P books a flight to C.*
- *initial query: person has profile "H"?*
- *next queries: which cars X are available at P for which price PX? After that query, all values that are potentially needed for the action are obtained.*
- *the test could be the following: test whether X is of the same size as any car that P owns, and costs below the maximal amount A that P is willing to pay.*
- *the action then consists of booking the "best" one.*

The test requires to access the underlying database and to evaluate a comparison. In the normal form, the queries and test are as follows (e.g., using XPath)

```
<eca:rule xmlns:travel="http://www.semwebtech.org/domains/2006/travel">
  <eca:event language="xml-ql-match">
    <travel:booked-flight name="{$P}" to="{$City}" />
  </eca:event>
  <eca:query variable="$OwnCar">
    <eca:opaque language= "XPath">
      /customers/customer[@name="$P"]/owncar/@type
    </eca:opaque>
  </eca:query>
  <eca:query variable="OwnClass">
    <eca:opaque language= "XPath">
      /types/type[@name="$OwnCar"]/@class
    </eca:opaque>
  </eca:query>
  <eca:query variable="$MaxPrice">
    <eca:opaque language= "XPath">
      /customers/customer[@name="$P"]/@maxprice
    </eca:opaque>
  </eca:query>
  <eca:query>
    <eca:use-variable name="$City" use="City" />
    <eca:opaque language= "XPathLog">
      /branches/branch[@city→$City]/cars[@type→Type and @price→Price] and
        /types/type[@name→Type and @class→Class]
    </eca:opaque>
```

```
  </eca:query>
  <eca:test xmlns:test="http://www.semwebtech.org/eca/2006/test">
    <test:and>
      <test:equals><test:variable>$OwnClass</test:variable><test:variable>Class</test:variable></test:equals>
      <test:less-equal><test:variable>Price</test:variable><test:variable>$MaxPrice</test:variable></test:less-equal>
    </and>
  </eca:test>
</eca:rule>
```

A.4.5 The Action Component

The action component actually enforces the consequences of an ECA rule. The action component is either an atomic action of an application domain, or a specification of a composite action, given in a *Composite Action Language (CAL)* implemented by *Composite Action Exection Engines (CAEs)*. For executing actions, two scenarios can be distinguished:[1]

- Rules can implement *global* behavior (general dynamic integrity constraints like policies in an application domain). Here, it is *not* clear which nodes should actually execute the action. In this case, "action brokers" are required that control the communication with the respective domain nodes; see Sections A.5.6.5.

- Many rules are registered by maintainers of domain nodes to implement reactions on events somewhere in the Semantic Web that also use data obtained by queries against the Semantic Web, but whose actions should be executed in the *own* domain node.

  For dealing with the first case, we propose to extend the action component with an attribute eca:execute-at whose value can be either a pseudo-variable "$owner" (the owner of the rule) or a hardcoded URL:

```
<eca:rule >
  <eca:event> ... </eca:event>
    :
  <eca:action execute-at="domain node identifier">
    action specification
  </eca:action>
</eca:rule>
```

  We propose that CALs also provide a similar means to add such information to atomic actions [the user can see if this is provided from the ontology metadata of the CAL].

A.4.5.1 Atomic Actions

Atomic actions can do the following:

- invoking actions of domain nodes (see Chapter A.5) by framework-native mechanisms,

- opaque actions that invoke arbitrary Web Services (by HTTP or SOAP messages),

- implement ECE rules: their action component is usually atomic and just raises an event in an application domain.

[1] probably there are even more ...

Invoking Actions at Framework-Aware Nodes. For executing the action for a given rule instance, the request contains the contents of the action component and the variable bindings given in the format described in Section A.3.2.2).

The request is sent to an action broker (cf. Section A.5.6.5) that identifies the actual receiver of the request (using the data contained in the request). Note that for a single such message with several variable bindings, *different* application nodes may be finally responsible. The selection is done by the action broker.

Example 27 (Canceling flights due to bad weather) *Consider the following case: at some airport the weather conditions are forecasted to become bad, so that incoming flights cannot land. There is then a rule "if the forecast is ... then for all flights landing in the afternoon, cancel these flights". All flights that are concerned can easily be selected from the flight schedule. Since these are operated by different airlines, the actions must be executes at different application nodes (that can in an RDF world be determined from the association between flight numbers and the corresponding airlines).*

```
<eca:request>
  <eca:action>
    <eca:input-variables names="flight reason" />
    <travel:cancel-flight flight=$flight>
      <travel:reason>$reason</travel:reason>
    </travel:cancel-flight>
  </eca:action>
  <logvars:variable-bindings>
    <logvars:tuple>
      <logvars:variable name="flight">LH123</logvars:variable>
      <logvars:variable name="reason">bad weather</logvars:variable>
    </logvars:tuple>
    <logvars:tuple>
      <logvars:variable name="flight">AL400</logvars:variable>
      <logvars:variable name="reason">bad weather</logvars:variable>
    </logvars:tuple>
  </logvars:variable-bindings>
</eca:request>
```

See Example 39 for the actual handling of that task by the Domain Broker and the airline nodes.

Opaque Actions. Opaque actions are similar to opaque queries. They contain a code fragment that has to be executed by some service.

- the code fragment is part of an URL to be appended to the service URL:

  ```
  <eca:rule>
      :
    <eca:action>
      <eca:opaque uri="domain-service-url">
        <eca:input-variables names="flight reason" />
        cancel-flight($flight,$reason)
      </eca:opaque>
    </eca:action>
  </eca:rule>
  ```

  will call e.g.

  domain-service-url:cancel-flight("LH123","bad weather")

- the code fragment is just an update operation, e.g. an SQL statement DELETE FROM *table* WHERE *condition*. In this case, the URL where the action has to be sent must be given in the opaque element. (The case is similar to submitting an XPath query to a given database.)

```
<eca:rule>
    :
  <eca:action>
    <eca:opaque uri="domain-service-url">
      <eca:input-variables names="flight reason" />
      DELETE FROM orders WHERE customerNo = 123
    </eca:opaque>
  </eca:action>
</eca:rule>
```

- the code fragment is a code fragment in some programming language that contains the URLs to be updated. In this case it can be evaluated by a "free" language service. (The case is similar to evaluate an XQuery query of the form FOR $x document(*url*).)

```
<eca:rule>
    :
  <eca:action>
    <eca:opaque lang="domain-service-url">
      <eca:input-variables names="flight reason" />
      DELETE FROM orders WHERE customerNo = 123
    </eca:opaque>
  </eca:action>
</eca:rule>
```

In all cases, the variable occurrences in the expressions are replaced in the same way as for opaque (HTTP) queries. Again, a wrapper can be used that provides a framework-aware interface and iterates over the bindings.

Raise Events. Events are raised in the XML format given in Section A.4.2.1. This can be encoded into an opaque action of sending an HTTP message with the raised event to an appropriate target (the event brokers that support the respective domain).

```
<travel:canceled-flight flight="LH123">
    <travel:reason>bad weather</travel:reason>
</travel:canceled-flight>
<uni:reg_open subject="Databases" />
<uni:register subject="Databases" name="John Doe" />
```

A.4.5.2 Composite Actions

Composite actions can e.g. be described by *process algebras*. Process Algebras describe the semantics of processes in an algebraic way, i.e., by a set of elementary processes (carrier set) and a set of constructors. The semantics can either be given as *denotational semantics*, i.e., by specifying the denotation of every element of the algebra (e.g., CSP – Communicating Sequential Processes, [29]), or as an *operational semantics* by specifying the behavior of every element of the algebra (e.g., CCS – Calculus of Communicating Systems, [47, 48]). Processes defined by Process Algebras can e.g. be used for the specification of *communication*, i.e., for basic protocols, or for defining the behavior of interacting (Semantic) Web Services (note that process algebras provide concepts for defining infinite processes).

The structure and markup of composite action languages becomes relevant when reasoning about a system should be done. For the above algebras, model checking is available for verification.

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

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

is the basic algebra – i.e., containing the least reasonable set of operators – for constructing processes over $\mathcal{A}$. $\bot$ is a constant denoting a deadlock, $+$ denotes alternative composition, and $\cdot$ denotes sequential composition: if x and y are processes, then $x + y$ and $x \cdot y$ are processes. These are essentially the processes that can be characterized in Dynamic Logic [28] and Hennessy-Milner-Logic [49].

A term markup of BPA ist straightforward:

```
<!ENTITY % operand "(alt | seq | atomic-action |
   ANY-from-other-namespace | opaque)">
<!ELEMENT alt (%operand;, %operand;, %operand;*)>
<!ELEMENT seq (%operand;, %operand;, %operand;*)>
<!ELEMENT atomic-action ANY>
```

BPA action specifications occur in rules in the usual form as

```
<eca:rule xmlns:travel="http://www.semwebtech.org/domains/2006/travel">
  <eca:event> ... </eca:event>
  <eca:query>... </eca:query>
  <eca:action xmlns:bpa="...">
    contents in bpa namespace
  </eca:action>
</eca:rule>
```

We do not give a concrete URL (and also no implementation) since composite actions are more comprehensively handled below with CCS.

CCS and CSP. The more sophisticated process algebras *CCS – Calculus of Communicating Systems* [47, 48]), *CSP – Communicating Sequential Processes* [29]), ACP [7], COSY [30], or the *Box Algebra* [9] based on the *Petri Box Calculus*, provide more involved composers, and use several kinds of actions. In addition to the common actions, they include "communication actions", i.e., sending and receiving messages, and also "reading actions" that access the state of a system. In our ontology, these are modeled as events and queries, or tests. Since process algebras allow not only for executing a piece of program code, but also the definition of more complex *processes*, including the definition of independent, communicating processes, the resulting model is also more expressive than current formalisms and languages for active rules.

With this, the action component can be used to specify e.g. the following concepts:

1. a sequence of actions to be executed (as in simple ECA rules),
2. a process that includes "receiving" actions (which are actually events in the standard terminology of ECA rules),
3. guarded (i.e., conditional) execution alternatives,
4. the start of a fixpoint (i.e., iteration or even infinite processes), and
5. a family of *communicating, concurrent processes.*

These patterns can be employed for specifying behavior: (2) can e.g. be used to define a negotiation strategy that communicates with a counterpart. (3) can include different reactions to the answers of the counterpart, (4) extends the behavior even to try again. Note that in these cases, only one side of the communication is specified, whereas the behavior of the counterpart is defined by other

rules (with another owner). (5) can be used to specify even more complex behavior of interacting (Semantic) Web Services as a reaction as known from the agent community.

These tasks can also be expressed by (sets of) simple ECA rules, but this leads to a much less intuitive, and hard-to-understand specification. The composition of the ECA and process algebra concepts (and ontologies) provides a comprehensive framework for describing behavior in the Semantic Web.

Calculus of Communicating Systems (CCS). CCS [47] extends BPA by more expressive operators. A CCS algebra with a carrier set $\mathcal{A}$ is defined as follows, using a set of process variables:

1. Every $a \in \mathcal{A}$ is a process expression.
2. With X a process variable, X is a process expression.
3. With $a \in \mathcal{A}$ and P a process expression, $a : P$ is a process expression (prefixing; sequential composition).
4. With P and Q process expressions, $P \times Q$ is a process expression (parallel composition).
5. With I a set of indices, $P_i : i \in I$ process expressions, $\sum_{i \in I} P_i$ (binary notation: $P_1 + P_2$) is a process expression (alternative composition).
6. With I a set of indices, $X_1, \ldots, X_k$ process variables, and $P_1, \ldots, P_k$ process expressions, $\mathsf{fix}_j \vec{X} \vec{P}$ is a process expression (definition of a communicating system of processes). The fix operator binds the process variables X_i, and fix_j is the jth one of the k processes which are defined by this expression.

Process expressions not containing any free process variables are *processes*.

The (operational) semantics of CCS is given by transition rules that immediately induce a naive implementation strategy (note that the semantics of CSP [29] is given as *denotational semantics*). By carrying out an action, a process changes into another process. Considering the modeling as a Labelled Transition System, a process can be regarded as a state or a configuration, which allows to use Model Checking for verifying properties of CCS specifications.

$$a : P \xrightarrow{a} P \quad , \quad \frac{P_i \xrightarrow{a} P}{\sum_{i \in I} P_i \xrightarrow{a} P} \text{ (for } i \in I)$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q'}{P \times Q \xrightarrow{ab} P' \times Q'} \quad , \quad \frac{P_i\{\mathsf{fix}\ \vec{X} \vec{P} / \vec{X}\} \xrightarrow{a} P'}{\mathsf{fix}_i \vec{X} \vec{P} \xrightarrow{a} P'}$$

Additionally, asynchronous CCS allows for delays:

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

with the corresponding transition rules

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

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

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

The possibility of Delay is especially important when "waiting" for something to occur, e.g., for synchronization.

Actions with Parameters. Actions are usually parameterized, e.g. "book flight no N on *date*". Communication between the rule components is provided by variable bindings. Accordingly, the specification of the action component uses variables as parameters to the actions.

Example 28 (Process Specifications in CCS)

- *A money transfer (from the point of view of the bank) is already a simple process:*

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

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

  $\mathsf{fix}\ X.(\mathsf{first\_of\_month} : \mathsf{debit}(Acc_1, Am) :$
  $\quad \mathsf{deposit}(Acc_2, Am) : \partial\ X)$

- *A more detailed view could e.g. check if the balance will stay positive, and if not, notify the account holder:*

  $\mathsf{fix}\ X.(\mathsf{first\_of\_month} : \mathsf{send\_query}(Acc_1 \geq Am?) :$
  $\quad ((\partial : \mathsf{rec\_msg(yes)} :$
  $\quad\quad \mathsf{debit}(Acc_1, Am) : \mathsf{deposit}(Acc_2, Am)) +$
  $\quad (\partial : \mathsf{rec\_msg(no)} : \mathsf{send\_msg(\$owner,...)})) : \partial\ X)$

  (using messaging for queries and message receipt events for answers).

Another way would be to express the same as a complete ECA rule "if the event first_of_month *occurs, then do ..." instead of a fixpoint process.*

Example 29 *Consider the following scenario: if a student fails twice in an exam, he is not allowed to continue his studies. If the second failure is in a written exam, it is required that another oral assessment takes place for deciding upon final passing or failure.*

This can be formalized as an ECA rule that reacts upon an event failed($Subject,$StudNo) *and then in a further query checks whether this is the second failure of* $StudNo *in* $Subject*, and whether the exam was a written one. The action component of the rule should then specify the process of (organizing) the additional assessment: as an action, the responsible lecturer will be asked for a date and time (send a mail), that will be entered by him into the system (in CCS: a "receiving" communication action; in our approach: an event). The action component is thus as follows:*

ask_appointment($Lecturer, $Subject, $StudNo) :
∂ proposed_appointment($Lecturer, $Subject, $DateTime) :
find_room($DateTime, $Room) :
inform($StudNo, $Subject, $DateTime, $Room) :
inform($Lecturer, $Subject, $DateTime, $Room)

In this example,
proposed_appointment($Lecturer, $Subject, $DateTime) *is an event – for this, it is allowed to be delayed (∂). In contrast, all other items are actions that are actually executed by the process as soon as possible.*

Note that entering the grade and further consequences are not covered by this action. Instead, it is appropriate to have a separate rule that reacts (again) on entering grades and, if the grade was established by such an additional assessment, take appropriate actions.

Conditions. In CCS and other process algebras, there is no explicit notion of states, the properties of a state are given by the (sequences of) actions which can be executed. When representing a stateful process, queries and values are represented e.g., as "read that $A > 0$", or by explicit messages (as the account balance in Example 28). We omit the "read", and allow queries and conditions as regular components of a process:

- "executing" a query means to evaluate the query, extend the variable bindings, and continue.
- "executing" a condition means to evaluate it, and to continue for all tuples of variable bindings where the condition evaluates to "true". For a conditional alternative $((c : a_1)+(\neg c : a_2))$, all variable bindings that satisfy c will be continued in the first branch, and the others are continued with the second branch.

Example 30 (Processes with Conditions)

1. *Consider again the scenario from Example 29, but now only one room is suitable for such assessments. Here, the process in the action part must iterate asking the lecturer for an alternative date/time until the room is available. This is done by combining CCS's fixpoint operator with a conditional alternative:*

   fix$X$.(ask_appointment($Lecturer,$Subj,$StudNo) :
   ∂ proposed_appointment($Lecturer,$Subj,$DateTime) :
   (available(room,$DateTime) +
   ($\neg$ available(room,$DateTime) : X))) :
   inform($StudNo,$Subj,$DateTime) :
   inform($Lecturer,$Subj,$DateTime)

   Here, ask_appointment *is an atomic action,* proposed_appointment *is an event, and* available *is a predicate (test).*

2. *The account check in Example 28 can also be expressed by a conditional alternative:*

   fix X.(first_of_month :
   (($Acc_1 \geq Am$? : debit(Acc_1, Am) : deposit(Acc_2, Am)) +
   ($Acc_1 < Am$? : send_msg($owner,...))) : ∂ X)

Figure A.4.3 shows the relationship between the process algebra language and the contributions of the domain languages and the event and test component languages.

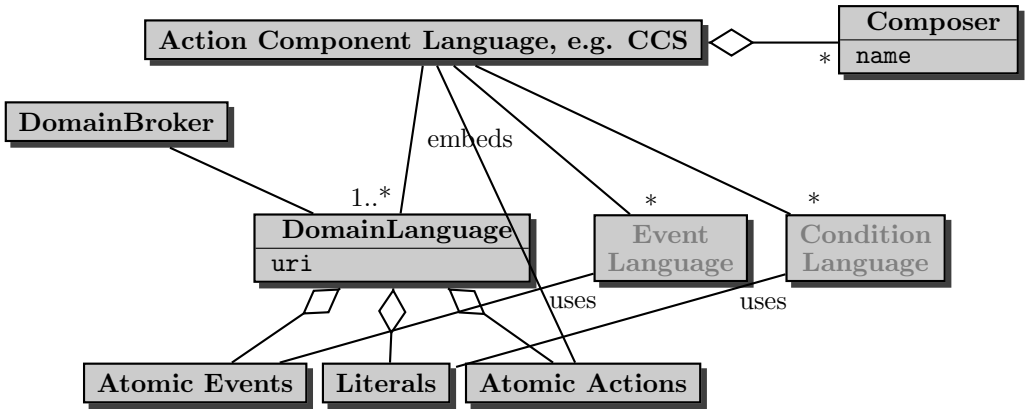


Figure A.4.3: Structure of the Action Component as an Algebraic Language using CCS

XML Markup for the Action Component. According to the above considerations, processes in the Framework are built over

- actions,
- events, and
- conditions

by using the CCS connectives. The language markup has the usual form of a tree structure over the CCS composers in the ccs namespace. The leaves are contributed by (i) atomic actions of the underlying domains, (ii) events and conditions/tests. The latter are not necessarily atomic, but are seen as black-boxes from the CCS point of view, containing markup from appropriate languages as used in the ECA event and test components (and handled by the respective services). A longer example will be given in Section A.4.5.5.

A.4.5.3 Atomic and Leaf Items

As discussed above, the leaves on the CCS level can be atomic actions, or embedded events, queries, or test subexpressions. In accordance with ECA-ML, the latter are embedded into ccs:event, ccs:query and ccs:test elements. The language identification is done again via the namespaces.

Atomic Actions. Atomic actions belong to some domain namespace, thus the element is in general the action in XML markup "itself" (including variables as {$varname}):

```
<domain-ns:action-name attributes>
  contents
</domain-ns:action-name>
```

As an example consider an atomic action that books a given flight (flight code bound to variable $flight) at a given date (bound to variable $date):

```
<travel:book-flight code="{$flight}" date="{$date}" />
```

Embedded Events. Embedded events are also leaves, contained in ccs:event elements (with the same semantics as eca:event elements on the ECA level):

```
<ccs:event xmlns:el="ev-uri">
  event expression in appropriate markup
</ccs:event>
```

Arbitrary event languages and formalisms that are supported by some service are allowed. Note that composite events integrate smoothly since they are considered to occur with the final detection of the composite event.

Embedded Conditions. Embedded tests are handled exactly in the same way:

```
<ccs:test xmlns:cl="cl-uri">
  test expression in appropriate markup
</ccs:test>
```

Embedded Opaque Items. Opaque actions (i.e., program code, mainly for queries, tests and also for actions) can be embedded as leaf elements:

<ccs:opaque {url="*node-url*"|language="*name*"}>
program code fragment
</ccs:opaque>

For such fragments, either a URL where the action has to be sent to (as HTTP GET) is given, or the language is indicated (then the fragment must contain the addressing of the target node itself).

A.4.5.4 Example: CCS

Following a straightforward principle for term markup, the CCS operators are represented by XML elements (with parameters as attributes) according to the following nearly-DTD specification:

```
<!ENTITY % operand "(delay | sequence |
   alternative | concurrent | fixpoint |
   atomic-action | event | query | test |
   opaque)">
<!ELEMENT delay EMPTY>
<!ELEMENT sequence
  (%operand;, %operand;+)>
  <!ATTLIST sequence mode "async">
<!ELEMENT alternative
  (%operand;, %operand;+)>
<!ELEMENT concurrent
  (%operand;, %operand;+)>
<!ELEMENT fixpoint (%operand;+)>
  <!ATTLIST fixpoint
     variables #REQUIRED NMTOKENS
     index #REQUIRED NMTOKEN
     localvars #IMPLIED NMTOKENS>
<!ELEMENT atomic-action ANY>
<!ELEMENT event ANY>
<!ELEMENT query ANY>
<!ELEMENT test ANY>
<!ELEMENT action ANY>
  <!ATTLIST event,test,action
          xmlns:%name; #REQUIRED %URI;>
<!ELEMENT opaque ANY>
  <!ATTLIST opaque
     language #IMPLIED CDATA
     url #IMPLIED CDATA>
  <!ATTLIST all-elements group-by "">
```

The semantics of the elements is described below.

- The content of all the "simple" operators consists of at least two subelements.
- <ccs:delay/> indicates a delay (for waiting, in case that synchronous context is used),
- <ccs:seq mode="*mode*">*contents*</ccs:seq> indicates a sequence. *mode* can be sync or async corresponding to synchronous CCS (with ":" as standard combinator) or asynchronous CCS (with "." as standard combinator); default is mode="async".

- <ccs:alt>*contents*</ccs:alt> stands for "$\sum$" and "+" (alternatives),
- <ccs:concurrent mode="*mode*">*contents*</ccs:concurrent> represents "×" and "|" (parallel),
- The <ccs:event xmlns:*lang*="*uri*">, <ccs:query xmlns:*lang*="*uri*">, <ccs:test xmlns:*lang*="*uri*">, and <ccs:action xmlns:*lang*="*uri*"> elements allow for embedded events, queries, tests, or actions (the latter even allow for embedding an action/process specification in another language).

Handling of "new" Variables in Fixpoint Processes. For integration with the ECA Framework that uses logical variables (that can be bound only once), variables that are bound during the evaluation of the fixpoint part must be considered to be local to the current iteration, and only the final result is then bound to the actual logical variable:

- <ccs:fixpoint variables="var_1 ... var_n" index="j"
  localvars="*list of variables*">
  contents
  </ccs:fixpoint>

provides the markup for fixpoint constructs. The var_i are the process variables, j is the index of the one of the processes that is chosen, and the variables distinguished to be local can be bound in each iteration; after reaching the fixpoint they keep the value of the last iteration.

Example 31 (Variables in Fixpoint Processes) *Consider again Example 30(1). There, each iteration of the fixpoint process searching for a date where the room is available binds $DateTime. The actual semantics is easy to understand and implement: just keep the last value.*

Grouping. For each subexpression, it can be specified if it is executed for the whole set, or separately for each tuple, or some grouping (in the same way as grouping in SQL) is applied. Clearly, subactions can only have finer granularity than the outer expressions). For specifying grouping, each action element has an optional attribute

group-by="*variable list*"

that indicates grouping. E.g., given variables X, Y, Z, group-by="X Y" means to execute the subexpression separately for all sets that have X and Y in common. Default is group-by="" which means to have one group with all tuples. For convenience, group-by="-separately" means to process every tuple separately, and group-by="-bulk" also means to have one group with all tuples.

A.4.5.5 Example in CCS

Consider a rule that does the following: if a flight is first delayed and then canceled (note: use of a join variable), make a reservation for each passenger at the airport hotel, and send each business class passenger an SMS.

```
<eca:rule xmlns:eca="http://www.semwebtech.org/eca/2006/eca-ml">
  <eca:event xmlns:snoopy="http://www.semwebtech.org/eca/2006/snoopy">
    <snoopy:sequence>
      <travel:delayed-flight flight="{$flight}" date="{$date}"/>
      <travel:canceled-flight flight="{$flight}" date="{$date}"/>
    </snoopy:sequence>
  </eca:event>
  <eca:query>
    <eca:opaque language="sparql" domain="travel">
      SELECT $booking, $name WHERE
      ($x  rdf:type  travel:flight)
```

```
      ($x travel:code $flight) ($x travel:date $date)
      ($x travel:booking $booking)
      ($booking travel:name $name)
    </eca:opaque>
  </eca:query>
  <eca:action xmlns:ccs="http://www.semwebtech.org/eca/2006/ccs">
    <ccs:concurrent>
      <ccs:atomic-action>
        <travel:reserve-room hotel="hotel uri" name="$name"/>
      </ccs:atomic-action>
      <ccs:sequence>
        <ccs:test >
          <eca:opaque language="sparql" domain="travel">
            ($booking travel:class travel:business-class)
          </eca:opaque>
        </ccs:test>
        <ccs:query>
          <eca:opaque language="sparql" domain="travel">
            SELECT $phone WHERE
            ($booking travel:contact-phone $phone)
          </eca:opaque>
        </ccs:query>
        <ccs:atomic-action>
          <comm:send-sms to= "$phone">
            "we are very sorry ... and booked a room in ... for you"
          </comm:send-sms>
        </ccs:atomic-action>
      </ccs:sequence>
    </ccs:concurrent>
  </eca:action>
</eca:rule>
```

- after the event part, $flight is bound to the flight number,
- after the query, for each booking there is a tuple of bindings $flight, $date, $booking, $name.
- The action component does two things in parallel: reserve rooms, and for each tuple (i.e., for each booking) check if it is a business class booking (test), if yes, get the contact phone number (query) and send an SMS.

A.4.5.6 Processing

Thus, the implementation of the CCS engine is only concerned with the actual CCS operators (i.e., the elements in the ccs: namespace. Atomic actions are executed "immediately" by submitting them to the domain nodes (if specified by an URL) or to a domain broker (see Section A.5.6) that is responsible for the domain, forwarding them to appropriate domain nodes).

The handling of embedded <ccs:event>, <ccs:query>, <ccs:test>, and <ccs:action> elements with embedded fragments of other languages is done in the same way as the evaluation of components by the ECA engine. This will be discussed in Section A.6.5.

A.4.5.7 ECA Rules vs. CCS

In general, it is possible to decompose a CCS description completely into ECA rules with atomic actions (or with restricting the action component to BPA process specifications), or to express ECA rules as a special form of CCS processes over the above-mentioned atomic items. The advantage

of supporting both formalisms in the framework lies in the appropriateness of modeling: there is behavior that is preferably and inherently formalized as ECA rules, and there is behavior that is preferably formalized in CCS. Providing both formalisms thus eases the modeling, and with this also the understandability and maintenance of behavior:

> "If something happens (event) in a certain situation (condition), then proceed according to a given policy (action, as CCS process)."

Having ECA rules and processes allows to model both *reactive* and *continuous* behavior in an appropriate way.

A.4.6 Summary

So far, the component languages as "intermediate" level between the surrounding ECA language and the domain languages have been discussed. The Web architecture that provides the global cooperation and communication, i.e., given a rule, finding appropriate services for the components will be discussed in Section A.6. The next section first deals with the domain level. The domain level is made up from autonomous domain nodes, e.g., airlines, train companies, and car rentals, and *domain brokers* that provide integrating functionality for domains.

Chapter (Appendix A: ECA Framework) A.5

Domains and Domain Nodes: Architecture and Communication

Each Semantic Web application uses one or more domains. Often, an application (and its rules) has a core domain, and also touches several other domains.

Example 32 *Consider a travel agency. It "lives" in the travel domain, i.e., its behavior and user interface are dealing with this domain, and for serving this central purpose it communicates intensively with many other nodes in this domain. Additionally, it touches e.g. the banking domain, and sometimes also the medical domain (if travelers request information about mandatory or recommended vaccinations) etc.*

Domain Nodes (DNs) have a basic functionality wrt. the concepts of a domain ontology. For a domain, there are usually many nodes that provide data and/or services in this domain. For most domains, even the set of participating nodes is dynamic. Nodes come and go.
In the following sections, we describe:

- generic ontology issues of domains,
- generic interfaces and functionality of domain nodes; architecture samples are given in Chapter A.7.
- communication and *brokering* issues.

A.5.1 Domain Ontologies

As discussed in Section A.2.1.3, a domain ontology is partitioned into (static) literal notions, namedactions, and named events, as shown in Figure A.5.1. Note that events and actions can also be structured into a class hierarchy (which is orthogonal to a "defined"-relationship between composite events/actions and simpler ones). All notions that describe and structure domains are defined in the "metadomain" associated with the `world` namespace, associated with the URL

```
http://www.semwebtech.org/domains/2006/world
```

```
<world:Domain rdf:type, owl:Class>
<world:Event rdf:type, owl:Class>
<world:Action rdf:type, owl:Class>
```

In the Framework, domain namespaces (e.g., travel) are usually associated with an URL where an RDF document can be found that provides information about the domain. This consists of the RDF/RDFS/OWL definition of the domain ontology itself.

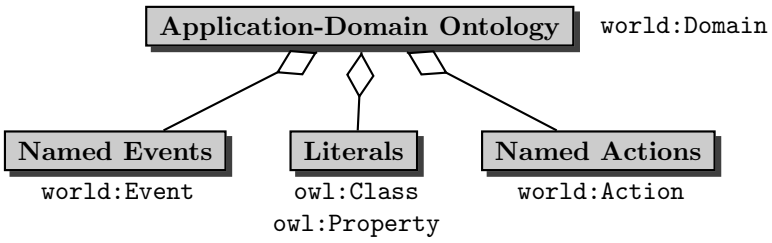


Figure A.5.1: Components of Domain Ontologies

Example 33 (Definition of the Travel Ontology) *The travel domain ontology provides –among others– the following classes, properties, events and actions:*

```
<travel, rdf:type, world:Domain>
<travel, rdf:type, world:Domain>
<travel:airline, rdfs:subClassOf, world:Domain-Service>
<travel:flight-event rdfs:subClassOf world:Event>
<travel:car-rental, rdfs:subClassOf, world:Domain-Service>
<travel:cancellation, rdf:type, world:Event>
<travel:canceled-flight, rdf:type, world:Event>
<travel:canceled-flight, rdfs:subClassOf, travel:cancellation>
<travel:canceled-flight, rdfs:subClassOf, world:Event>
<travel:cancel-flight, rdf:type, world:Action>
<travel:delay-flight, rdf:type, world:Action>
<travel:fully-booked, rdf:type, world:Event>
<travel:fully-booked, rdf:type, world:Event>
```

With such an (event) class hierarchy, there could be a rule that informs about any cancellations of any means of transportation. Note that it is not possible to express that has-flight *is a (mandatory) property of flight-events, flights, flight-actions etc.*

Application services (e.g., airlines) use (one or more) domain ontologies, and they are supported by *application services* that are available at some URL (note that this connects with the concepts in Section A.6.1.2). An application service can support multiple domains, e.g., an airline service will support travel and business (for running its business, paying taxes), etc. For supporting the work of domain brokers (see Section A.5.6), the information about an application service specifies which notions of a domain are actually supported by the service (e.g., an airline supports travel:airport, but not travel:cancel-train).
Such *dynamic* information about a domain is also supported by the `world` ontology:

- which actual Semantic Web Nodes support a domain and its concepts, and
- which domain brokers support the domain:

```
<world:Service, rdf:type, world:Service>

<world:Domain-Service, rdfs:subClassOf, world:Service>
<world:uses-domain, rdf:type, rdf:Property>
<world:uses-domain, rdfs:domain, world:Domain-Service>
<world:uses-domain, rdfs:range, world:Domain>
<world:has-service, owl:inverseOf, world:uses-domain>
<world:supports, rdf:type, rdf:Property>
<world:supports, rdfs:domain, world:Domain-Service>
<world:supports, rdfs:range, owl:Class>
```

```
<world:Domain-Broker, rdfs:subClassOf, world:Service>
<world:has-domain-broker, rdf:type, rdf:Property>
<world:has-domain-broker, rdfs:domain, world:Domain>
<world:has-domain-broker, rdfs:range, world:Domain-Broker>
```

This information is maintained by one or more *Domain Service Registries (DSRs)* that provide metadata about the services. Note that the DSR is not a static thing, but serves as a registry (where e.g. new services can be added). [although the prototype will probably first come with an XML file].

Note that this does not yet relate application services with domain brokers – this has *additionally* to be done (this allows for negotiation and selection, e.g. based on trust and recommendation).

A.5.2 Description of Application Services

Example 34 (Metadata on Application Services) *Consider application services* Orafly *and* Exist-Cars *in the traveling area (running the respective business and being implemented on some infrastructure):*

```
<orafly, rdf:type, world:company>
<orafly, business:has-business, world:airline>
<orafly, world:uses_domain, http://www.semwebtech.org/domains/2006/travel>
<!-- the following is the actual service in the Semantic Web -->
<orafly, world:has-service, http://.../oracle-airline>
<!-- the following is the address of the Homepage in the HTML Web-->
<orafly, world:has-url, http://.../oracle-airline/index.html>
<orafly, world:uses-domain, travel>
<orafly, world:uses-domain, business>
<orafly, world:supports, travel:airport>
<orafly, world:supports, travel:flight-connection>
<orafly, world:supports, travel:is-delayed>
<orafly, world:supports, business:has-taxnumber>
<orafly, business:has-taxnumber, "eur-0815">
<exist-cars, rdf:type, world:company>
<exist-cars, business:has-business, world:car-rental>
<exist-cars, world:uses-domain, http://www.semwebtech.org/domains/2006/travel>
<exist-cars, world:has-service, http://.../exist-cars>
<exist-cars, world:has-url, http://.../exist-cars/index.html>
```

The behavior and interfaces of application services are described next.

A.5.3 Basic Functionality of Domain Nodes/Domain Node Interfaces

The basic functionality of a domain node can be more or less primitive.

A.5.3.1 Providing Static Data

Files. The most simple domain "node" is just an data source that can be read as a whole. Such standalone XML or RDF files on the Web can e.g. be accessed via a generic XQuery or SPARQL wrapper service:

- XQuery: let $doc := document("*url*") ...
- SPARQL: select ... from *url* where ...

Opaque Queries. While the above files are the simplest kind of "Web resources", many data sources in the current Web provide a simple querying interface via HTTP GET or POST (e.g., an SQL, XPath, XQuery, RDQL, or SPARQL query interface). Wrt. the framework, such queries are handled as opaque queries whose answer are e.g. sets of XML nodes. The ECA engine deals with this via a generic wrapper for HTTP or XQuery etc. (cf. Section A.7.1.2).

Framework-Aware Quering. In full exploitation of the Framework, also the query component will be stated in structured way by an algebraic query language in XML or RDF markup. Only literal queries will occur at the domain nodes (in RDF/OWL: for extension of classes and relationships). The communication (especially the formatting of the results) is then done on the communication format given in Section A.3.1.4.

A.5.3.2 Providing Behavior

Behavior for Web nodes means usually to be able to process requests, i.e., actions. These can be given as opaque code in the local programming language (e.g., PL/SQL or XUpdate) or method calls (using SOAP markup; methods must be implemented locally as procedures or by ACA rules).

Full-fledged Semantic Web Domain nodes support actions of the corresponding domain ontologies (i.e., things ?X such that (?X, rdf:type, world:Action) holds) that are communicated according to the downward communication format given in Section A.3.1.4.

For legacy nodes that support only simple database updates (SQL, XUpdate), wrappers can be used for adapting them to the ontology level.

Filtering Relevance of Requested Actions. As will be described below, due to the domain brokering, there may be update requests that are actually not relevant for a node. Thus, such updates are first checked by an outer layer (implemented similar to INSTEAD-triggers that decide if (and optionally how) they are executed.

A.5.3.3 Reporting Behavior: Providing Atomic Events

The current Web architecture does not consider "events" communicated by *push* communication. For the Framework, nodes are expected to emit atomic low-level events (i.e., things ?X such that (?X, rdf:type, world:Event) holds). They are communicated in XML format as described in Section A.4.2.1, or later in RDF format.

Note that there are many legacy nodes that do not support events. They have to be monitored to detect their events. Communication of "events" or "news" is currently partially supported by *RSS Feeds* that provide events for *pull* communication (see Section A.5.6.3).

In the Framework, events are forwarded to event brokers (as separate services) where filtering and communication to the outside takes place.

A.5.4 Rules in Ontologies

Full-fledged nodes should also support local reactivity, as e.g. provided by SQL Triggers, reactive rules. There are also several kinds of rules in ontologies:

1. logical derivation rules for deriving concept membership or instances of properties,
2. ECE rules: derive composite events from simpler ones (e.g., "flight 50% booked" from "if a booking occurs such that 50% are reached"),
3. ACA rules: map higher-level actions to simpler ones "book a return ticket to X on dates A and B" to "book a ticket on date A" and "book a ticket back on date B". (note that this defines the meaning of "return ticket" without having an explicit concept of a "return ticket" in the ontology.)

Note that some instances of such rules belong to the ontology, while others possibly only belong to certain services. Here, the rules that belong to the ontology are of interest.

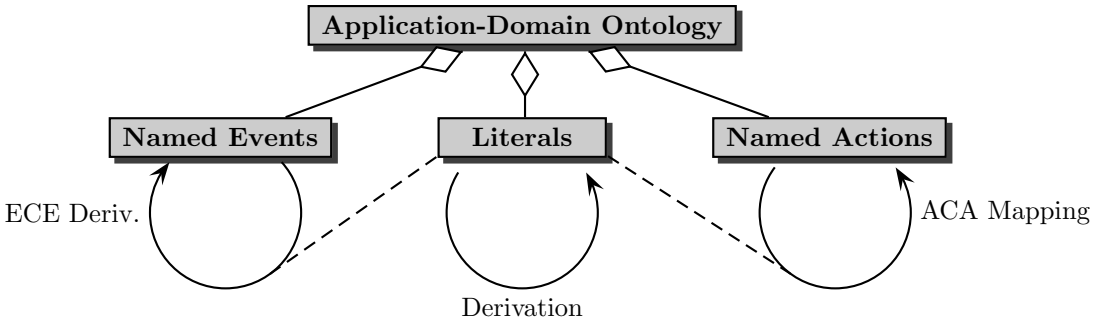


Figure A.5.2: Derivation and Mapping Rules for Events, Literals and Actions

ECE rules define derived, usually composite, events. ACA rules define composite actions. Both have a close similarity to ECA rules: both make use of component languages, and their implementation is preferably based on ECA rules. In both cases, the condition component often serves not only as a condition, but as a query part that extends the variable bindings (both by new variables, and also by duplicating tuples during this extension).

A.5.4.1 Derivation Rules

Derivation rules can be formulated as OWL axioms (subclasses, definitory axioms of classes as intersections/unions/restrictions of others), or by full (first-order) derivation rules. For the latter, there are several proposals:

- the RuleML initiative that works upon markup for derivation rules
- derivation rules in Jena.

The actual evaluation of such rules can be located in the domain broker and optionally also in the application node itself. Note that individual nodes can additionally have own local rules.

A.5.4.2 ECE Rules

ECE rules define derived, usually composite, events.

Example 35 (Derived Event) *Consider an airline that raises the price for flights after 50% of a plane are booked. This can be done by a rule reacting on a derived event: "when 50% of the seats of flight number $f on date $date are booked then ...". The derived event "50% of the seats of flight number $f on date $date are booked" has to be defined in the ontology, and it is raised by another ECE rule "when a flight is booked and this is just the booking that exceeds the 50% of the seats quota, then raise the event" half-booked.*

problem in test!!!

```
<world:definition syntax= "xml">
  <world:defined>
    <!-- pattern of the event to be derived -->
    <travel:half-booked flight="$f" date="$d"/>
  </world:defined>
  <world:defined-as>
    <!-- E/C components how to derive it -->
    <eca:event>
      <travel:booking flight="$f" date="$d"/>
    </eca:event>
```

```
    <eca:test>
      <eca:opaque language="xpath">
        <!-- problem: against which database? – needs RDF! -->
        count($flight[@date="$d"]/booking) = $flight/id(@aircraft)/@number-of-seats div 2
      </eca:opaque>
    </eca:test>
  </world:defined-as>
</world:definition>
```

Note that event types *event-type* that are defined as derived events also have to be "declared" in the OWL part of the ontology as <*event-type* rdf:type world:Event>.

The implementation of derived events in course of event brokering will be discussed in Section A.5.6.2.

A.5.4.3 ACA Rules

ACA rules define complex actions in terms of simpler ones. ACA rules exist on two levels: (i) abstract specification of composite named actions as processes over simpler named, still abstract actions, and (ii), local implementation of named actions by operations on a data model.

Composite abstract named actions of the ontology can be specified as processes consisting of less abstract named actions. In this case, the specification is actually of the same structure as the action components of ECA rules (and the query component is also the same as for ECA rules). Such rules correspond to *definitions* of higher-level actions in the ontology. They usually hold ontology-wide.

```
<world:definition syntax= "xml">
  <world:defined>
    <banking:money-transfer amount="$amount" from="$from" from="$to"/>
  </world:defined>
  <world:defined-as>
    <eca:action>
      <ccs:concurrent xmlns:ccs="...">
        <banking:debit account="$from"/>
        <banking:deposit account="$to"/>
      </ccs:concurrent>
    </eca:action>
  </world:defined-as>
</world:definition>
```

Note that action types that are defined as composite actions also have to be "declared" in the OWL part of the ontology as <*action-type* rdf:type world:Action>.

The implementation of composite actions in course of action brokering and ACA rules will be discussed in Sections A.5.6.6 and A.5.7.

A.5.4.4 Discussion: Comparison with RuleML Proposal

RuleML proposes a rule markup as ruleml:imp, ruleml:head, ruleml:body for representing rules. We did adhere to this for the following reasons;

- imp, head, body are clear for derivation rules and for event definition rules (since these are also a kind views), but *not* for ACA rules.
- imp, head, body refer to syntactic notions of the representation of rules as formal languages grammars, *not* to their semantics as derivation or reduction rules.

Making rules *full citizens* of an ontology (like owl:class and owl:property) allows for a more appropriate modeling wrt. the different kinds of rules as

- derivation rules (logical rules, ECE rules),
- reduction rules (ACA rules),
- integrity rules as assertions.

A.5.5 Domain Node Local Behavior

Some domain nodes have nontrivial local behavior. This can be black-box behavior, implemented in any programming languages, or given by ACA, ECA and ECE rules (often by database-level triggers). The latter are e.g. used for defining derived events.

The information flow between events and actions is depicted in Figure A.5.3 and contains the following types of rules:

1. low-level ECA rules as triggers for local integrity maintenance in a knowledge base,
2. ECE rules: derive and raise application-level events based on internal changes knowledge base level,
3. global ECA rules that use application-level events,
4. ACA mapping: map high-level actions to lower-level (e.g. INSTEAD OF triggers).

Amongst these, (1), (2), and partially also (4) are based on *database level and knowledge base level triggers*.

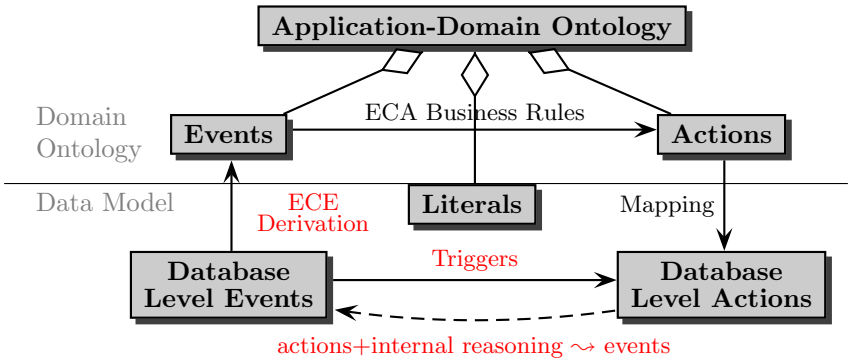


Figure A.5.3: Interference of Events, Actions, and Literals

An RDF/OWL domain node with local active behavior based on Jena [31] is described in [46].

Local Implementation of Abstract Actions by Data Model Operations

Named actions are mapped onto actual update operations on the data model level of individual domain nodes:

```
IMPLEMENT
<travel:delay-flight flight="{$flight} time="{$min}" />
BY
UPDATE arrivals
SET time:=time+$min WHERE flight=$flight
```

Such rules are usually local to an application node (since they heavily depend on its internal data model and schema).

A.5.6 Domain Brokering

A.5.6.1 Event Brokering

Event Brokering functionality provides the mediation between event providers (i.e., domain nodes) and event consumers. The latter are the AEMs as the mediators towards the ECA engines: they take events and produce answers.

The communication of events on this level is depicted in Figure A.5.4. The main communication between DNs and EBs is very simple: A DN forwards all events (as XML fragments according to the domain ontology) to its peer EB(s).

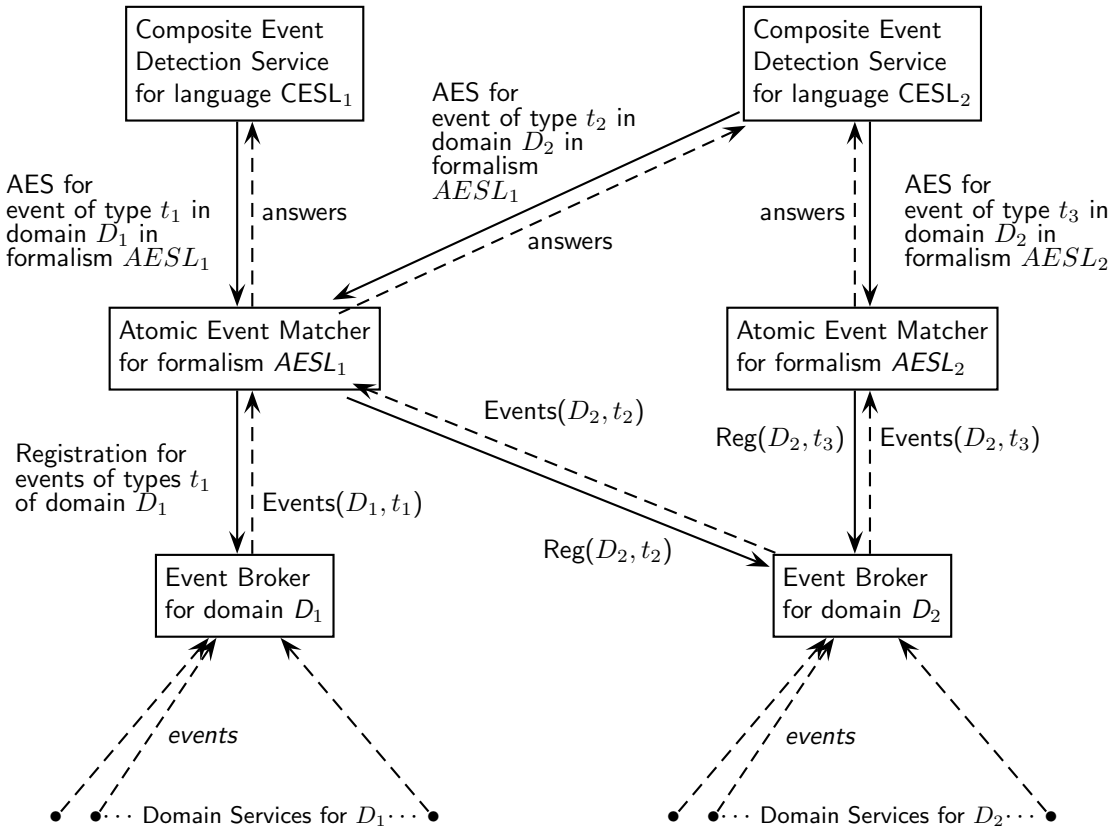


Figure A.5.4: Architecture: Communication with Event Brokers

Communication of Events between AEM and EB. As described in Section A.4.3, the CEDs (and in case of an atomic event component also the ECA engine) register AESs at the AEMs to be detected. The AEMs have to be aware of all relevant events. For this, they determine the domain (URI) of the event to be detected. With this URI, they can find out at the *Languages and Services Registry (LSR)* (see Section A.6.1.2) what EBs are serving this domain. The AEM then tells an EB to forward them the required events. The specification which events should be forwarded contains the domain name/URL (note that an EB may support more than one domain) and optionally also the type of the event.

The registration message follows the patterns discussed in Section A.4.1, e.g.,

to: service-URL of the EB
<register>

```
  <Reply-To>URL where the events are expected at the AEM </Reply-To>
  <domain>domain-uri</domain>
  <event-type type>event type</event-type>
</register>
```

Example 36 (Registration at the Event Broker) *Assume an AEM for the sample XML-QL-style event matching formalism gets the following AES to be detected:*

```
  <travel:canceled-flight xmlns:travel="http://www.semwebtech.org/domains/2006/travel"
      number="{$flight}" />
```

It then explores which EB serves the travel domain (for details, see Chapter A.6). Using the fact that the root element of the event is travel:canceled-flight, it tells the EB to forward all travel:canceled-flight to it.

Note that when using more involved ontologies and RDF/OWL-based AES formalisms, event classes and subclasses can be defined. In such cases, simple matching of element name is not appropriate, but OWL reasoning has to be applied.

From this moment on, the EB forwards all events of this type to the AEM. In case an AES is deregistered, the AEM can also deregister (note that for this, it has to do bookkeeping if other AESs still use the respective elements).

Optional and Additional Functionality of Event Brokers. Note that the *Event Brokers (EBs)* can optionally provide AEM functionality for some formalisms. Since every service registers its functionality in the LSRs, this will be exploited correctly when the CED is searching for an AEM to match certain AESs.

There can be domain nodes that do not provide events. In this case, event brokers can also apply *continuous-query-event (CQE) rules* for detecting events by monitoring the respective sources.

A.5.6.2 Brokering of Derived Events

Derived events are specified as ECE rules whose defined-as subelement consists of a (composite) event specification and an optional query and/or test. From these, an ECA rule that explicitly raises the derived event can be generated in a straightforward way:

```
<eca:rule>
  <!-- contents of the body of the ECE rule definition -->
  <eca:action>
    <eca:raise-event>
      <!-- head of the ECE rule -->
    </eca:raise-event>
  </eca:action>
</eca:rule>
```

Note that there can be event types that are both directly supported by some domain nodes (in most cases, these have their own rules to derive them), and for which the ontology provides an ECE derivation rules.

A.5.6.3 RSS-based Event Brokering

Event Brokering for a given domain may include to raise events obtained from RSS feeds (e.g., from bioinformatics sources). This can already be done in an ECA-way, using an ontology of RSS feeds:
The broker polls the feed regularly. It is then processes by removing items that have already processed. Then, for each new item an event

<rss:new-feed-item url="the feed-url">one item</rss:new-feed-item>

is raised. From that, the contents (mainly the description element) is used in an *application-specific* way to raise application-dependent events. For this, it is necessary that the event part of the application ontology covers the events that are reported by the RSS feeds.

Remark *This will be applied in a Bioinformatics case study.*

A.5.6.4 Query Brokering

Clients can send queries as request either to certain domain nodes (especially opaque queries), or to the domain broker. The domain broker knows the respective ontology that consists of

- OWL statements and
- rules.

The domain broker is responsible as a mediator to answer the query by using application services. For this, a lot of algorithms for mediating and integrating information can be applied. To get the infrastructure running, a simple approach is followed first.

Prototype: Simple Approach

Example 37 *Consider a query for "connections from Göttingen to St. Malo". The ontology specifies that "connection" is the transitive closure of train, flight, and ship connections.*

Thus, the broker can ask for all "connection" instances in any domain node, and then try to combine them.

Decomposition of the Query. Decomposition of a query means to collect all static notions (concepts and properties) that are relevant for answering the query. Note that for forwarding queries, declarations of

- owl:inverseOf and
- owl:equivalentProperty

must be considered. Moreover, if the ontology contains rules of the form $head \leftarrow body$, and the notion in the head is asked, notions occurring in the bodies have to be answered. For this, the rules have to be given in an appropriate markup (e.g., RuleML).
The relevant facts about a rule can be represented in RDF as follows (derivable from RuleML markup):

[Draft]
<*rule*, defines, *notion*>
<*rule*, uses, *notion*>

Selecting nodes to be queried. When the decomposition is computed, each of the notions is forwarded to relevant nodes, i.e., all services known to the domain broker that world:support the notion.

Combining the answers and answering the original query. The domain broker collects the answers and by this obtains all instances of relevant class memberships and properties (as RDF triples). It takes the union of this (as a local RDF knowledge base) and answers the original query.

A.5.6.5 Action Brokering

Clients can request actions either at certain domain nodes (especially opaque actions as explicit update statements), or at the domain broker. The broker has to forward the task to one or more domain nodes.

Example 38 *Consider the case that the domain node representing* Frankfurt Airport *decides that a given flight has to be delayed by one hour due to bad weather conditions, e.g., by a rule*

```
<eca:rule xmlns:travel="http://www.semwebtech.org/domains/2006/travel">
  <eca:event>bad snow conditions detected </eca:event>
  <eca:query>all $flights departing in the next hour </eca:query>
  <eca:action>
    <travel:delay-flight code="{$flight}" delay="1h">
      <travel:reason>bad weather conditions</travel:reason>
    </travel:delay-flight>
  </eca:action>
</eca:rule>
```

Note that the travel *ontology contains a triple*

```
(travel:delay-flight, rdf:type, world:action)
```

that indicates that this is indeed an action. Consider the action instance

```
<travel:delay-flight code="LH123" delay="1h">
  <travel:reason>bad weather conditions</travel:reason>
</travel:delay-flight>
```

or, using an RDF URI [means, we have to go somehow from XML events to RDF events]

```
delay-flight(iata://flights/lh123, "1h", "bad weather conditions")
```

Then, it is intuitively clear that the booking action is actually only executed at the corresponding airline.

The action can be mapped onto RDF-level updates either by the domain broker, or it is sent as an action to relevant nodes and mapped there.

Mapping of the Action by the Domain Broker. The domain ontology contains an ACA rule that specifies how the action is mapped onto RDF-level updates, e.g.

```
IMPLEMENT
travel:delay-flight($flight-code, $time, $reason)
WHERE $flight-uri := uri($flight-code)
BY
ASSERT ($flight-uri, travel:flight-is-delayed, $time)
ANNOTATE WITH (travel:reason, $reason)
```

The broker must then check which nodes in the travel domain could be interested in this (i.e., nodes that world:support the predicate travel:flight-is-delayed are the airlines, but e.g. not the train companies and hotels).

Thus, in a first approach, the update is sent to all nodes that support the corresponding predicate. The nodes must then decide based on the actual data, if they are actually concerned.

Note that the reader knows that the update does only concern the *Lufthansa* airline that actually operates LH123, but this requires not only to use metadata, but also data. For implementing this, the ontology must for each action (type) specify, which are the relevant nodes

(?A,has-relevant-node, ?Airline) :-
(?A, rdf:type, travel:delay-flight),
(?A, talks-about, ?Flight),
(?Flight, travel:operated-by, ?Airline).

If such a specification exists in the ontology, it can be used. Otherwise the "broadcast" as above has to be done. Note that (?Flight, travel:operated-by, ?Airline) cannot be answered from the action only, thus either *all* nodes have to be asked if they operate this flight, or such "key" information must also be present in the domain broker (thus, it is quite useful that the broker also maintains a knowledge base with unchanging domain-dependent knowledge.

Example 39 (Cancellation of multiple flights) *Consider again the action component from Example 27. Since the action is travel:cancel-flight, the action is submitted to a domain broker that supports the travel namespace. It then determines the nodes that support the action travel:cancel-flight. Then (either as one request with multiple variable bindings, or as one request for each binding), the task is submitted to* each *o these nodes (i.e., each airline). The airlines themselves then decide what they have to do (e.g., when the action+binding is to cancel flight LH123, only the Lufthansa node will actually execute it, whereas the AirFrance node will just do nothing).*

Mapping of the Action by the Domain Nodes. When the domain broker does not resolve the action, it is forwarded to the individual domain nodes. Again, the metadata about the nodes contains information which world:Actions they world:support, so the action is again forwarded to all nodes that support travel:delay-flight:

(http://lufthansa.com, world:supports, travel:delay-flight)
(http://airfrance.com, world:supports, travel:delay-flight) etc.

The individual nodes then resolve the actions by local ACA triggers (where also data conditions are checked), e.g.

IMPLEMENT
travel:delay-flight($flight-code, $time, $reason)
WHERE *flight is operated by us*
AND $flight-uri := uri($flight-code)
BY
ASSERT ($flight-uri, travel:flight-is-delayed, $time)
ANNOTATE WITH (travel:reason, $reason)
// *and ignore delays of all other flights*

Actions vs. Event-Driven Architecture In an event-driven architecture, it is recommended to replace the action by raising an event "this must be done" on which the corresponding domain nodes react.

Example 40 (Actions vs. Events) *Consider the following case: at some airport the weather conditions are forecasted to become bad, so that incoming flights cannot land. There is then a rule "if the forecast is ... then for all flights landing in the afternoon, cancel these flights". All flights that are concerned can easily be selected from the flight schedule. Since these are operated by different airlines, the action "cancel-flight($flightno)" must be directed to several targets (airlines).*

Here it is better to extend the domain ontology by an event must-be-canceled($flightno) on which the respective airlines can react.

A.5.6.6 Brokering of Derived Actions

In case that all actions of an application node come through a domain broker, it is not necessary that it implements the ACA rules by itself (it should then also not list the action as supported in its service description)

Domain brokers should support these rules. Since the rules have a close similarity to ECA rules, they are handled by an architectural extension of ECA engines and appropriate communication; see Section A.5.7.

A.5.7 Handling of Composite Actions by ACA Rules

As discussed in Sections A.5.4.3 and A.5.6.6, ACA rules are a suitable paradigm for expressing actions on a higher abstraction level that are defined as composite actions (e.g., defining a money transfer as a debit followed by a deposit). ACA rules usually use only a single domain, but can also be extended to multiple domains.

The structure of ACA rules is closely related to ECA rules: the information flow between ACE and ECE by variable bindings is the same, and the C and E components are the same as in ECA rules (although, the C component is often empty).

Even the "ON .. DO" structure is the same: on *invocation of an action* and on *occurrence of an event*. ACA rules are triggered only upon atomic invocations (requesting some action (name) with parameters). Thus, a comparison with ECA rules with atomic events is appropriate. The specification of the invoking action is expected to use the same mechanisms as for atomic event specification. Thus, the AEMs can also be employed here.

For handling ACA rules, the ECA engine architecture can be extended as to handle also ACA rules. When a domain broker is initialized with an ontology, it registers all ACA rules of the ontology at an ACA-aware ECA engine. Actions that are defined via ACA rules are then processed similar to atomic event patterns and atomic events.

ACA-Aware ECA-Engines. The ACA rules use a dialect of ECA-ML:

- eca:aca-rule for ACA rules,
  <!ELEMENT aca-rule (%variable-decl,define-action,query*,test?,action+)>
- eca:define-action for the action that is defined by the rule.

(Note that ACA rules with a markup as ECA rules will also be processed correctly since the operational semantics is the same).

Actions as Events. The domain broker registers ACA rules at an ACA-aware ECA engine. The ECA engine registers the AAS (atomic action specification, analogous to AES) at an AEM. The AEM registers at one or more domain brokers (for the domain, or for *domain:action-name*), and the domain brokers submit these actions like events to the AEM. Thus, only the domain broker must be aware if a registration by the AEM is concerned with an event or an action (which is known by the ontology). The ACA-aware ECA engine is aware whether it deeals ith an ECA or an ACA rules, but it does not make a difference in the processing.

Actions separated from Events. An alternative processing can be applied in a local, restricted environment: the ECA engine provides a separate method for registering ACA rules. The domain broker registers its ACA rules at such an ACA-enabled ECA engine. The ACA-enabled ECA engine registers the AAS (atomic action specification, analogous to AES) at an AAM (which is similar to an AEM, but does only matching for a given formalism and does not register the action at a domain broker). The ECA rule remembers the AEM. At runtime, the domain broker sends actions that are defined via ACA rules to the ACA/ECA-engine which forwards them to the AEMs.

Again, the AAM could be implemented as an extension of an AEM which distinguishes between events and actions by providing separate interfaces.

Discussion.

- as long as for each domain, there is only one domain broker, this is sufficient.
- if there are multiple domain brokers that support an ontology, *each* of them would register the rule. In the same way, each of them would be told to execute a composite action, and each of them would notify the ACA engine (yielding n invocations; even worse if they employ different ACA/ECA services).

The issue of duplicating rules, events and actions will be discussed in Section A.6.6.

Chapter (Appendix A: ECA Framework) A.6

Web Architecture, Ontology, Language and Service Metadata

The Semantic Web as a whole, and also its dynamic aspects as implemented in the ECA framework combine a multitude of ontologies. Domains and languages are themselves resources (identified by their namespace URI). In this section, we propose an ontology of languages and language-related notions together with a Web-Service-based architecture where each language is associated with one or more Web Services that are "responsible" for the language.

A.6.1 Ontology of Languages and Services

A.6.1.1 The Ontology

The concepts of the Framework are described in a ontology. There are three main kinds of things:

- *domain ontologies*, e.g. travel:, uni: that provide concepts (classes), properties/relationships, atomic events and actions. They have been described in Chapter A.5.
- *languages*, e.g. ECA-ML, SNOOP, the relational algebra, or CCS, that provide means for structuring composite events, queries, actions, or even processes. These languages can also been seen as ontologies.

  Additionally, on a lower level, there are actual *programming languages* like XQuery, SQL, or Java, and formalisms e.g. for atomic event matching or first-order logic predicates that do not define a language ontology, but are just languages.
- *services* that actually implement the languages. Each kind of service offers specific *tasks* related to that language.

We focus here mainly on the component and domain languages for the Framework, but most considerations hold also for any language in the XML world. The ontology will be given in OWL below.

Languages in the ECA Framework. The ECA framework distinguishes several classes of languages (as shown in Figure A.2.8):

- the ECA language,
- languages for the event component,
- languages for the query component,

- languages for the test component,
- languages for the action component,
- domain languages.

Apart from the Framework, there may be various other kinds of languages that also form subclasses of "Language". Every language is associated as a resource with its namespace URI (as already used in the examples in the previous section; similarly the W3C languages have their namespace URIs). The current W3C proposals do not specify what is actually behind these URIs. Below, we describe what information is required for using languages in the ECA Framework.

All Languages: Information about the Language. Independent from the characteristics of a language as a "language" or as a domain language, some information about its XML representation should be given.

- all languages that are represented by an XML markup, e.g., all above XML markup examples for component languages, or also languages like XHTML, GML (a markup language for Geography), and the Mondial language: a DTD and/or XML Schema. This can be used for validating language expressions. Note that the schema must in general care for nested expressions of other languages.

Since this information is a document (DTD or XSD), it is just a resource, without further ontology descriptions (which will be discussed below).

Generic Programming and Specification Languages. These are in our Framework mainly ECA-ML and the component languages, but the same considerations also hold for other programming languages.

- specification of the language elements (e.g., the operators that an algebraic language provides).
- specification of the recommended and allowed result types (e.g., "an XML document" (XML transformation languages), or "set of tuples of variable bindings").
- if in the future, an ontology of formal semantics will come up, the description of the semantics of a language should also be given.
- a reference processor that interprets the language. For the component languages, this are the evaluation services. For XSLT, this would be a service where one can send an XSLT document (and optionally an XML instance) and gets back the result. Note that in general, there will be mltiple processors for a language. Thus, for using them throughout the Web, *registries* for processing services provide the link between languages and actual services (see Section A.6.5.1).

Domain Languages. Domain languages are common ontologies, which should also contain actions and events. They are supported by domain nodes, and portals, information brokers, and event brokers.

- markup languages for application domains (e.g., Mondial, traveling, or banking): an RDF/RDFS or OWL description about the notions of that language. For languages that include events and actions, these should also be specified there.

  Languages of application domains (e.g., traveling, or banking) can also be supported by services that provide access to the domain information (static and dynamic), e.g., portals, information brokers and event brokers.

Services. In the same way as there are classes of languages, there are the corresponding classes of services.

For being integrated into the ECA framework, the component Web services must implement appropriate communication for receiving tasks (expressions of the language and variable bindings), additional information (e.g., events), and communicate results as discussed in Sections A.3.1. Additionally there are several properties of services from the technical point of view that are to be indicated in a service description (see Section A.6.5.1).

A.6.1.2 Framework Ontology Metadata

```
<?xml version="1.0"?>
<!-- filename: languages-ontology.rdf -->
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
         xmlns:owl="http://www.w3.org/2002/07/owl#"
         xmlns="http://www.semwebtech.org/2006/meta#"
         xml:base="http://www.semwebtech.org/2006/meta#">
  <owl:Class rdf:ID="ontology"/>
  <owl:Class rdf:ID="domain-ontology">
    <rdfs:subClassOf rdf:resource="#ontology"/>
  </owl:Class>

  <owl:Class rdf:ID="language"/>

  <owl:Class rdf:ID="rule-language">
    <rdfs:subClassOf rdf:resource="#language"/>
  </owl:Class>
  <owl:Class rdf:ID="algebraic-language">
    <rdfs:subClassOf rdf:resource="#language"/>
  </owl:Class>
  <owl:Class rdf:ID="eca-language">
    <rdfs:subClassOf rdf:resource="#rule-language"/>
    <has-service-type rdf:resource="#eca-service"/>
  </owl:Class>
  <owl:Class rdf:ID="event-algebra">
    <rdfs:subClassOf rdf:resource="#algebraic-language"/>
    <has-service-type rdf:resource="#composite-event-detection-engine"/>
  </owl:Class>
  <owl:Class rdf:ID="atomic-event-formalism">
    <rdfs:subClassOf rdf:resource="#language"/>
    <has-service-type rdf:resource="#atomic-event-matcher"/>
  </owl:Class>
  <owl:Class rdf:ID="query-language">
    <rdfs:subClassOf rdf:resource="#language"/>
    <has-service-type rdf:resource="#query-service"/>
  </owl:Class>
  <owl:Class rdf:ID="action-language">
    <rdfs:subClassOf rdf:resource="#language"/>
    <has-service-type rdf:resource="#action-service"/>
  </owl:Class>
  <owl:Class rdf:ID="process-algebra">
    <rdfs:subClassOf rdf:resource="#algebraic-language"/>
    <rdfs:subClassOf rdf:resource="#action-language"/>
    <has-service-type rdf:resource="#action-service"/>
```

```
</owl:Class>

<owl:DatatypeProperty rdf:ID="has_name">
  <rdfs:domain rdf:resource="#language"/>
</owl:DatatypeProperty>
<rdf:Property rdf:ID="has-markup">
  <rdfs:domain rdf:resource="#language"/>
  <rdfs:range rdf:resource="#markup-description"/>
</rdf:Property>
<!-- markup description is DTD cup XMLSchema documents -->
<!-- note that the next also defines "language-class" -->

<rdf:Property rdf:ID="has-service-type">
  <rdfs:domain rdf:resource="#language-class"/>
  <rdfs:range rdf:resource="#service"/>
</rdf:Property>

<rdf:Property rdf:ID="has-operator">
  <rdfs:domain rdf:resource="#algebraic-language"/>
  <rdfs:range rdf:resource="#operator"/>
</rdf:Property>
<owl:Class rdf:ID="operator"/>

<rdf:Property rdf:ID="recommended-result-type">
  <rdfs:domain rdf:resource="#language"/>
  <rdfs:range rdf:resource="#result-type"/>
</rdf:Property>
<rdf:Property rdf:ID="allowed-result-type">
  <rdfs:domain rdf:resource="#language"/>
  <rdfs:range rdf:resource="#result-type"/>
</rdf:Property>
<owl:Class rdf:ID="result-type"/>

<rdf:Description rdf:ID="variable-bindings">
  <rdf:type rdf:resource="result-type"/>
</rdf:Description>
<rdf:Description rdf:ID="answers">
  <rdf:type rdf:resource="result-type"/>
</rdf:Description>
<rdf:Description rdf:ID="result-set">
  <rdf:type rdf:resource="result-type"/>
</rdf:Description>

<!-- services are also classified -->
<owl:Class rdf:ID="service"/>
<owl:Class rdf:ID="service-class"/>

<owl:Class rdf:ID="eca-service">
  <rdf:type rdf:resource="#service-class"/>
  <rdfs:subClassOf rdf:resource="#service"/>
</owl:Class>
<owl:Class rdf:ID="composite-event-detection-engine">
  <rdf:type rdf:resource="#service-class"/>
  <rdfs:subClassOf rdf:resource="#service"/>
```

```
  </owl:Class>
  <owl:Class rdf:ID="atomic-event-matcher">
    <rdf:type rdf:resource="#service-class"/>
    <rdfs:subClassOf rdf:resource="#service"/>
  </owl:Class>
  <owl:Class rdf:ID="query-service">
    <rdf:type rdf:resource="#service-class"/>
    <rdfs:subClassOf rdf:resource="#service"/>
  </owl:Class>
  <owl:Class rdf:ID="process-algebra-service">
    <rdf:type rdf:resource="#service-class"/>
    <rdfs:subClassOf rdf:resource="#service"/>
  </owl:Class>
  <owl:Class rdf:ID="domain-broker">
    <rdf:type rdf:resource="#service-class"/>
    <rdfs:subClassOf rdf:resource="#service"/>
  </owl:Class>
  <owl:Class rdf:ID="domain-node">
    <rdf:type rdf:resource="#service-class"/>
    <rdfs:subClassOf rdf:resource="#service"/>
  </owl:Class>

  <!-- services provide tasks (that are identified by their urls)
    that have a name and a task description -->
  <owl:Class rdf:ID="task"/>
  <owl:Class rdf:ID="task-description"/>

  <rdf:Property rdf:ID="provides-task">
    <rdfs:domain rdf:resource="#service"/>
    <rdfs:range rdf:resource="#task"/>
  </rdf:Property>

  <rdf:Property rdf:ID="meta-provides-task">
    <rdfs:domain rdf:resource="#service-class"/>
    <rdfs:range rdf:resource="#task"/>
  </rdf:Property>

  <rdf:Property rdf:ID="has-task-description">
    <rdfs:domain rdf:resource="#task"/>
    <rdfs:range rdf:resource="#task-description"/>
  </rdf:Property>
</rdf:RDF>
```

[Note that the above XML/RDF is valid "striped RDF" and can be validated and visualized with the RDF validator at `http://www.w3.org/RDF/Validator/`]

A.6.2 Architecture and Processing: Cooperation between Resources

Rules can be evaluated locally at the nodes where they are stored, or they can be registered at some *rule evaluation service*. The rule evaluation engine –both local or as a service– then manages

the actual handling of rules based on the namespace URI references or the language attributes. As described above, every component (i.e., events, conditions, and actions) carries the information of the actual language it uses in its xmlns:*namespace* URI (note that this even allows for nested use of operations of *different* event algebras). Via *Languages and Services Registries (LSRs)* (cf. Section A.6.5.1), the URLs and communication details of language processors can be found out. In the following, we describe the most general Web-wide architecture. Section A.6.3 describes variants that can especially applied in "closed" environments.

A.6.2.1 Event Detection

For event detection, at least two resources (or services) must cooperate: Event detection splits into matching the specification against events (*event algebra* and *atomic event matching*, and obtaining the relevant atomic events. Thus, the event component processor must be aware of the relevant atomic events.

When a rule or an event specification is submitted for registration, this has to be accompanied by information which resource(s) provide the atomic events (e.g., "@snoop: my bank is at *uri*, please supervise my account and tell me if a composite event *ev* occurs), or the detection service even has to find appropriate event sources (by the namespaces of the atomic events). The detection service then contacts them directly. The latter is e.g. appropriate for booking travels where the client is in general not aware of all relevant events (e.g., "@snoop: you know better than me who is well-informed about events relevant for traveling, please detect the event ev_{travel} for me"), as illustrated in Figure A.6.1: A client registers a rule (in the travel domain) at the ECA engine (Step 1.1). The ECA engine again submits the event component to the appropriate CED service (1.2), here, a SNOOP service. The SNOOP engine looks at the namespaces of the atomic events and sees that the travel ontology is relevant. The SNOOP service registers all atomic event patterns at the appropriate AEM (1.3). The AEM contacts a travel event broker (1.4) who keeps it informed (2.2) about atomic events (e.g., happening at Lufthansa (2.1a) and SNCF (2.1b). The AEM matches the events against the registered patterns, and in case of a success, reports the matched event and the extracted variable bindings to the SNOOP service (3). Only after detection of the registered composite event, the SNOOP submits the result to the ECA engine (4).

A.6.2.2 Query Processing

For queries, the opaque case using a common query language like XPath or XQuery on a given URL (XML document) is frequent. These query langauges often directly supported by the data sources. Otherwise, a "free" query service must be used. The handling of opaque queries in the prototype is described in Section A.7.1.2.

For the non-opaque case, the basic schema is similar to event processing, consisting of the algebraic part and querying the actual concepts. Since every concept comes from an ontology (with namespace), the appropriate domain broker can be contacted. The graphics in Figure A.6.1 does not show the evaluation of a query component.

A.6.2.3 Action Processing

The basic schema is symmetric to event processing, consisting of the algebraic part and executing the actual actions (which requires to determine *where* the action has to be executed): The action is submitted to an appropriate action language service (here: CCS (5.1)) that in turn submits the atomic actions to domain broker (travel, 5.2a) which forwards the actions to the respective domain nodes (5.3a), and to the domain-independent services, here smtp (5.2b) which then sends a message.

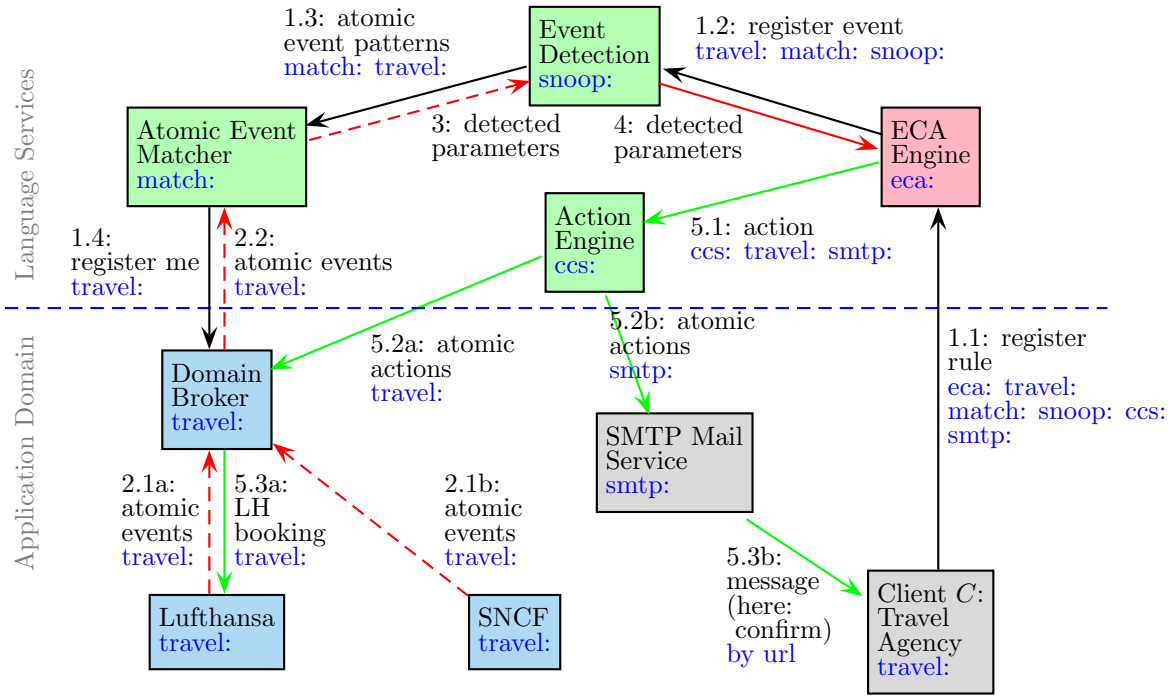


Figure A.6.1: Communication: Event Processing

A.6.3 Architectural Variants

In addition to the above architecture, multiple intermediate variants are possible that combine functionality. In general, these are obtained by changing the communication paths and/or combining functionality in a node. The following paragraphs list (not necessarily in a systematic way) some variants.

Domain-Broker/Application-Centered. An application or portal can provide brokering service *together* with (selected) E, Q&T, and A languages (note that portals often already provide a query interface).

For event detection, a portal can offer to "process" composite event specifications. The client then submits its composite event specification to the portal. For processing it, it can either directly implement a CEL, or employ another service. The main point is that the portal is aware of all relevant events in the application domain and can feed them directly into the event detection.

Example 41 (Banking with Event Detection) *A bank can e.g. offer such functionality. Then, customers can place their composite events there and say "@bank: please trace the following composite event in language L on my account" (and employ a suitable event detection service for L)".*

The same holds for process specifications in the action part, when all actions have to be executed on the same portal or even on the same node.

Integrating AEM Services. For the actual location of the atomic event detection, there are again several alternatives: The separated architecture uses separate *Atomic Event Matchers (AEMs)* that implement an atomic event specification formalism, and that themselves are informed about the events by the application services. Instead, this "simple" matching can also be integrated either with the CED or with the domain portal:

- application services (e.g. for travel:) provide matching functionality (having an AEM for some specification formalism), or
- event brokers (e.g. for travel:) can provide matching functionality (also having an AEM for some specification formalism).

ECA-Engine-Centered. In this case, there is no domain broker, but the ECA engine plays the "central" role (see Figure A.6.2): clients C register rules to be "supervised" at a rule execution service R. For handling the event component, R reads the language URI of the event component, and registers the event component at the appropriate event detection service S (note that a rule service that evaluates rules with events in different languages can employ several event detection mechanisms).

During runtime, the clients C forward all events to R, that in turn forwards them to all event detection engines where it has registered event specifications for C, amongst them, S. Relevant events from outside can be "imported" from an external domain broker.

S is "application-unaware" and just implements the semantics of the event combinators for the incoming, non-interpreted events. In case that a (composite) event is eventually detected by S, it is signalled together with its result parameters to R. R takes the variables, and evaluates the query&test (analogously, based on the respective languages), and finally executes the action (or submits the execution order to a suitable service). In the same way, defined actions (for application of ACA rules) are communicated via the ECA engine.

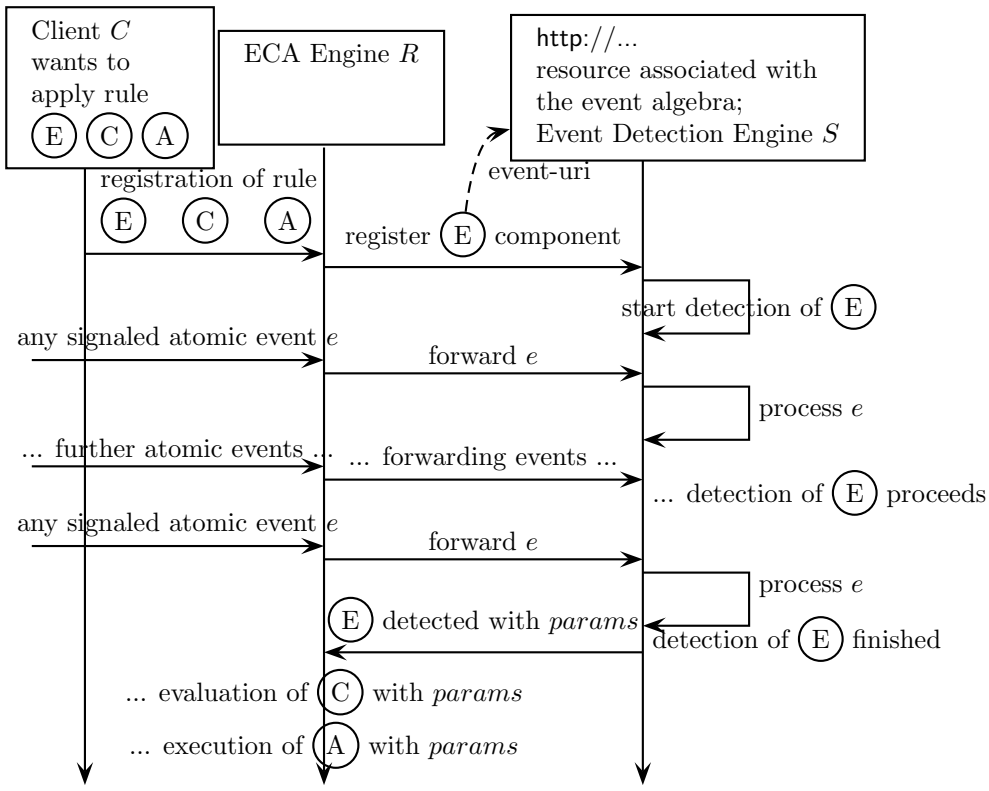


Figure A.6.2: Dependent Communication

The dependent variant is especially preferable in closed environments where e.g. an application service (a university or an airline) wants to apply (i) only own rules to (ii) a central management of events (that can be both local events and events arriving at a given input interface).

A.6.4 Service Interfaces and Functionality [Subject to Change]

The material in this section is subject to change. It will probably be validated and extended during further implementation.

For a uniform communication between the different kinds of services, each service type provides certain tasks. The languages & service ontology specifies which tasks are provided by the different classes of services. The *actual* communication interfaces of *each service* are then handled by *Language&Service Registries (LSRs)* (see Section A.6.5.1) which are based on the lists of tasks given next. The complete ontology is given in RDF in Section A.6.4.5.

A frequent pattern is to register/submit components and subexpressions to appropriate services. For this, addresses and details of the communication format must be specified. the Additionally, the answers must be received (note that the respective addresses are communicated with the Reply-To, but details of the communication format must be specified. For that, also the tasks for receiving information are listed below.

In the below list, [R] means to provide a service or to receive information while [S] means just to call a service (send information). Optionally, a service should give the description of the tasks offered by them (called when some other service wants to invoke that task); the contents of these task descriptions is discussed in Section A.6.5.1 (otherwise the owner of the service has to submit this information manually to an LSR).

A.6.4.1 ECA Services

An ECA service must implement the ECA-ML language and adhere to the abstract semantics of ECA rules given in Sections A.3.1 and A.3.4.

A.6.4.1.1 Upper Interface:

Registration of Rules: [R] a rule can be registered by a predefined URI (e.g., if the rule is part of a rule set of a domain service, its URI will be determined there), or just as an XML fragment or an RDF graph (e.g., by a user). In the latter case, it is associated with a URI relative to the ECA service. This URI is communicated to the registrant in the response.

Deregistration of Rules: [R] Rules can be deregistered using their URI.

Disabling and Enabling Rules: [R, Optional] Rules can be disabled and enabled using their URI. Disabling a rule means to cancel all ongoing dependent event detections. Rule instances that are under actual execution at this point will be completed. Enabling a rule means that detection processes will be started at this moment. Past atomic events do not contribute to the detection. Note that a "grey area" exists for "past" events that become known only now to a service.

A.6.4.1.2 Lower Interface:

Registration/Evaluation of Components: The individual components must be submitted to appropriate services. Additionally, the answers must be received:

- [S] Composite event component: register at composite event detection service (CED).
- [S] Atomic event component: in case that the event component is just an atomic event, it can either be registered via a CED, or directly at an AEM, using the same downward communication for atomic events that a CED uses (see Section A.4.3).
- [R] receive answers for detected events.
- [S] Query components: submit query together with bound variables to appropriate service and receive answer.

- [S] Test component: analogous. Often the test component just uses simple (comparison) predicates between variables that can be evaluated locally.
- [S] Action component: submit action together with bound variables to appropriate service. Optionally, an answer can be received (success/failure) which can be used in transactional environments.
- [S] optionally, query and action components can be registered a priori, and actual evaluation/execution the refers to an id and just submits the current variable bindings. (In that case, the choice of the service is done once, and must be remembered.)

Validation of Components: [S, Optional] Submit component markup/code to appropriate service for validation. Note that validation has to be done hierarchically in the same way as the evaluation.

A.6.4.2 (Algebraic) Component Languages/Services (General)

- upper interface: receive requests:
  - CED: [R] registrations, deregistrations
  - AEM: [R] registrations, deregistrations
  - QE: [R] queries,
    optional: [R] registrations, deregistrations, invocations for given variable bindings
  - CAE: [R] composite action specifications,
    optional: registrations, deregistrations, invocations for given variable bindings
  - [R, optional] a method that validates a given statement,
  - [R, optional] a method that returns for a given statement the used variables, the input variables, the output variables, and the returned variables (based on the declarations of the atomic expressions in the component),
- [R] upper interface: send answers.
- lower interface: communicate with the domains
  - CED: register/deregister AEDs at AEMs [S], receive answers [R]
  - AEM: register/deregister events at EBs [S], receive events [R]
  - QE: ask domain nodes/portals/brokers [S], receive answers [R]
  - CAE: forward atomic actions to be invoked to domain nodes/portals/brokers [S]. For some formalisms: communication of embedded event patterns and queries [S/R].

A.6.4.3 Domain Brokers

The domain brokers act as mediators between domain nodes and component language services.

Initialization.

- [R] register an ontology (in most cases when the domain broker is initialized)
  input: the ontology

Information about Domain Nodes.

- [R] receive messages from domain nodes which classes, properties, event types and action types they support
  - prototype: as a whole description (cf. Section A.5.2)
  - future: as individual messages (in case a node changes its behavior)

Mediation of Requests.

- [R] receive registrations/deregistrations for events (optionally: of given types),
- [R] receive events and [S] forward them to AEMs,
- [S] send events to registered AEMs,
- [R] receive queries, and [S] evaluate them against domain, and return answers,
- [R] receive atomic actions to be executed and [S] forward them to domain nodes.

A.6.4.4 Domain Services

- provide a service that implements the domain and provides an appropriate communication interface (calling an atomic "thing" with some parameters).
  - [R] answering queries,
  - [S] providing events (either upon registration or to a fixed event broker),
  - [R] executing actions of the domain (submitted in XML).
  - [S, optional] send list of supported notions of the ontology to a domain broker (otherwise: support for the whole namespace is assumed).

A.6.4.5 The Services Ontology

The services ontology defines a "name" (= resource identifier) for each of the tasks of each kind of service. This will be used in the LSR when describing how to invoke a certain task of a certain service.

```
<?xml version="1.0"?>
<!-- filename: services-ontology.rdf -->
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
         xmlns:owl="http://www.w3.org/2002/07/owl#"
         xmlns="http://www.semwebtech.org/2006/meta#"
         xml:base="http://www.semwebtech.org/2006/meta#">

  <rdf:Description rdf:about="#eca-service">
    <meta-provides-service rdf:resource="#eca-service/register-rule"/>
    <meta-provides-service rdf:resource="#eca-service/deregister-rule"/>
    <meta-provides-service rdf:resource="#eca-service/disable-rule"/> <!-- opt -->
    <meta-provides-service rdf:resource="#eca-service/enable-rule"/> <!-- opt -->
    <meta-provides-service rdf:resource="#eca-service/receive-detected-event"/>
    <meta-provides-service rdf:resource="#eca-service/receive-query-answer"/>
    <meta-provides-service rdf:resource="#eca-service/give-service-description"/> <!-- opt -->
  </rdf:Description>
  <rdf:Description rdf:about="#composite-event-detection-engine">
    <meta-provides-service rdf:resource="#ced-engine/register-event-pattern"/>
    <meta-provides-service rdf:resource="#ced-engine/deregister-event-pattern"/>
    <meta-provides-service rdf:resource="#ced-engine/validate-pattern"/> <!-- opt -->
    <meta-provides-service rdf:resource="#ced-engine/analyze-variables"/> <!-- opt -->
    <meta-provides-service rdf:resource="#ced-engine/receive-detected-event"/>
    <meta-provides-service rdf:resource="#ced-engine/give-service-description"/> <!-- opt -->
  </rdf:Description>
  <rdf:Description rdf:about="#atomic-event-matcher">
```

```
    <meta-provides-service rdf:resource="#aem/register-event-pattern"/>
    <meta-provides-service rdf:resource="#aem/deregister-event-pattern"/>
    <meta-provides-service rdf:resource="#aem/validate-pattern"/> <!-- opt -->
    <meta-provides-service rdf:resource="#aem/analyze-variables"/> <!-- opt -->
    <meta-provides-service rdf:resource="#aem/receive-event"/>
    <meta-provides-service rdf:resource="#aem/give-service-description"/> <!-- opt -->
  </rdf:Description>
  <rdf:Description rdf:about="#query-service">
    <meta-provides-service rdf:resource="#qs/evaluate-query"/>
    <meta-provides-service rdf:resource="#qs/register-query"/> <!-- opt -->
    <meta-provides-service rdf:resource="#qs/deregister-query"/> <!-- opt -->
    <meta-provides-service rdf:resource="#qs/invoke-query"/> <!-- opt -->
    <meta-provides-service rdf:resource="#qs/validate-pattern"/> <!-- opt -->
    <meta-provides-service rdf:resource="#qs/analyze-variables"/> <!-- opt -->
    <meta-provides-service rdf:resource="#qs/give-service-description"/> <!-- opt -->
  </rdf:Description>
  <rdf:Description rdf:about="#action-service">
    <meta-provides-service rdf:resource="#action-service/execute-action"/>
    <meta-provides-service rdf:resource="#action-service/register-action"/> <!-- opt -->
    <meta-provides-service rdf:resource="#action-service/deregister-action"/> <!-- opt -->
    <meta-provides-service rdf:resource="#action-service/receive-query-answer"/> <!-- opt -->
    <meta-provides-service rdf:resource="#action-service/receive-detected-event"/> <!-- opt -->
    <meta-provides-service rdf:resource="#action-service/validate-pattern"/> <!-- opt -->
    <meta-provides-service rdf:resource="#action-service/analyze-variables"/> <!-- opt -->
    <meta-provides-service rdf:resource="#action-service/give-service-description"/> <!-- opt -->
  </rdf:Description>
  <rdf:Description rdf:about="#domain-broker">
    <meta-provides-service rdf:resource="#domain-broker/register-ontology"/> <!-- init -->
    <meta-provides-service rdf:resource="#domain-broker/register-ontology"/> <!-- opt -->
    <meta-provides-service rdf:resource="#domain-broker/receive-bulk-support-information"/>
    <meta-provides-service rdf:resource="#domain-broker/update-support-information"/> <!-- opt -->
    <meta-provides-service rdf:resource="#domain-broker/register-for-event"/>
    <meta-provides-service rdf:resource="#domain-broker/receive-event"/>
    <meta-provides-service rdf:resource="#domain-broker/execute-query"/>
    <meta-provides-service rdf:resource="#domain-broker/receive-query-answer"/>
    <meta-provides-service rdf:resource="#domain-broker/execute-action"/>
    <meta-provides-service rdf:resource="#domain-broker/give-service-description"/> <!-- opt -->
  </rdf:Description>
  <rdf:Description rdf:about="#domain-node">
    <meta-provides-service rdf:resource="#domain-node/receive-query"/>
    <meta-provides-service rdf:resource="#domain-node/receive-action"/>
    <meta-provides-service rdf:resource="#domain-node/give-service-description"/> <!-- opt -->
  </rdf:Description>
</rdf:RDF>
```

[Note that the above XML/RDF is valid "striped RDF" and can be validated and visualized with the RDF validator at `http://www.w3.org/RDF/Validator/`]

A.6.5 Locating and Contacting Language Services [Subject to Change]

The material in this section is subject to change. It will probably be validated and extended during further implementation.

Rules and component expressions consist of nested expressions in several languages/namespaces (ECA language, event language, atomic event specification language, domain vocabulary). The handover between services has to take place at the "namespace borders". The processor that processes the surrounding language has to do the following:

- identify the embedded fragment (as XML subtree), and
- determine what processor is responsible to process it.

For performing the latter, the namespace and/or language information of the subtree contains the information about the used language:

- embedded algebraic languages: namespace of the root element of the subexpression,
- embedded atomic event specifications: namespace of the root element of the subexpression, or language attribute (cf. Section A.4.2.2),
- opaque expressions: language attribute.

If a service is identified, the actual communication details (e.g., where to send the message, how to wrap it, and whether it supports multiple tuples of variable bindings) are determined. All required information is available in *Languages and Services Registries (LSRs)* that are discussed in Section A.6.5.1. Note that the required algorithms are the same for every invocation (done by ECA engines, action engines (if the action embeds events or queries), and also for nested event components). This "trader" functionality is provided by a *Generic Request Handler, GRH* (Section A.6.5.4) that can be implemented as a standalone service or provided as an instance of a downloadable Java class.

A.6.5.1 Language&Service Registries

The languages and the actual service instances are handled by *Language&Service Registries (LSRs)* [in general one LSR would be sufficient, but as there is no central thing in the Semantic Web, it cannot be expected that there is a central LSR – for running any kind of business it is sufficient to know a *good* LSR - or to know an ECA engine that knows a good LSR. Language&Service Registries will -in the final version- be services where others can get/look up information how some task wrt. a language can be requested:

- input: a language (by its URI) and a "task", e.g., "register an event component (in that language, for future detection)",
- answer: the task description in XML or RDF format.

The ontology discussed in Section A.6.1.2 specifies [in the future] for each kind of service all available tasks (mandatory or optional). Note that an actual service can also provide functionality of several service classes.

Identification: Mapping from Languages to Services. A first task of the LSR is to provide a mapping from the languages to actual services that can handle the tasks. This includes the *registration* of services. For each registered service, a more detailed description how to employ its tasks is then also included in the LSR.

Communication Modalities. If a service for a certain language and a certain functionality is identified, tha actual communication (which URI, which format) must be determined.

A.6.5.2 Interface Descriptions of Individual Tasks

Framework-Aware Services should provide a service description for each task that deals with the following issues:

- identification of the task via its URI defined in the *Services Ontology* in Section A.6.4.5.
- URLs where the respective input is expected (relative to the service url; items that are not given default to the service url),
- format of the input (wrapping),
- functionality (e.g., built-in join functionality for supporting sideways-information-passing strategies in queries).

Characteristics: Input Formats. Every request contains the information where the answer must go (Reply-To), an identifier to associate the answer (in asynchrounous cases), the message contents itself (which is in often an XML fragment in the respective language), and optional variable bindings. The input format specifies how these components are communicated with an individual given task:

- `Reply-To`: header or body (default: header as X-Reply-To)
- `Subject`: header or body (default: header as X-Subject)
- `input-format` [CED, AEM, QL, T, A]: message input formats requested by component services:
  - default: `item *` – the message body contains a sequence of elements (which elements should be clear from the context situation).
    - In most cases, this is a fragment of the respective language, and optionally variable bindings,
    - sometimes (e.g. for the event stream), it's just any elements.
  - optional: the above-mentioned items are wrapped into an element with a given name: `element` *`name`* (with arbitrary namespace - the receiver will at most check the local-name [sometimes even not this if the element is just needed to have any root node])

Characteristics: Functionality.

- `variables` [CED, AEM, Q, T, A]: are variable bindings accepted?
  - `*`: a set of tuples of variable bindings in the markup given in Section A.3.2.1 is accepted.
  - `1`: one tuple of variable bindings in the markup given in Section A.3.2.1 is accepted at a time (means that the calling service must iterate).
  - `no`: no variable bindings are accepted (means that the calling service must iterate and replace the variables by their values as strings in the code).
  - `ignore`: variable bindings are ignored (e.g., when deregistering somtheing that has been registered with variable bindings.
- `join-enabled` [event, event matcher, query]: does it make sense to send variable bindings that are used for join semantics?
  - `yes`: the service implements join semantics. Variable bindings can be sent for optimized answering.
  - `no` (default): send only the `input` variables that will be bound by the service.

- services that accept only a single input tuple should indicate whether the returned tuple(s) are an extension of the input tuple, or if they do not "echo" all input variables:
  `return-input-vars` [event, event matcher, query]: are the input variables returned with the answer? "echoing" these variables is in general needed for assigning the returned tuples to tuples on the ECA level; see Section A.3.1.5 and requirements on Page 48.

  - `yes`: result can immediately be joined.
  - `no`: the calling service must itself care about the assignment of the returned answers with the corresponding tuples of variable bindings.

- `asynchronous` [query, test]: is the answer necessarily returned immediately, or can it be returned asynchronously (in this case, the immediate answer is "OK", and the result is latter returned with appropriate identification (Subject).

- `return-incomplete` [query, test]: does the answer contain/consider all tuples, or is it possible that several results are sent?

  - for ECA services: `yes/no` indicate if it can process incomplete answers (if this is indicated with the answer). Default: no.
  - for query/text services: `yes/no` indicate if the service probably returnd incomplete answers (and indicates this with the answer). Default: no.

Example 42 (Service Description) *The following is a fragment of the service description of a CED (assumed at URL* url*) that allows to register CESs at* url*/register. A registration message must have the following format:*

- *Reply-To-address must be given in the HTTP message header,*
- *the Subject (=identifier) must be given as an element in the body,*
- *the task itself is always given in the body,*
- *and one tuple of variable bindings can be given (which is common for CEDs),*
- *and the whole body is encapsulated in a single <register> element*

Note that line 13 refers to `http://www.semwebtech.org/2006/languages#qs/register-query` *due to the setting of xml:base.*

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:languages="http://www.semwebtech.org/2006/languages#"
         xmlns="http://www.semwebtech.org/2006/lsr#"
         xml:base="http://www.semwebtech.org/2006/languages#">
 <languages:query-service rdf:about="http://foo.nop/fancy-query-engine">
   <!-- sample: only one task description
        clients can register queries at: -->
   <has-task-description>
     <task-description>
       <!-- reference to the described task according to the Service Ontology -->
       <task rdf:resource="#qs/register-query"/>
       <!-- url of the service where the task can be actually invoked -->
       <provided-at rdf:resource="./register"/>
       <Reply-To>header</Reply-To>
       <Subject>body</Subject>
       <input>element register</input>
```

```
      <variables>1</variables>
      <!-- means: Subject in the message header, contents is
            <register>$event$,$var-bindings$</register> with
           max. 1 tuple of variable bindings -->
    </task-description>
  </has-task-description>
 </languages:query-service>
</rdf:RDF>
```

[Note that the above XML/RDF is valid "striped RDF" and can be validated and visualized with the RDF validator at `http://www.w3.org/RDF/Validator/` *]*

A message thus could look as follows, sent to `url/register`*:*

```
X-Reply-To: <http://bla.nop/i-want-my-answers-here>
<register>
 <Subject>this-is-my-id-for-this-task-007</Subject>
 <evt:operator xmlns:evt="....">
   contents
 </evt:operator>
 <log:variable-bindings xmlns:log="...">
   <log:tuple>...</log:tuple>
 </log:variable-bindings>
</register>
```

When a service registers itself at a LSR, it either submits its SD or a reference to it.

Non-Framework-Aware Services. In case that Non-Framework-Aware Services are used in a rule, the ECA engine needs a separate SD. For this, the ECA engine provides a task where SDs can be submitted which is then associated with the service uri uri_s e.g. by an RDF tuple.

- the framework maintainers can add SDs for services that are used frequently,
- clients who submit a rule (whose namespace URI refer to such a service) can submit the specification of the service.

A.6.5.3 The Languages and Services Registry RDF Model

Since the metadata structure is complex, it its preferably expressed in RDF. Nevertheless, for a first prototype on the XML level, we assume that some metadata is available as an XML file.

The service descriptions are represented in the prototype as an XML/RDF file at `http://www.semwebtech.org/2006/`
For each language, the following is given:

- language URI (mandatory),
- language name , e.g. "XQuery", "SNOOP", may be null,
- type of language (ECA, CEL, AESL, ...),
- List of appropriate services by URIs and their Service Descriptions.

Naming Schema. The languages developed in this framework are located in the "www.semwebtech.org" domain (which is registered by the DBIS group at Göttingen University); there also most of the reference services are running.

Namespaces refer to "http://www.semwebtech.org/*/2006" (to have room for new versions in the following years). The language namespaces are maintained as "hash namespaces" (according to the terminology in [56]) and described by RDF/RDFS files accessible at these URLs.

```
<?xml version="1.0"?>
<!-- filename: lsr.rdf -->
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:languages="http://www.semwebtech.org/2006/meta#"
         xmlns:lsr="http://www.semwebtech.org/2006/lsr#"
         xmlns="http://www.semwebtech.org/2006/lsr#"
         xml:base="http://www.semwebtech.org/2006/meta#">

<languages:eca-language rdf:about="http://www.semwebtech.org/eca/2006/eca-ml">
 <languages:shortname>eca-ml</languages:shortname>
 <languages:name>ECA Markup Language</languages:name>
 <languages:is-implemented-by>
  <languages:eca-service
     rdf:about="http://www.semwebtech.org/eca/2006/services/eca-engine">
   <has-task-description>
    <task-description>
      <task rdf:resource="#eca-service/register-rule"/>
      <provided-at rdf:resource="????"/>
      <Reply-To>????</Reply-To>
      <Subject>????</Subject>
      <input>????</input>
      <variables>?</variables>
    </task-description>
   </has-task-description>
   <has-task-description>
    <task-description>
      <task rdf:resource="#eca-service/deregister-rule"/>
      <provided-at rdf:resource="????"/>
      <Reply-To>????</Reply-To>
      <Subject>????</Subject>
      <input>????</input>
      <variables>?</variables>
    </task-description>
   </has-task-description>
   <has-task-description>
    <task-description>
      <task rdf:resource="#eca-service/receive-detected-event"/>
      <provided-at rdf:resource="????"/>
      <Reply-To>????</Reply-To>
      <Subject>????</Subject>
      <input>????</input>
      <variables>?</variables>
    </task-description>
   </has-task-description>
   <has-task-description>
    <task-description>
      <task rdf:resource="#eca-service/receive-query-answer"/>
```

```
      <provided-at rdf:resource="????"/>
      <Reply-To>????</Reply-To>
      <Subject>????</Subject>
      <input>????</input>
      <variables>?</variables>
    </task-description>
   </has-task-description>
  </languages:eca-service>
 </languages:is-implemented-by>
</languages:eca-language>

<languages:event-algebra
  rdf:about="http://www.semwebtech.org/eca/2006/snoopy">
 <languages:name>SNOOP (from the Sentinel Database System)</languages:name>
 <languages:is-implemented-by>
  <languages:composite-event-detection-engine
        rdf:about="http://www.semwebtech.org/eca/2006/services/snoopy">
   <has-task-description>
    <task-description>
      <task rdf:resource="#ced-engine/register-event-pattern"/>
      <provided-at
        rdf:resource="http://www.semwebtech.org/eca/2006/services/snoopy/????"/>
      <Reply-To>????</Reply-To>
      <Subject>????</Subject>
      <input>????</input>
      <variables>ignore</variables>
    </task-description>
   </has-task-description>
   <has-task-description>
    <task-description>
      <task rdf:resource="#ced-engine/deregister-event-pattern"/>
      <provided-at
        rdf:resource="http://www.semwebtech.org/eca/2006/services/snoopy/????"/>
      <Reply-To>????</Reply-To>
      <Subject>????</Subject>
      <input>????</input>
      <variables>ignore</variables>
    </task-description>
   </has-task-description>
   <has-task-description>
    <task-description>
      <task rdf:resource="#ced-engine/receive-detected-event"/>
      <provided-at
        rdf:resource="http://www.semwebtech.org/eca/2006/services/snoopy/????"/>
      <Reply-To>????</Reply-To>
      <Subject>????</Subject>
      <input>item</input>
      <variables>*</variables>
    </task-description>
   </has-task-description>
  </languages:composite-event-detection-engine>
 </languages:is-implemented-by>
</languages:event-algebra>
```

```
<languages:atomic-event-formalism
  rdf:about="http://www.semwebtech.org/eca/2006/aes-xmlql">
 <languages:name> Atomic Event Detection by XML-QL</languages:name>
 <languages:is-implemented-by>
  <languages:atomic-event-matcher
     rdf:about="http://www.semwebtech.org/eca/2006/services/aem-xmlql">
   <!-- this aem receives each task at a separate url. Tasks are
     not wrapped into any additional element, events come as elements
     <namespace:localname> -->
   <has-task-description>
    <task-description>
      <task rdf:resource="#aem/register-event-pattern"/>
      <provided-at
        rdf:resource="http://www.semwebtech.org/eca/2006/services/aem-xmlql/register"/>
      <Reply-To>body</Reply-To>
      <Subject>body</Subject>
      <input>item *</input>
      <variables>1</variables>
    </task-description>
   </has-task-description>
   <has-task-description>
    <task-description>
      <task rdf:resource="#aem/deregister-event-pattern"/>
      <provided-at
        rdf:resource="http://www.semwebtech.org/eca/2006/services/aem-xmlql/deregister"/>
      <Reply-To>body</Reply-To>
      <Subject>body</Subject>
      <input>item *</input>
      <variables>ignore</variables>
    </task-description>
   </has-task-description>
   <has-task-description>
    <task-description>
      <task rdf:resource="#aem/receive-event"/>
      <provided-at
        rdf:resource="http://www.semwebtech.org/eca/2006/services/aem-xmlql/events"/>
      <!-- other parameters are not given. Events are simply sent. -->
    </task-description>
   </has-task-description>
  </languages:atomic-event-matcher>
 </languages:is-implemented-by>
</languages:atomic-event-formalism>

<languages:atomic-event-formalism
  rdf:about="http://www.semwebtech.org/eca/2006/aes-xpath">
 <languages:name> Atomic Event Detection by XML-QL</languages:name>
 <languages:is-implemented-by>
  <languages:atomic-event-matcher
     rdf:about="http://www.semwebtech.org/eca/2006/services/aem-xpath">
   <!-- this aem receives everything at the same url. Tasks are
     wrapped as <register> or <deregister> (without namespace), events
     come as elements  <namespace:localname> -->
   <has-task-description>
    <task-description>
```

```
      <task rdf:resource="#aem/register-event-pattern"/>
      <provided-at
        rdf:resource="http://www.semwebtech.org/eca/2006/services/aem-xpath"/>
      <Reply-To>body</Reply-To>
      <Subject>body</Subject>
      <input>element register</input>
      <variables>1</variables>
    </task-description>
   </has-task-description>
   <has-task-description>
    <task-description>
      <task rdf:resource="#aem/deregister-event-pattern"/>
      <provided-at
        rdf:resource="http://www.semwebtech.org/eca/2006/services/aem-xpath"/>
      <Reply-To>body</Reply-To>
      <Subject>body</Subject>
      <input>element register</input>
      <variables>ignore</variables>
    </task-description>
   </has-task-description>
   <has-task-description>
    <task-description>
      <task rdf:resource="#aem/receive-event"/>
      <provided-at
        rdf:resource="http://www.semwebtech.org/eca/2006/services/aem-xpath"/>
      <!-- other parameters are not given. Events are simply sent. -->
    </task-description>
   </has-task-description>
  </languages:atomic-event-matcher>
 </languages:is-implemented-by>
</languages:atomic-event-formalism>

<languages:query-language rdf:about="http://www.w3.org/XQuery">
 <languages:name>XQuery (Opaque)</languages:name>
 <languages:is-implemented-by>
  <languages:query-service
        rdf:about="http://www.semwebtech.org/eca/2006/services/????">
   <has-task-description>
    <task-description>
      <task rdf:resource="#query-engine/evaluate-query"/>
      <provided-at
        rdf:resource="http://www.semwebtech.org/eca/2006/services/????/????"/>
      <Reply-To>????</Reply-To>
      <Subject>????</Subject>
      <input>????</input>
      <variables>*</variables>
    </task-description>
   </has-task-description>
  </languages:query-service>
 </languages:is-implemented-by>
</languages:query-language>

<languages:process-algebra
  rdf:about="http://www.semwebtech.org/eca/2006/ccs">
```

```
 <languages:name>XQuery (Opaque)</languages:name>
 <languages:is-implemented-by>
  <languages:action-service
        rdf:about="http://www.semwebtech.org/eca/2006/services/ccs">
   <has-task-description>
    <task-description>
      <task rdf:resource="#query-engine/evaluate-query"/>
      <provided-at
        rdf:resource="http://www.semwebtech.org/eca/2006/services/ccs/????"/>
      <Reply-To>????</Reply-To>
      <Subject>????</Subject>
      <input>????</input>
      <variables>*</variables>
    </task-description>
   </has-task-description>
  </languages:action-service>
 </languages:is-implemented-by>
</languages:process-algebra>

<languages:domain
   rdf:about="http://www.semwebtech.org/domains/2006/travel">
 <languages:name>Traveling</languages:name>
</languages:domain>

<!-- note that the relationship between domains and application
 services is n:m (a service may implement multiple domains), and
 the relationship between domains and domain broker services is
 also n:m (a broker may support multiple domains) -->

<languages:domain-broker
   rdf:about="http://www.semwebtech.org/eca/2006/services/travel-broker">
   <languages:name>Travel Domain Broker</languages:name>
   <supports-domain>http://www.semwebtech.org/domains/2006/travel</supports-domain>
   <!-- to be extended -->
</languages:domain-broker>

<languages:application-node rdf:about="http://tobeextended.de/orafly">
  <name>Oracle Airlines</name>
  <uses-domain>http://www.semwebtech.org/domains/2006/travel</uses-domain>
  <!-- list of all names in the domain (events, actions, concepts, properties)
      that are supported by that application node -->
  <supports>flight</supports>
  <supports>cancel-flight</supports>
  <supports>flight-is-canceled</supports>
  <supports>canceled-flight</supports>
  <!-- to be extended -->
</languages:application-node>
</rdf:RDF>
```

The LSR can be found as an RDF/XML file at `http://www.semwebtech.org/2006/lsr/lsr.rdf`. [Note that the above XML/RDF is valid "striped RDF" and can be validated and visualized with the RDF validator at `http://www.w3.org/RDF/Validator/`]

The following SPARQL query illustrates the combination of the RDF information described

in this chapter.

```
# call
# jena -q -il RDF/XML -if lsr.rdf languages-ontology.rdf services-ontology.rdf -qf lsr-query.sparql

PREFIX meta: <http://www.semwebtech.org/2006/meta#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX eca: <http://www.semwebtech.org/eca/2006/>
PREFIX lsr: <http://www.semwebtech.org/2006/lsr#>

SELECT ?L ?S ?T ?URL
WHERE {

 ?L rdf:type meta:language .
 ?L meta:is-implemented-by ?S.
 ?S lsr:has-task-description ?TD.
 ?TD lsr:task ?T.
 ?TD lsr:provided-at ?URL.

 }
```

A.6.5.4 Service Brokering: Generic Request Handlers

Given any "task" in some language (e.g., the execution of a query component in XQuery), the LSR serves for identifying a target service, and for obtaining information about the communication. Below, we distinguish two kinds of task handling:

- component language tasks, i.e., (composite) event languages, queries, and (composite) actions,
- specialized auxiliary tasks, such as event matching.

Handling Component Language Tasks

The handling of component language fragments includes the communication of variable bindings in both directions. It occurs in the following cases:

- obviously: execution of a component of an ECA rule,
- detection of events or execution of queries in composite action specifications,
- nested subevents in another language inside a composite event,
- analogously for nested query expressions (e.g., a join between a Datalog query with the result of an XQuery expression).

Thus, it is reasonable to specify and implement this trader functionality separately (instead of intergating it within the ECA engine, which would be sufficient at a first glance). Since the functionality of the *Generic Request Handler (GRH)* is (i) simple, (ii) canonic, and (iii) frequently needed, it is designed as a class that can be instantiated for each kind of service that needs it. ECA engines or action language engines that support embedded queries and events will probably have their "own" instances. In contrast, composite event specifications that use subevents from different languages are expected to be much less frequent, so a CED may not have an own GRH but can use a "free" one (as a Web Service).

Use of the GRH: Integration within an ECA Engine. The functionality and use of the GRH is illustrated here together with an ECA engine, cf. Figure A.6.3.

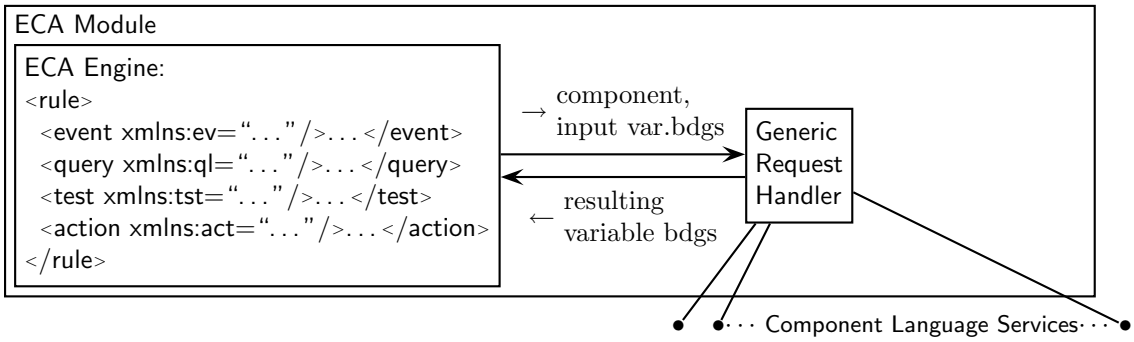


Figure A.6.3: Architecture of an ECA Module with own GRH

The core *ECA engine* implements *only* the handling of variable bindings and the ECA rule semantics. It is supported by a GRH instance for communicating with autonomous component services. The GRH communicates the components to suitable services. It also receives the answers and wraps them into the agreed format of variable bindings (in case of non-framework-aware services).

GRH: Functionality

- dealing with requests coming from the ECA engine, forwarding them to component services (decoding the namespace URL and addressing an appropriate Web Service is hardcoded in the first prototype that supports only a very restricted number of component languages; this will later be separated in the LSR),
- receive answers, transform them into variable bindings (in <variable-bindings> format).

GRH: Interface

- in from above: requests from ECA (complete components in ECA-ML and contents),
- out to below (autonomous services): requests to framework-aware and non-framwork-aware (HTTP) services,
- in from below: results in the <logvars:answers> format from framework-aware component services and unwrapped results from non-frame-aware services (HTTP answers).

Interface of Modules towards their GRH

When the GRH is not a separate service, but a local instance, the invocation of the GRH call can be done as a method call. Additionally, there may be "free" GRH services.

- out: requests to GRH: components in XML Markup of the individual languages,
- in: answers from GRH, in <variable-bindings> format.

Global Cooperation via GRHs.

As discussed above, GRH instances can be embedded into modules that have to deal with (embedded) language fragments, or can be standalone. Thus, not only the ECA engine uses a GRH, but also the actual processing of the action component with its embedded expressions makes use of a

GRH. The handling of embedded <ccs:event>, <ccs:query>, <ccs:test>, and <ccs:action> elements with embedded fragments of other languages is done in the same way as described in Sections A.4. Atomic actions are executed "immediately" by submitting them to the domain nodes (if specified by an URL) or to a domain broker (responsible for the domain, forwarding them to appropriate domain nodes). Figure A.6.4 illustrates this by an ECA engine and an action engine (CCS).

Example 43 *Consider again the sample rule given in Section A.4.5.5. The rule is submitted to the ECA engine, which registers the event pattern via the GRH at the snoopy composite event detection engine. When the event pattern is detected for some flight, snoopy informs the ECA engine. The ECA engine submits the query with flight and date to the GRH which forwards it to a domain broker that supports the travel domain and receives a set of extended answer tuples (flight, date, booking, name). The ECA engine then submits the action component with the obtained variable bindings via the GRH to the CCS engine. The CCS engine starts two concurrent threads. The first of them just submits for each tuple of variable bindings (i.e., for each person) the atomic action <travel:reserve-room .../> to a domain broker that supports the travel domain (which will in turn invoke the reservation at the indicated hotel). The second thread first (again via the GRH and a travel domain broker) evaluates the test whether the given booking is a business class booking – all tuples where this is not the case are removed. Then, it extends the variable bindings by phone. Finally, the action <comm:send-sms .../> together with the (remaining) tuples is sent to the GHR which forwards it to a domain broker that supports the comm domain, which will then forward it to an appropriate service node that sends the message.*

Specialized Request Handling

There are also many situations that require only specialized handling of requests where a Generic Request Handler is not necessary (nevertheless, it could applied). For instance, the handling of the communication between CEDs and AEMs requires can be integrated within the CEDs:

- registration/deregistration: for each atomic event, determine its namespace and formalism,
- ask the LSR for an appropriate service and where to send the event pattern, and if it has to be wrapped into a special XML element.
- send the pattern.

A.6.5.5 Using the LSR

For "isolated" tasks, e.g. evaluating a query or executing a composite action, a suitable service can be contacted without considering past communication. On the other hand, if communication refers to a previous activity (e.g., deregistering something that has been registered before), this must obviously been done by the same service (recall that the same formalism can be implemented by several services). For that, a client (e.g., the ECA engine when registering event patterns) must do bookeeping where something has been registered, and contact this service later again.

In many cases (also others), a client can take advantage from *caching* information about a "friend" service. Then, it will not have to ask the LSR each time.

Robustness against changing Service Descriptions. When changing the Service Description of some service, its owner must update the data about its service in the LSR (which will be a service of its own in a later version). When a client's request that used communication bookkeeping fails, the client should ask the LSR if the service description of that service changed.

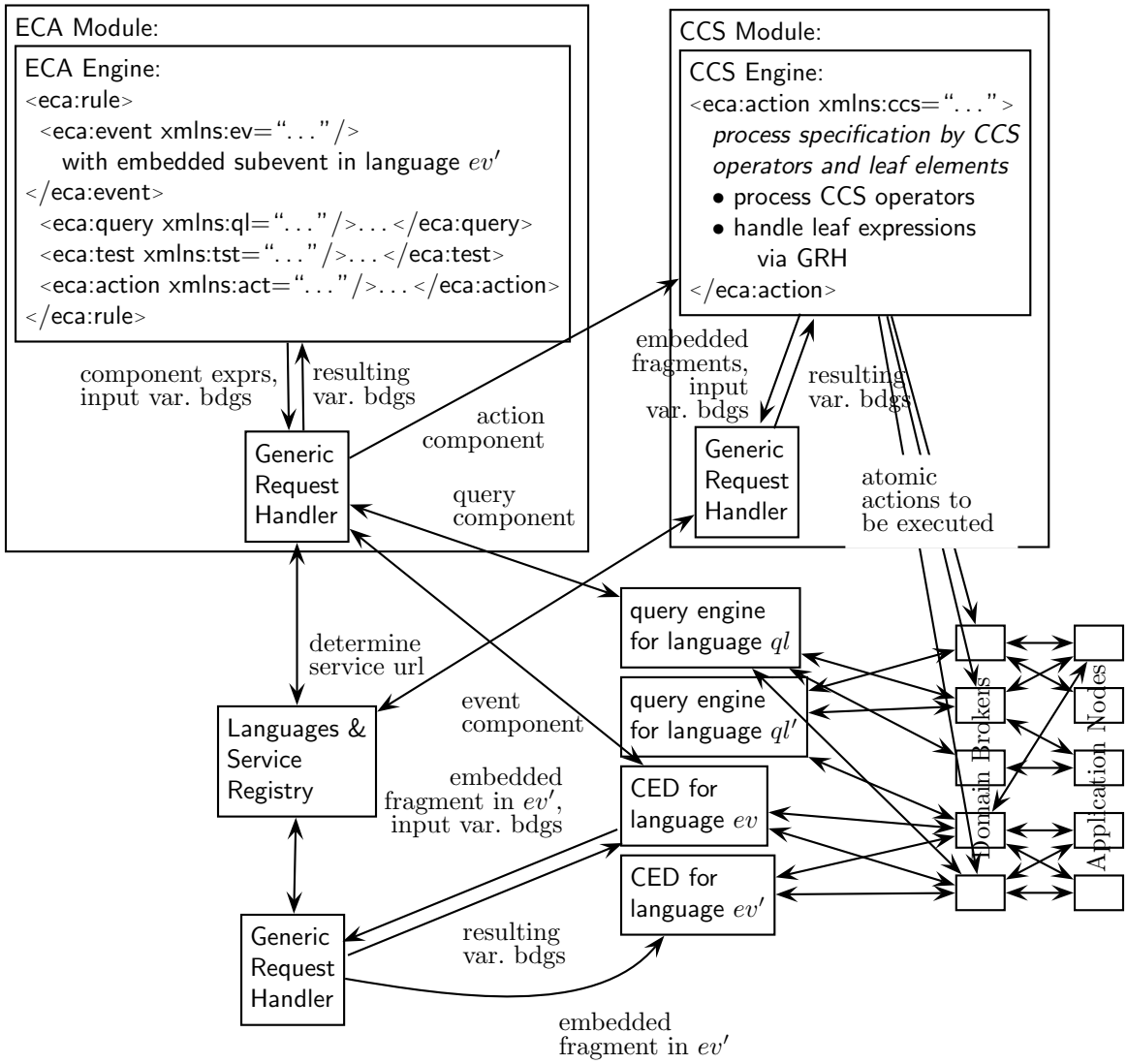


Figure A.6.4: Global Coooperation using GRHs

A.6.6 Issue: Redundancy and Duplicates in Communication

When a larger number of nodes (application nodes, domain brokers, language nodes) is considered, there is a problem of redundancy and duplicates: if an application node sends events to multiple event brokers, and each of them forwards the event to the AEMs that registered for the event type, there will be multiple detections of the event.

The duplication can be handled either by bookkeeping, or by restricting the communication. Since bookkeeping would require an agreement between multiple services that also influences the internal processing of events, a restricted communication strategy is recommended, as sketched below:

For an application node that supports domains $d_1, \ldots, d_n$:

- for each d_i, it registers at all domain brokers for d_i that it considers relevant, trustable etc.
- for each domain d_i, it sends its events only to one domain broker, and executes actions only from that domain broker (i.e., registers its actions only at that domain broker).

- for each domain d_i, it registers for the static notions that it supports at all of the domain brokers for d_i. Queries for d_i are thus answered by all domain brokers.

For an AEM:

- for an event specification that uses an event type t in domain d, it registers for t at all domain brokers for d that it considers relevant, trustable etc.

For an action engine:

- for an action in domain d to be executed, it sends the action to all domain brokers for d that it considers relevant, trustable etc.

For query engines:

- not yet relevant since we only have opaque queries against certain sources, but neither a "global" query language nor an integrating query answering engine.

So far, the communication prevents duplication of events or actions:

- integration of events: takes place at the AEMs and CEDs,
- integration of data (queries) takes place at the Domain Brokers.

The remaining problem are the ACA rules.
For domains:

- every domain (owner) distinguishes a primary domain broker that submits the ACA rules to *one* ECA engine (which will collect relevant events and action executions via the AEMs).

Chapter (Appendix A: ECA Framework) A.7

Implementation and Prototype

This chapter describes the current state of the prototype implementation at Göttingen University. As pointed out in the previous section about global architectures, the implementation of the framework consists of separate modules which can also exist in different instances, even in different implementations, sharing common interfaces as defined above. The interfaces can be refined during the development with the reference prototype.
The current architecture consists of modules of the following kinds:

- (language-independent) ECA engines,
- (language-independent) atomic event matching engines (AEM),
- composite event detection engines (CED),
- query language engines (QLE), optionally directly coupled with a database,
- action language engines (ALE),
- Generic Request Handlers (GRH; optionally integrated as instances into the ECA or ALE engines),
- domain information brokers/services (DB),
- language and service registries ("Yellow Pages") (LSR),

We will usually use the term "Service" when looking from the outside (i.e., interfaces and URLs), and "Engine" when looking at the inside, i.e., algorithms and languages.
The prototype is implemented incrementally. It will contain at least one instance of every language type.

A.7.1 Simple Setting for an ECA Prototype

For running a simple ECA prototype, the components have to be as simple as possible. The first step is to simulate them at all by "faking" appropriate <logvars:answers> messages, and in the next step simple component expressions are used:

- Events: atomic events are specified in the domain ontologies. They have a name, and properties. Atomic events have been described in Section A.4.2.1.
- Queries: atomic facts are different from atomic events. Considering the instance level of RDF and OWL, they are (i) class atoms and (ii) property triples. in XML, atomic facts are mixed up in the tree structure, and also the query languages are not based on atomic expressions.

The simplest case here is to use existing services, i.e., to query existing databases. This is e.g. done by XQuery. Here, *opaque expressions* are used. The use of opaque query expressions is investigated in Section A.7.1.2.

- Tests: atomic tests are mainly predicates that are independent from the application domain, like "=", "<", ">" etc. Domain-dependent predicates have to be dealt with in the query part. Nevertheless, the simplest case of a test is – no test at all.
- Actions: atomic actions are specified in the domain ontologies. They have a name, and are invoked with arguments. Similar to the query component, also *opaque* actions can be considered which are also easily available by existing Web Services. Simple actions also include raising events (and thus introduce a feedback cycle) and sending messages (smtp).

The most simple, standalone ECA prototype works as follows:

- fake responses from event detection
- use opaque queries
- write the actions to be executed into a log.

A.7.1.1 ECA Module Implementation

The prototype is implemented incrementally. The first module was an ECA Engine prototype whose first version has been developed by simulating the other services [59] (first presentation in late 2005).

The ECA engine implements the functionality described in Sections A.3.1-A.3.4: registration of rules, breaking them into parts, registering the event part at an appropriate service, receiving answers (variable bindings), invoking query services, evaluating conditions and invoking action services. It is responsible for dealing with constructs on ECA-ML, i.e., the eca namespace.

Architecture

The core *ECA engine* implements *only* the handling of variable bindings and the ECA rule semantics. It is supported by an instance of a *Generic Request Handler* (see Section A.6.5.4 for communicating with autonomous component services. The GRH communicates the components to suitable services. It also receives the answers and wraps them into the agreed format of variable bindings (in case of non-framework-aware services). The architecture is as shown in Figure A.6.3.

ECA: Functionality

- ECA semantics (handling variable bindings, joins),
- control flow in ECA rules.

ECA: Interface

- in from above: registration of rules,
- out to below: requests to GRH: components in XML Markup of the individual languages,
- in from below: answers from GRH, in <variable-bindings> format.

Lessons Learnt. During the development of the first version, separately developed in a BSc thesis [59], the GRH was "invented" and closely coupled with the ECA engine. It did not only communicate requests and answers, but also had access to the rules where it executed the methods for their logical semantics. This turned out not to be appropriate since the GRH functionality must also be used by the (later) CCS engine. A redesign took place.

A.7.1.2 Handling Opaque Queries

The conceptually simplest step (making a first ECA prototype run) is to investigate opaque queries. Since many repositories in the Web are non-semantic, the handling of opaque queries in XPath or XQuery is also required for the full framework.

A.7.1.2.1 Communicating with Primitive Services

When using opaque components, the most basic assumption is that the service is not aware of anything in our framework (i.e., neither that the language itself is marked up, nor that it receives any variable bindings according to the format described in Section A.3.1.2).
In these cases, the opaque fragment is submitted as a string; variables must be included in the opaque code:

- XPath and HTTP-based queries: variable occurrences in the query string are string-replaced by the actual value (only possible for literals).
- XQuery queries: variables are bound by let-statements before submitting the query to the service (possible for literals and XML fragments).

For this, it must be possible to see or derive from the eca:opaque element which variables are used.

- general case: additionally to its text contents (which is the event/query/action statement), the eca:opaque element can contain elements of the form <input-variable name="*eca-var-name*" use="*local-var-name*"/> that indicate that the value of *eca-var-name* should be replaced for the string *local-var-name* in the query (e.g., for embedding JDBC where variables are only named ?1, ?2 etc.)
- short forms: when the variable names in the opaque part coincide with those on the ECA level, <eca:opaque input-variables="var_1 ... var_n"> or multiple subelements of the form <input-variable name="var_i"/> can be used.
- XPath: if the string "$*var-name*" occurs in the query, it refers to the same variable in the ECA rule. It must be replaced.
- XQuery: if the string "$*var-name*" occurs in the query, it refers to the same variable in the ECA rule. It must be replaced, or a let-statement must be written in front.

In these simple cases, the query does not bind additional variables but the results are just answer sets. They are then only bound to variables on the rule level.

This mechanism can deal with most simple HTTP GET Web Services. Applications are here mainly application services, but also generic (query) language services can be used.

A.7.1.2.2 Framework-Aware Wrappers

Non-Framework-aware services can be wrapped into framework-aware ones. With the latter, communication can take place as described in Section A.3.2. The wrapper performs the tasks as described just above. For each tuple of distinct variable bindings (after restricting to the variables actually used in the component!), the query (or action) must be evaluated and handled separately. This makes it reasonable to provide a generic wrapper functionality that fills the gap between the ECA functionality and primitive services:

- the ECA service submits opaque fragments to a wrapper using the above format for downward communication.
- the wrapper then appropriately calls the primitive service as described above,
- and collects the answers and gives them back using the above format for upward communication. Note that opaque fragments can bind variables, or they only return functional services.

Generic Opaque XQuery Wrapper. As already mentioned on page 32, framework-aware wrappers for non-framework-aware language services (e.g., XQuery) are both useful for developing a prototype and for integrating frequently used Web Services.

As a specific format we propose to support XQuery (e.g. for opaque query components against documents on the Web) by a standalone wrapper service that accepts the format given in Section A.3.2.2 and employs a plain service. It iterates over the input variable bindings and evaluates queries by adding let *var* := *xml-fragment* statements in front of the query. It collects and joins the results and returns them in the upward communication format discussed in Section A.3.2.3.

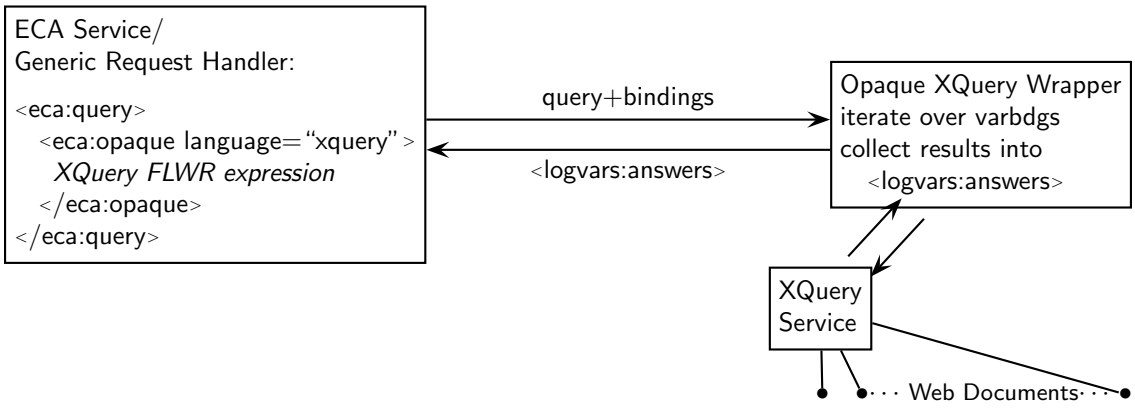


Figure A.7.1: Architecture: Generic Opaque Prototype Pattern (e.g., XQuery)

Generic Action Services. In the same way as there are domain query services and generic services that e.g. evaluate XQuery queries that contain a document("...") expression, there can be services that execute explicit updates. Since there is not yet an update language for the Semantic Web, neither for XML, we give only a sketch of such a fragment:

```
<eca:rule>
    :
  <eca:action>
    <eca:atomic>
      <eca:opaque language="xquery+updates">
        <eca:input-variables names="flight reason today"/>
        update
        for $f in document("http:www.lufthansa.de/schedule.xml")//flight
        where $f/@nr=$flight and $f/@date=$today
        insert <canceled> {$reason} </canceled> into $f
      </eca:opaque>
    </eca:atomic>
  </eca:action>
</eca:rule>
```

A.7.1.2.3 Raising Events by Opaque Atomic Actions

As described in Section A.2.3.6 for the language binding for opaque components, there the language binding is done via an explicit service URL that is contacted via an HTTP GET method. This mechanism can be "misused" especially in the prototype for raising events in the action component: the URL is the url of the service where the event should be "visible" (which is in general a domain broker).

```
<eca:action>
  <eca:opaque method="post" uri="http://www.cs.uni-goe.de/munopag/incomingevents">
    <uni:register subject="Databases" name="John Doe"/>
  </eca:opaque>
</eca:action>
```

"sends" the event representation <uni:register subject="Databases" name="John Doe"/> to the exam administration system of the CS department.

A.7.1.2.4 Web Service Calls via HTTP/SOAP

In the same way as queries via HTTP GET are supported, also actions via HTTP or SOAP Web Service Calls are supported by an appropriate wrapper.

A.7.1.2.5 Opaque: Matching Regular Expressions

Many applications require to process non-XML contents, often text (e.g., from RSS feeds). For this, the "event" contains the RSS body, and the query part must extract variable bindings from this.

We propose to use regular expressions for extracting such data by regular expressions. The uses the same match predicate as defined as a built-in predicate in [38] for the Florid system in F-Logic. The proposed markup is as follows:

```
<eca:query>
  <eca:opaque language="regexp">
    match("textslvariable", "textslregexpr" "textslvariable+")
  </eca:opaque>
</eca:query>
```

A.7.1.2.6 Summary

The above kinds of services have been implemented together with the ECA engine as a first, simple version of a prototype. A demo+testing version is available since early 2006 at `http://www.semwebtech.org/eca/frontend`. Since it is an experimental environment, it is sometimes down due to restructurings.

A.7.2 Extending the Prototype with Component Services

In the second step, component engines (CED+AEMs and ALE) have been implemented that communicate with the ECA engine: they are written wrt. the Göttingen ECA engine and are also expected to participate in cross-communication with the Lisbon one.

A.7.2.1 Composite Event Detection and Atomic Event Matchers

Services for composite event detection according to the general interfaces are developed separately.

A first integrated CED+AEM: Snoopy. A CED for a language closely related to the SNOOP event algebra [17] of the "Sentinel" active database system has been implemented in [61]. The CED uses the XML markup presented in Section A.4.2.6.

As a separate, standalone thesis, the original version incorporated composite event detection in the SNOOP language directly with an AEM for the XML-QL Style matching formalism described in Section A.4.2.2.1. The task of the thesis was specified to do this completely in XML/XSL (which is not the most efficient solution, but provides a better demonstration of the algorithm):

- an XSL stylesheet that maps a SNOOP expression into an automaton, represented in XML,
- an XSL stylesheet that, given an atomic event, transforms the XML representation of an automaton situation into the representation of the subsequent state and outputs the detected composite events.

The resulting code has been split according to the performed tasks into a CED and an AEM module as follows:

Separate XML-QL-Matching style AEM. The matching part of the stylesheet has been separated and wrapped into an XML-QL-Matching style AEM. The module is available since early 2006 in the prototype and allows to replace the "faked" answers to event detection by actually sending atomic events.

Separate Snoopy CED. The remaining part that implements the SNOOP language is installed as a separate service.

An XPath-based AEM. Another AEM using the XPath-navigation based formalism described in Section A.4.2.2.2 has also been implemented as an XSL stylesheet. It is currently being wrapped.

A.7.2.2 Queries and Updates

Here, several alternatives already exist that only need to be wrapped according to the general interface. Pure language implementations for stating queries on independent XML instances have to be distinguished from XML databases. In most cases, *opaque* query components will be used since there are not yet query languages using XML-markup.

Saxon. Saxon [32] is an XQuery implementation that allows to state queries against XML sources on the Web. A framework-awara wrapper around saxon makes it immediately integrable. Saxon does not deal with updates.

Commercial XML-enabled and XML database systems. With Oracle, IBM DB2, MS SQL Server etc., most classical relational systems habe been XML-enabled in the last years. For queries, the SQLX standard [22] is supported that embeds XPath into SQL. Updates are implemented via transformations; thus triggers on the XML level as described in Section A.2.2.1 are currently not supported.

eXist. eXist [23] is an open-source XML database that runs as a Web service. For queries, XQuery is supported; also XSLT is supported for transformations. Updates are possible via XUpdate and the XQuery+Update extension in the style of [62, 37]. A framework-aware wrapper around an eXist database that supports queries and XUpdate has been implemented. An eXist database is used currently as a substitute stub for queries against domain nodes. Several domain node architectures (Oracle, Jena) and a generic Domain Broker are currently under implementation.

Jena. Jena [31] is an open-source framework that provides functionality as an RDF query language; also OWL reasoners can be integrated. A Jena-based node has been developed in [64], using Pellet [53] as attached OWL reasoner. It supports queries (SPARQL), atomic updates, and database triggers on the RDF/OWL model:

- for the event part see Section A.2.2.2.
- query part: an SPARQL query that may use the variables bound in the event part (which is only one tuple in this case) and binds additional variables that occur free in the query;

- test part: included with query part (conjunctive query);
- action part:
  - update of the local RDF database (opaque) of the form

    <eca:opaque> *rdf update expression*</eca:opaque>
  - raising an (internal) event (analogously to RAISE_EVENT for the Oracle node described below):

    <eca:raise-event> *atomic event in markup* </eca:raise-event>

    The raised event is always sent to the "hull" of the node.

  Note that the bodies can contain free variables (that in the opaque case need to be declared as input-variable in the markup according to Section A.7.1.2). The action must be carried out for *each* tuple of variable bindings (that can be several after evaluating the query).

SQL Database Node. An SQL-based node (sample scenario: car rental company) is under implementation in [26]. It is based on Oracle, especially its Rule Engine (available since version 10g).

A.7.2.3 Composite Actions and Processes

A CCS-based action engine implementing the behavior described in Section A.4.5 is under implementation in [60]. It will share the GRH functionality with the ECA engine. The GRH and the ECA engine are just being adapted to this requirement.

A.7.3 Application Domains

Domain Nodes. Application nodes consist mainly of local databases and application functionality, as much as possibly also defined by rules. Domain nodes based on SQL, XML, and RDF are under implementation. An eXist database is used currently as a substitute stub for queries against domain nodes. The nodes under development will provide local active functionality (e.g., by internal database triggers) and support for events, queries and actions according to Section A.5.3.

An SQL-based node (sample scenario: car rental company) is under implementation in [26]. It is based on Oracle, especially its Rule Engine (available since version 10g).

Domain Broker. A generic domain broker according to Section A.5.6 is under implementation in [35].

A.7.4 Infrastructure

The first node uses hard-coded URIs and a fixed communication format/wrapping. Its development has lead to a clear understanding and isolation in a separate class of the GRH functionality. The LSR is currently simply implemented as an XML/RDF file (see Section A.6.5.3) which can be found at `http://www.semwebtech.org/2006/lsr/lsr.rdf`. The current refinement and adaptation to the Lisbon Services will lead to a full specification and trading functionality of the LSR and GRH modules.

A.7.5 Using the Prototype Demonstrator

The demo frontend at `http://www.semwebtech.org/eca/frontend` provides a choice of predefined actions:

- register and deregister predefined rules. Some rules serve for illustrating the ECA functionality (atomic events, simple opaque queries, faked actions), others serve for testing composite events,
- send (predefined) faked event detection answers to the ECA engine,
- send (predefined) atomic events to the AEMs: they will match them and inform the CED,
- Using the form, users can also write own rules and register them,
- Using the form, users can also write own events and send them to the AEMs.
- Users can also program an own service and feed the AEMs with atomic events via HTTP following the communication specification given in the LSR.

A.7.6 Anticipated "Foreign" Modules

A.7.6.1 Wrapped Composite Event Detection Engines

XChange. XChange is currently closely connected with the Xcerpt system. Here, a wrapper that returns variable bindings instead of immediately submitting them to Xcerpt is required. A first version can be a simple Xcerpt rule that sends a message that contains simply the XML serialization of the variable bindings. For this XChange/Xcerpt must be wrapped in a Web Service.

RuleCore. The RuleCore event detection module returns variable bindings. Wrapping it as a Web Service makes it immediately integrable.

A.7.6.2 Wrapped Query Engines

Florid/LoPiX. Florid (F-Logic Reasoning in Databases) [25] is an implementation of the F-Logic knowledge management and database formalism [33]. Its migration to XML, LoPiX (Logic Programming in XML) [40], provides similar features based on a data model that extends XML. A wrapper into a Web service has been implemented recently which makes Florid/LoPiX available as a service for opaque queries.

Xcerpt. Xcerpt is the XML query language developed by REWERSE WG I4. It allows to state queries against XML sources on the Web. Wrapping it as a Web Service makes it immediately integrable. A database implementation and a lifting to RDF are planned. Xcerpt supports updates of documents via XChange's update actions. Updates are currently implemented by translating them into the definition of a view and then replacing the original document by this view. Thus, triggers on the XML level as described in Section A.2.2.1 are currently not supported.

A.7.7 Pilote Applications

The online ECA engine cannot be used for useful applications since there is no user authentification. Any registered rule in it that e.g. updates via HTTP and XUpdate an XML database will be visible to others that then can misuse the information for also updating that database. Pilote applications can install a separate ECA instance (that uses the existing component engines) with own rules.

Chapter (Appendix A: ECA Framework) A.8

Abbreviations

ECA: Event-Condition-Action

ECE: Event-Condition-Event derivation rule

ACA: Action-Condition-Action implementation rule

CQE: Continuous-Query-Event derivation rule

GRH: Generic Request Handler (ECA Engine Implementation]

CES: Composite Event Specification

CEL: Composite Event Language

CED: Composite Event Detection (Service) (for some CEL)

AES: Atomic Event Specification

AESL: Atomic Event Specification Language

AEM: Atomic Event Matcher (for some AESL)

QL: Query Language

QE: Query Language (for some QL)

CAL: Composite Action Language

CAE: Composite Action Execution Engine (for some CAL)

EB: Event Broker

DB: Domain Broker (usually contains an Event Broker)

DN: Domain Node

DSR: Domain Service Registry

LSR: Languages and Services Registry

Chapter (Appendix A: ECA Framework) A.9

DTD of ECA-ML

```
<!ELEMENT rule (%variable-decl, event, query*, test?, action+)>
<!ELEMENT event (%variable-decl, (atomic-event | opaque | any-element-of-a-cesl))>
<!ELEMENT query (%variable-decl, (opaque | any-element-of-a-ql))>
<!ELEMENT test (%variable-decl, (opaque | any-element-of-boolean-ml))>
<!ELEMENT action (%variable-decl, (opaque | any-element-of-an-action-ml))>
  <!ATTLIST action executed-at (URL | "owner") #IMPLIED>
```

Chapter (Appendix A: ECA Framework) A.10

DTD for Logical Stuff: Variable Bindings etc.

```
<!ELEMENT answers (answer*)>
<!ELEMENT answer (result?, variable-bindings?)>
<!ELEMENT result ANY>
<!ELEMENT variable-bindings (tuple+)>
<!ELEMENT tuple (variable+)>
<!ELEMENT variable ANY>
<!ATTLIST variable name CDATA #REQUIRED
          ref URI #IMPLIED>    <!-- variable has either ref or content-->
```

Appendix B

The XChange Prototype

The syntax and semantics of XChange has not been changed or extended wrt. the description given in [1]. The following description of the prototype is taken from [55].

Chapter (Appendix B: XChange) B.1

A Prototypical Runtime System

This part of the thesis discusses the proof-of-concept implementation that has been developed for the language XChange. At moment of writing, the XChange prototype does not implement all features of the language; the development of the XChange prototype is ongoing work. This part gives a description of its current status accompanied by suggestions on where and how the implementation is to be changed or extended for fully implementing XChange.

The XChange prototype has been implemented in Haskell [63], a functional programming language. Chosing Haskell has been strongly motivated by the existing Xcerpt[1] prototype implementation, which is implemented in Haskell. Recall that not only Web queries and deductive rules specified in Xcerpt need to be evaluated for executing XChange programs, but also Simulation Unification is employed for evaluating XChange atomic event queries. Thus, the prototype implementation of Xcerpt has been "extended" so as to implement the language XChange.

For space reasons, this part of the thesis does not offer a complete discussion of every aspect of the implementation; it offers a high-level guide to XChange's implementation and a general view over the structure of the source code. It also abstracts away from details on the Xcerpt implementation; more information on Xcerpt's prototype implementation can be found in [58], pages 207-225.

B.1.1 Overview. Source Code Structure

For executing XChange programs, an implementation of the language XChange needs to provide, besides a parser for the language, components for evaluating the 'event part' and the 'condition part', and executing the 'action part' of XChange rules. These components need to communicate through substitution sets for the variables with at least one defining occurrence in the rule parts. The implementation needs to convey the semantics of the three rule parts, which has been presented in [1].

Mirorring the three parts of an XChange reactive rule, the main module of the XChange prototype implements an *event handler*, a *condition handler*, and an *action handler*. They are defined as functions and run separately. Communication between the functions implementing the event, condition, and action handlers is realised through *channels*, an extension to Haskell found in Concurrent Haskell [54]. Channels provide a buffered First In, First Out message-pasing communication between the component handlers. The flow of messages between the handlers mirrors the flow of substitution sets between the parts of an XChange reactive rule.

The event handler has the abilities to receive atomic events, evaluate the event queries registered in the system, and release events whose lifespan has expired. Detected answers to the these event queries are communicated to the condition handler through a *condition channel*. Upon successful evaluation of an event query eq, the condition handler evaluates the Web query q of the

[1] Xcerpt Project, `http://www.xcerpt.org`

rule having *eq* as event query (for determining which parts form a registered reactive rule, each rule gets an identifier at registration); evaluation of Web queries is based on Xcerpt's abilities. The successful evaluation of the Web query *q* is signalled to the action handler by writing the substitution set obtained from the evaluation of *eq* and *q* to the *action channel*. The action handler executes the action of the rule having 'event part' *eq* and 'condition part' *q*. It executes local updates by transforming update terms into Xcerpt goals and evaluate these goals. At moment, the prototype implementation of XChange does not offer support for executing transactions on the Web.

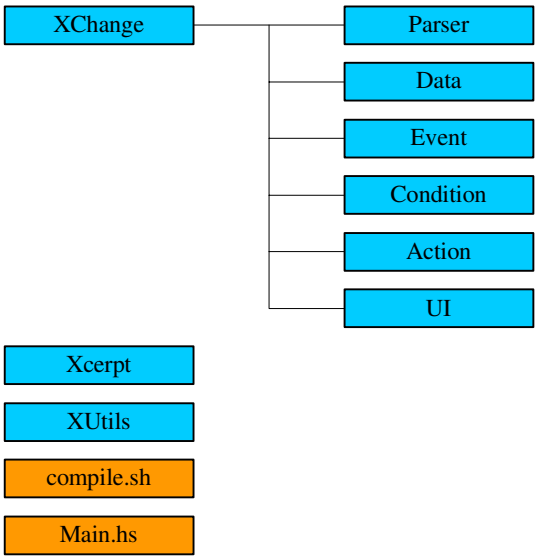


Figure B.1.1: Overall module and file structure

The structure of the source code is depicted in Figure B.1.1. By using the hierarchical module mechanism of Haskell, the code is structured in different modules: Module `XChange` implements the language XChange by using module `Xcerpt`, which implements the language Xcerpt. Module `XUtils` implements some date, list, and string utilities (i.e. functions needed in module `XChange`). Files `compile.sh` and `Main.hs` are used for compiling XChange (see Section B.1.7). Module XChange is made of the following submodules:

XChange.Parser provides lexer and parser modules for parsing XChange programs into the data structures defined in `XChange.Data`.

XChange.Data provides data structures (e.g. data structures into which programs are parsed and channel data structures) and functions on these structures.

XChange.Event provides functions for receiving events, evaluating event queries, and deletion of events.

XChange.Condition provides functions for evaluating Web queries and deductive rules.

XChange.Action provides functions for executing actions (e.g. for transforming update terms into Xcerpt goals and evaluate these goals).

XChange.UI provides functions for the command line and for debugging purposes.

The next sections discuss in more detail the functions and data structures defined in the modules `XChange.Parser`, `XChange.Data`, `XChange.Event`, `XChange.Condition`, and `XChange.Action`. The module `Xcerpt` is described in [58].

B.1.2 XChange Parser

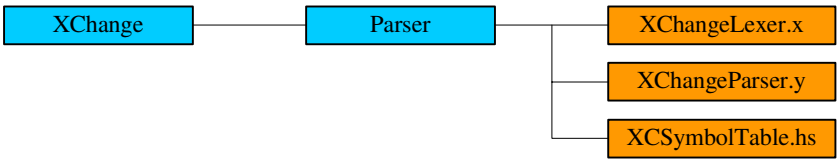


Figure B.1.2: Module and file structure of XChange.Parser module

Given an XChange program to be executed, two steps are realised for transforming the program into the data structures used by the event, condition, and action handlers: A lexical analysis is performed for transforming the characters of the program into a sequence of tokens. Based on the language grammar, the sequence of tokens is transformed into XChange data structures (defined in module `XChange.Data`).

The module and file structure of the `XChange.Parser` module is presented in Figure B.1.2. The XChange parser has been implemented by extending the parser for Xcerpt programs. The lexical analysis of XChange programs is done by an XChange lexer built by using the lexer generator Alex [20]. The parser of XChange programs has been built by using the parser generator Happy [39].

Module `XChange.Parser.XChangeLexer` implements the XChange lexer; it defines tokens in terms of regular expressions. The function `lexer` in `XChangeLexer.x` takes a string (an XChange program) and return a list of tokens ([`Token`]). Module `XChange.Parser.XCSymbolTable` defines the function `resolveSymbols` that looks up in a symbol table for correctly analysing XChange programs. The symbol table contains tokens for the language keywords; it should be modified when XChange constructs are modified or extended.

Module `XChange.Parser.XChangeParser` defines the function `parseXCProgram` for parsing the list of tokens resulted from the lexical analysis of a program. The parser generator Happy uses grammar rules in a syntax similar to Backus-Naur Form (BNF); rules are extended for defining the action to be taken when "encountering" given specifications. For example, the next given code specifies an excerpt of the grammar rule defining XChange event query specifications. An event query specification is either a term specification, or a keyword (here `or`, `and`, `andthen`) followed by single or double opening braces, a list of event query specifications, and corresponding closing braces. An event query (instance of type `EvQuery`) is to be returned.

```
PEvQuery :: { EvQuery }
PEvQuery : PTerm                              { EvQTerm $1 }
       | or  '{' PEvQueryL '}'                { EvQOr $3 }
       | and '{' PEvQueryL '}'                { EvQAnd $3 }
       | andthen '[' PEvQueryL ']'            { EvQAndthen $3 False }
       | andthen '[' '[' PEvQueryL ']' ']'    { EvQAndthen $4 True }
       ...
```

Clearly, `XChangeParser.y` contains the grammar rules for XChange event queries, Web queries, actions, and rules. Such grammar rules are used for building the data structures introduced in

the next section. The function for performing the lexical analysis and building the XChange data structures is also defined in `XChangeParser.y`:

```
parseXChange = parseXCProgram . resolveSymbols . lexer
```

B.1.3 XChange Data Structures

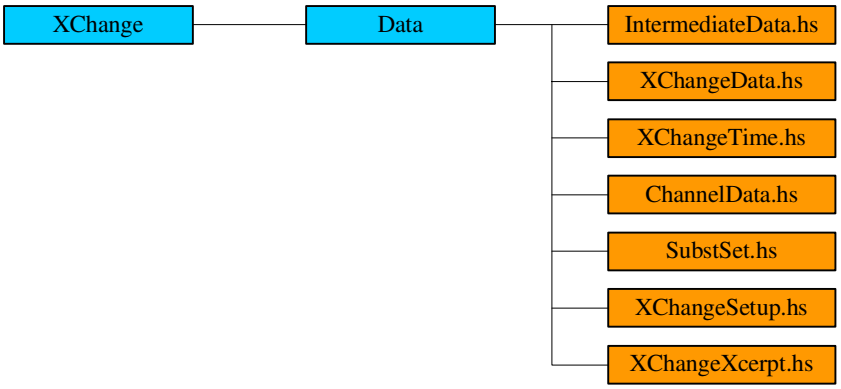


Figure B.1.3: Module and file structure of XChange.Data module

The module `XChange.Data` defines the data structures on which the event, condition, and action handlers work; it also provides functions over these data structures. The module and file structure of the `XChange.Data` module is given in Figure B.1.3; the module is made of the following submodules:

XChange.Data.IntermediateData defines the data structures (Haskell data types) whose "instances" are built when parsing XChange programs. Recall the previous example giving an excerpt of the parser's code; it returns an instance of the type `EvQuery`. The following code gives an excerpt of the definition for the data type `EvQuery` (corresponding to an XChange event query):

```
data EvQuery = EvQTerm { eterm :: Term }
             | EvQOr  { evq_queries :: [EvQuery] }
             | EvQAnd { evq_queries :: [EvQuery] }
             | EvQAndthen { evq_queries :: [EvQuery], evq_partial :: Bool }
             ...
             deriving (Eq,Show)
```

For example, the data type for a temporally ordered conjunction event query is given as a constructor `EvQAndthen` followed by a list of elements of type `EvQuery` and a boolean value expressing whether the event query specification is total or partial. The data type corresponding to an XChange program is defined as:

```
data XCProgram = XCProg [XCRule] deriving Show
```

That is, the XChange parser "transforms" an input XChange program into a list of rules (elements of type `XCRule`) having as constructor `XCProg`.

XChange.Data.XChangeData defines a slight modification of the data types returned by the XChange parser; these types are further used by the event, condition, and action handlers. For example, a rule identifier type is defined here that is to be associated to each rule registered in the system. Also, the time specifications (time points, time intervals, durations) returned as strings by the parser are transformed into XChange time types (e.g. type `XChangeDuration` for a length of time).

XChange.Data.XChangeTime defines the XChange time types (`XChangeTime`, `XChangeDuration`) that are needed for evaluating event queries correctly. The module provides also functions for transforming strings in XChange time types and the relations between time points and durations, respectively (e.g. equality relation, comparison of time points and durations, respectively). For example, the data type `XChangeDuration` is defined as a time difference or an integer having as constructors `XChangeTimeDiff` and `XChangeIntDuration`, respectively:

```
data XChangeDuration
    = XChangeTimeDiff TimeDiff |
      XChangeIntDuration Int
```

and the equality relation on durations:

```
instance Eq XChangeDuration where
    (XChangeTimeDiff td1) == (XChangeTimeDiff td2)
      = (td1 == td2)
    (XChangeIntDuration i1) == (XChangeIntDuration i2)
      = (i1 == i2)
```

The XChange time types are used also for the parameters of XChange event messages, i.e. raising time and reception time.

XChange.Data.ChannelData defines data structures `AtomicEvent` (as a term with a reception time) and `CompositeEvent` (as a list of `AtomicEvent` instances, a beginning and an ending time, and a constraint). Constraints represent the possible variable substitutions; they are used in defining the data structure `Firing`, instances of which are communicated through the channels expressing successful evaluations of parts of an XChange rule (identified by `XCRuleId`). Functions on these data structures are also provided (e.g. show functions for firings output).

XChange.Data.XChangeXcerpt, XChange.Data.SubstSet provides data structures and functions from Xcerpt that are needed for evaluating event queries and Web queries.

XChange.Data.XChangeSetup defines configuration data used for executing XChange programs. The event, condition, and action channels are defined in this module; also functions for the output of e.g. intermediate states or firings, and other useful data (e.g. port number for server) are declared.

B.1.4 XChange Event Handler

The XChange event handler receives event messages, evaluates the event queries registered in the system, and deletes events whose lifespan has expired. The module and file structure of the module `XChange.Event` is given in Figure B.1.4. The module `XChange.Event` consists of the following submodules:

XChange.Event.EventReceiver defines the function `eventReceiver` for receiving event messages from TCP/IP connections using the default port 4711. The received event messages are augmented with reception time and identifier before writing them in the event channel for the event handler:

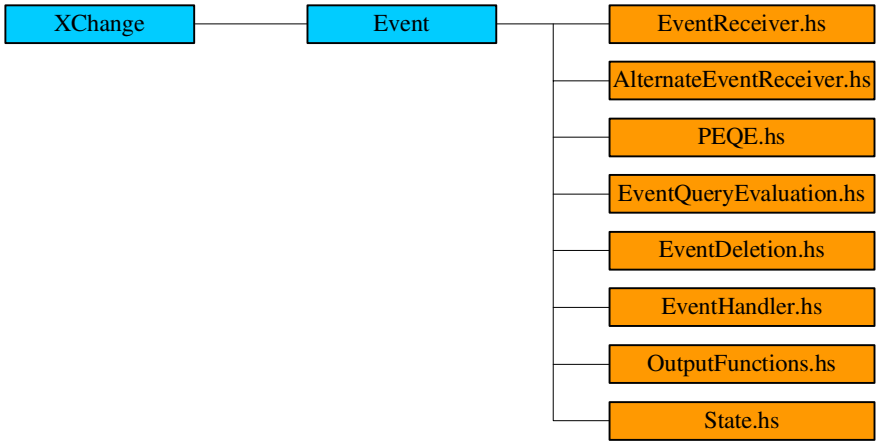


Figure B.1.4: Module and file structure of XChange.Event module

```
let ae = (AtomicEvent term (XChangeClockTime rcptTime))
writeChan channel ae
```

The prototype uses TCP/IP socket communication; receiving and sending event messages over HTTP is planned.

XChange.Event.AlternateEventReceiver is used for evaluating event queries against the event messages contained in given files; it is useful for debugging purposes.

XChange.Event.PEQE implements the operator trees for XChange event queries; a discussion on the defined data structures and functions can be found in [21], Section 8.3.

XChange.Event.EventQueryEvaluation defines the function `evaluateQuery` that performs the event query evaluation:

```
evaluateQuery :: AtomicEvent -> PartialEventQueryEval ->
                 (PartialEventQueryEval, [CompositeEvent])
```

The function takes an atomic event and a partial evaluated event query, it returns an updated partial evaluation and a list of composite events. A more detailed discussion on `evaluateQuery` is given in [21], Section 8.3.

XChange.Event.EventDeletion provides functions for releasing events after their lifespan has expired.

XChange.Event.EvenHandler defines the main loop of the event handler, a tail-recursive function

```
eventHandlerLoop :: XChangeSetup -> State -> IO()
```

where the state of the event handler is a list of partial evaluations (`PartialEventQueryEval`) for the event queries of the rules registered in the system (identified by `XCRuleId`):

```
newtype State = State [(PartialEventQueryEval, XCRuleId)]
```

XChange.Event.OutputFunctions provides functions for the output of received events, firings, and registered rules.

XChange.Event.State provides the definition of type `State` given above.

B.1.5 XChange Condition Handler

The XChange condition handler evaluates Web queries and deductive rules when a new firing is signalled from the event handler. The evaluation of Web queries and deductive rules is based on the prototype implementation of Xcerpt.

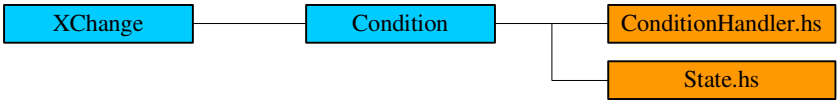


Figure B.1.5: Module and file structure of XChange.Condition module

The main loop of the condition handler (`conditionHandlerLoop`) is implemented as a tail-recursive function so as to get an infinite loop in Haskell. The type of the function is

```
conditionHandlerLoop :: XChangeSetup -> State -> Program -> IO()
```

where the state of the condition handler is a list of Web queries (`Query`) registered in the system and associated with the corresponding rule identifiers (`XCRuleId`):

```
newtype State = State [(Query, XCRuleId)]
```

and the program (of type `Program`) contains the deductive rules of the XChange program to be executed. The result type of the function is the `IO()`-monad.

The module and file structure of the module `XChange.Condition` implementing the XChange condition handler is depicted in Figure B.1.5. The submodules of `XChange.Condition` are:

XChange.Condition.ConditionHandler defines the function presented above that implements the condition handler. When a new firing is written in the condition channel, the condition handler looks for the Web query associated with the rule identifier (recall that a firing is made of a constraint and a rule identifier) in its state. The Web query is evaluated against the specified Web resources (if a resource specification is given) or against the set of deductive rules (`xcerptRules`) contained in the XChange program executed.

The evaluation returns either a constraint `False` (expressing unsuccessful evaluation) or a constraint expressing possible bindings for the variables; in the latter case, a new firing is written to the action channel representing the modified constraints (obtained from the event handler and condition handler) associated with the rule identifier.

XChange.Condition.State provides the definition of type `State` given above.

B.1.6 XChange Action Handler

The XChange action handler executes the actions specified in the 'action part' of XChange reactive rules. Similar to the functions implementing the event and condition handlers, the main loop of the action handler (`actionHandlerLoop`) is a tail-recursive function:

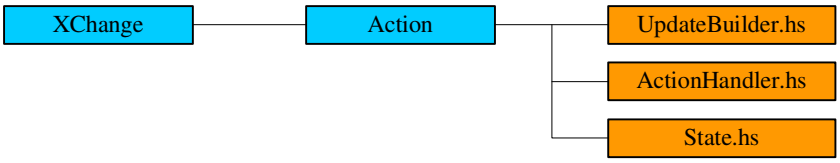


Figure B.1.6: Module and file structure of XChange.Action module

```
actionHandlerLoop :: XChangeSetup -> State -> IO()
```

where the state of the action handler is a list of tuples of actions registered in the system (`XCAction`) and corresponding rule identifiers (`XCRuleId`):

```
newtype State = State [(XCAction, XCRuleId)]
```

Again, the result type of the function is the `IO()`-monad.

The module and file structure of the `XChange.Action` module is given in Figure B.1.6. The submodules of `XChange.Action` are:

XChange.Action.UpdateBuilder defines the function `createUpdateGoal` that takes an update term, a list of resources to be modified and returns an Xcerpt goal that is used to construct the data after the update:

```
createUpdateGoal :: Term -> [Resource] -> Rule
```

The function implements the needed rewriting rules for transforming XChange update terms into Xcerpt goals; the rules are given in Appendix B.2.

XChange.Action.ActionHandler defines the `actionHandlerLoop` function discussed above. Upon reception of a new firing consisting of a constraint and a rule identifier from the condition handler, the action handler looks for the action to be executed (using its state and the rule identifier). The function `executeAction` executes the retrieved action; at moment, it implements the execution of local updates by constructing data after the update.

XChange.Action.State provides the definition of type `State` given above.

At present, the action handler executes XChange local updates as reactions to (atomic or composite) events. The execution of remote updates and raising events are to be implemented in the near future. The implementation of local updates is the most important part of the action handler; it acts as a building block for the execution of remote updates. For executing a remote update u_r specified in an XChange program P, P sends a request to the XChange processor at the Web site whose data is to be modified by u_r. Thus, the desired update u_r is to be executed locally by the XChange processor receiving the update request.

Raising and sending event messages can be easily implemented: The given event term(s) and the constraints (variable substitutions) received through the action channel are used to construct data term(s) to be sent. The construction of data terms can be realised by using the `applySubstitutions` function of Xcerpt (defined in module `Xcerpt.EngineNG.Substitution`):

```
applySubstitutions :: Term -> [Substitution] -> [Term]
```

and the function `getSubstSet` (defined in module `XChange.Data.SubstSet`) for obtaining the set of substitutions from the constraint received through the action channel:

```
getSubstSet :: Constraint -> SubstSet
```

where the type `SubstSet` is defined in module `XChange.Data.SubstSet` as

```
newtype SubstSet = SubstSet [Substitution]
```

The obtained data terms need to be augmented with the event messages' parameters `sender` and `raising- time` by using a function similar to `augmentEventTerm` defined in module `XChange.Event.Event-Receiver`. The sending of the constructed event messages can be implemented similarly to the reception of event messages (see `eventReceiverLoop` in module `XChange.Event.EventReceiver`).

B.1.7 Building and Running XChange

The source code of the XChange prototypical implementation is available at `http://www.pms.ifi.lmu.de /mitarbeiter/patranjan/`. For building and running XChange, one needs to compile XChange with the Glasgow Haskell Compiler[2] (GHC); the version of GHC used in compiling the XChange source code is GHC 6.2.2. The shell script `compile.sh` calls GHC on the given file. In a Unix shell, one needs to do

```
> ./compile.sh Main
> mv Main xchange
```

Now, one can execute XChange programs. The command line syntax for running an XChange program is:

```
> xchange [Options] Program [Event Messages Files]
```

where

Options are the supported command line options; they are prefixed by − and provided in a short and a long form (as is common on Unix systems). The options provided by `xchange` are given in the following:

```
Short form  Long form                      Description

-r[FILE]  --receivedEventOutput[=FILE]  write output of received events to FILE
-i[FILE]  --intermedEventOutput[=FILE]  write output of intermediate state to
                                        FILE
-c[FILE]  --cleanedStateOutput[=FILE]   write output of cleaned state to FILE
-f[FILE]  --firingsOutput[=FILE]        write output of firings to FILE
-d[FILE]  --debugOutput[=FILE]          write debug output to FILE
-p[PORT]  --port[=PORT]                 set server port (default: 4711)
```

Program gives the XChange program to be executed (e.g. `./test/test5.xchange`); the current XChange prototype runs XChange programs written using the term syntax of the language. An XChange parser for an XML-based syntax of the language is to be developed.

Event Messages Files give the files from which the event messages are to be used for evaluating the given XChange program; this option is provided for debugging purposes.

The prototype implementation of XChange is not the result of the work of a single person. The implementation of Xcerpt, which is the query language integrated into XChange, is the outcome of Dr. Sebastian Schaffert's efforts with contribution of a couple of graduate students. The evaluation

[2]The Glasgow Haskell Compiler, `http://www.haskell.org/ghc/`

of XChange event queries (for atomic and composite event detection) has been developed as part of the master's thesis of Michael Eckert, work supervised by Prof. Dr. François Bry and the author. The integration of the Xcerpt and XChange implementations as well as other implementation tasks (the transformation of the rewriting rules for update terms from an "informal" description into Haskell code) have been done in collaboration with Oliver Friedmann, a student assistant in the XChange project.

Chapter (Appendix B: XChange) B.2

Updates through Construction: Rewriting Rules

This part of the thesis gives rewriting rules for transforming an XChange elementary update into a corresponding Xcerpt goal, i.e. a goal that constructs the data after the update. This represents the approach taken in XChange for executing elementary updates; the main challenges and ideas of the approach have been presented in [1].

Given an XChange elementary update u, the following code (implementing the rewriting rules) constructs a corresponding Xcerpt goal G of the form $ConstructTerm \leftarrow_g QueryTerm$. The structure of the subjacent query term and the update operations of u are taken into account. The resources of u (i.e. persistent data to be modified) are just 'forwarded' to the query and construct part of the Xcerpt goal.

The following code is found in module XChange.Action.UpdateBuilder (see Appendix B.1.1 for the module and file structure of the XChange prototype implementation and Appendix B.1.6 for the module and file structure of the action handler). *Note* that `--` precedes comments in the following function definitions.

```
module XChange.Action.UpdateBuilder (createUpdateGoal) where

-- System-related imports
import IO
import System
import Control.Concurrent.Chan

-- Import data structures and functions over lists
import XUtils.ListUtils
import XChange.Data.XChangeXcerpt
import Xcerpt.Data.Program


-- Creates an Xcerpt goal (of type Rule) from an update
-- term (type Term) and resources (type Resource)
createUpdateGoal :: Term -> [Resource] -> Rule
createUpdateGoal t r =
        let
          (term, query) = createUpdateGoal' t getBaseName
        in
          Goal {output = r,
                rhead = maybeTermToTerm term,
```

```
                rbody = termToQuery (maybeTermToTerm query) r}


-- An Xcerpt goal consists of head (type Term) and body (type Term)
type UpdGoal = (Maybe Term, Maybe Term)


-- Creates an Xcerpt goal given a term (type Term) and a
-- base name (type String) for the fresh variables
createUpdateGoal' :: Term -> String -> UpdGoal


-- The following function is applied to every child of an
-- update term and 'concatenate' the results
createUpdateGoal' e@Elem {children = oldChildren, total = t} base =
  let
     -- Variable basename for children
     childrenBase = getVarName base 0
     -- Variable name
     freshVar = getVarName base 1
     -- Position var name
     posVar = getVarName base 2
     -- See Xcerpt, Comparison.hs
     cmpRoutine = ("compareIntTerm", compareIntTerm)
     -- Generates an all optional var Fresh order by position
     -- for the goal's head
     headVar = Optional SortAll { variables = [posVar], cmp = cmpRoutine,
                                  template = [ (Var freshVar)]}
     -- Generates an position var Fresh1 optional var Fresh2
     -- for the goal's body
     bodyVar = Optional TPos {pos = (Var posVar), content =  (Var freshVar)}
     -- Apply createUpdateGoal' on every child and return a
     -- couple of maybe Term lists
     newChildren = unzip (mapIdx (\a i -> createUpdateGoal' a (sb i)) oldChildren 0)
                   where
                   sb i = getSubBaseName childrenBase i
     -- Children of the goal's head
     headChildren = concatListCond (filterMaybe (fst newChildren)) [headVar] (not t)
     -- Children of the goal's body
     bodyChildren = concatListCond (filterMaybe (snd newChildren)) [bodyVar] (not t)
  in
    -- The goal's head is total and ordered
    (Just e{ordered = True, total = True, children = headChildren},
    Just e{children = bodyChildren})


-- For transforming an insert operation: the construct
-- term occurs in the goal's head and nothing in its body
createUpdateGoal' (Insert term) _ = (Just term, Nothing)


-- For transforming a delete operation: the query term
-- occurs in the goal's body and nothing in its head
createUpdateGoal' (Delete term) _ = (Nothing, Just term)
```

```
-- For transforming a replace operation: the construct term
-- occurs in the goal's head and the query term in its body
createUpdateGoal' (Replace a b) _ = (Just b, Just a)


-- Variable need to occur in both parts (head and body) of a goal
createUpdateGoal' (Var s) _ = (Just (Var s), Just (Var s))


-- Rest remains unchanged
createUpdateGoal' t _ = (Just t, Just t)


-----------------------------------
-- Helper functions are defined next
-----------------------------------

-- Compares two integer terms
compareIntTerm :: Term -> Term -> Ordering
compareIntTerm (TInt a) (TInt b) = compare a b


-- Returns a dummy term given nothing and the associated term otherwise
maybeTermToTerm :: Maybe Term -> Term
maybeTermToTerm (Just t) = t
maybeTermToTerm Nothing = TOr [] []


-- Creates a dummy framework around term
termToQuery :: Term -> [Resource] -> Query
termToQuery t r = QTerm {resources = r, term = t}


-- Creates a variable name
-- getVarName "Test" 5 = "Test5"
getVarName :: String -> Int -> String
getVarName s i = s ++ show i


-- Creates a sub variable name
-- getSubBaseName "Test5" 3 = "Test5_3"
getSubBaseName :: String -> Int -> String
getSubBaseName s i = s ++ "_" ++ (show i)


-- Creates a base var name
getBaseName :: String
getBaseName = "Fresh_"
```

Functions defined in module `XUtils.ListUtils` are used for defining the function `createUpdateGoal`; they are given next.

```
module XUtils.ListUtils where
```

```
...

-- Maps a function on array by adding index argument
mapIdx :: (a -> Int -> b) -> [a] -> Int -> [b]
mapIdx f (x:xs) i = (f x i) : (mapIdx f xs (i + 1))
mapIdx _ [] _ = []

-- Returns all Just elements
filterMaybe :: [Maybe a] -> [a]
filterMaybe l = [x | Just x <- l]

-- Concats two lists on codition
concatListCond :: [a] -> [a] -> Bool -> [a]
concatListCond x y True = x ++ y
concatListCond x y False = x

...
```

The given Haskell code implements rewriting rules for transforming XChange elementary updates into corresponding Xcerpt goals. These rules have been implemented for proof-of-concept purposes; they cover a representative "class" of XChange update terms. Ongoing work concerns testing the implemented rules to determine to which extent all possible XChange update patterns are covered and to reveal details that have been possibly neglected.

Acknowledgements

This research has been co-funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

Bibliography

[1] José Júlio Alferes, Ricardo Amador, Erik Behrends, Mikael Berndtsson, François Bry, Gihan Dawelbait, Andreas Doms, Michael Eckert, Oliver Fritzen, Wolfgang May, Paula Lavinia Pătrânjan, Loic Royer, Franz Schenk, and Michael Schröder. Specification of a model, language and architecture for evolution and reactivity. Technical Report I5-D4, REWERSE EU FP6 NoE, 2005. Available at `http://www.rewerse.net`.

[2] José Júlio Alferes, Ricardo Amador, and Wolfgang May. A general language for evolution and reactivity in the semantic web. In *Principles and Practice of Semantic Web Reasoning (PPSWR)*, number 3703 in Lecture Notes in Computer Science, pages 101–115. Springer, 2005.

[3] José Júlio Alferes, Mikael Berndtsson, François Bry, Michael Eckert, Wolfgang May, Paula Lavinia Pătrânjan, and Michael Schröder. Use cases in evolution and reactivity. Technical Report I5-D2, REWERSE EU FP6 NoE, 2005. Available at `http://www.rewerse.net`.

[4] James Bailey, Alexandra Poulovassilis, and Peter T. Wood. An Event-Condition-Action Language for XML. In *Int. WWW Conference*, 2002.

[5] Erik Behrends, Oliver Fritzen, Wolfgang May, and Franz Schenk. Combining ECA Rules with Process Algebras for the Semantic Web. In *Rule Markup Languages (RuleML)*, number to appear. IEEE, 2006.

[6] Erik Behrends, Oliver Fritzen, Wolfgang May, and Daniel Schubert. An ECA Engine for Deploying Heterogeneous Component Languages in the Semantic Web. In *Web Reactivity (EDBT Workshop)*, number 4254 in Lecture Notes in Computer Science, pages 887–898. Springer, 2006.

[7] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 1(37):77–121, 1985.

[8] Mikael Berndtsson and Marco Seiriö. Design and Implementation of an ECA Rule Markup Language. In *Rule Markup Languages (RuleML)*, number 3791 in Lecture Notes in Computer Science, pages 98–112. Springer, 2005.

[9] Eike Best, Raymond Devillers, and Maciej Koutny. The Box Algebra = Petri Nets + Process Expressions. *Information and Computation*, 178:44–100, 2002.

[10] Harold Boley, Mike Dean, Benjamin Grosof, Michael Sintek, Bruce Spencer, Said Tabet, and Gerd Wagner. FOL RuleML: The First-Order Logic Web Language. `http://www.ruleml.org/fol/`.

[11] Angela Bonifati, Daniele Braga, Alessandro Campi, and Stefano Ceri. Active XQuery. In *Intl. Conference on Data Engineering (ICDE)*, pages 403–418, San Jose, California, 2002.

[12] Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Pushing Reactive Services to XML Repositories Using Active Rules. In *World Wide Web Conf. (WWW 2001)*, pages 633–641, 2001.

[13] A. J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133(2):205–265, 1994.

[14] Anthony J. Bonner and Michael Kifer. Transaction logic programming. In David S. Warren, editor, *Intl. Conference on Logic Programming (ICLP)*. MIT Press, 1993.

[15] François Bry and Paula-Lavinia Pătrânjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. In *20th ACM Symp. Applied Computing*. ACM, 2005.

[16] François Bry and Sebastian Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. In *Intl. Conf. on Logic Programming (ICLP)*, number 2401 in LNCS, pages 255–270. Springer, 2002.

[17] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proceedings of the 20th VLDB*, pages 606–617, 1994.

[18] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML. In *8th. WWW Conference*. W3C, 1999. World Wide Web Consortium Technical Report, NOTE-xml-ql-19980819, `www.w3.org/TR/NOTE-xml-ql`.

[19] Document object model (DOM). `http://www.w3.org/DOM/`, 1998.

[20] Chris Dornan, Isaac Jones, and Simon Marlow. *Alex User Guide*. `http://www.haskell.org/alex/`.

[21] Michael Eckert. Reactivity on the Web: Event Queries and Composite Event Detection in XChange. Master's thesis, Institute for Informatics, University of Munich, Germany, 2005.

[22] Andrew Eisenberg and Jim Melton. SQL/XML and the SQLX informal group of companies. *SIGMOD Record*, 30(3):105–108, 2001. See also `www.sqlx.org`.

[23] eXist: an Open Source Native XML Database. `http://exist-db.org/`.

[24] Florid homepage. `http://www.informatik.uni-freiburg.de/~dbis/florid/`, 1998.

[25] Jürgen Frohn, Rainer Himmeröder, Paul-Th. Kandzia, Georg Lausen, and Christian Schlepphorst. FLORID: A prototype for F–Logic. In *Intl. Conf. on Data Engineering (ICDE)*, 1997.

[26] Carsten Gottschlich. To be extended. Master Thesis, Univ. Göttingen, 2006.

[27] Eric N. Hanson and Samir Khosla. An introduction to the triggerman asynchronous trigger processor. In *Rules in Database Systems (RIDS)*, number 1312 in Lecture Notes in Computer Science, pages 51–66. Springer, 1997.

[28] D. Harel. *First-Order Dynamic Logic*. Number 68 in Lecture Notes in Computer Science. Springer, 1979.

[29] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[30] R. Janicki and P. E. Lauer. *Specification and Analysis of Concurrent Systems – the COSY Approach*. EATCS Monographs on Theoretical Computer Science. Springer, 1992.

[31] Jena: A java framework for semantic web applications. `http://jena.sourceforge.net`.

[32] Michael Kay. SAXON: an XSLT processor. `http://saxon.sourceforge.net/`.

[33] Michael Kifer and Georg Lausen. F-Logic: A higher-order language for reasoning about objects, inheritance and scheme. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pages 134–146, 1989.

[34] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.

[35] Tobias Knabke. Development of a domain broker. Master's Thesis, Univ. Göttingen, 2006.

[36] Alexander Kozlenkov and Michael Schroeder. PROVA: Rule-based Java-scripting for a bioinformatics semantic web. In E. Rahm, editor, *International Workshop on Data Integration in the Life Sciences - DILS*. Springer, 2004.

[37] Patrick Lehti. Design and Implementation of a Data Manipulation Processor for an XML Query Language (diploma thesis), August 2001. Technische Universität Darmstadt.

[38] Bertram Ludäscher, Rainer Himmeröder, Georg Lausen, Wolfgang May, and Christian Schlepphorst. Managing semistructured data with florid: A deductive object-oriented perspective. *Information Systems*, 23(8):589–612, 1998.

[39] Simon Marlow and Andy Gill. *Happy User Guide*. `http://www.haskell.org/happy/`.

[40] Wolfgang May. LoPiX: A system for XML data integration and manipulation. In *Intl. Conf. on Very Large Data Bases (VLDB), Demonstration Track*, pages 707–708, 2001.

[41] Wolfgang May. The LOPix system, 2001. `http://dbis.informatik.uni-goettingen.de/lopix/`.

[42] Wolfgang May. A rule-based querying and updating language for XML. In *Workshop on Databases and Programming Languages (DBPL 2001)*, number 2397 in Lecture Notes in Computer Science, pages 165–181, 2001.

[43] Wolfgang May. XPath-Logic and XPathLog: A logic-programming style XML data manipulation language. *Theory and Practice of Logic Programming*, 4(3):239–287, 2004.

[44] Wolfgang May, José Júlio Alferes, and Ricardo Amador. Active rules in the semantic web: Dealing with language heterogeneity. In *Rule Markup Languages (RuleML)*, number 3791 in Lecture Notes in Computer Science, pages 30–44. Springer, 2005.

[45] Wolfgang May, José Júlio Alferes, and Ricardo Amador. An ontology- and resources-based approach to evolution and reactivity in the semantic web. In *Ontologies, Databases and Semantics (ODBASE)*, number 3761 in Lecture Notes in Computer Science, pages 1553–1570. Springer, 2005.

[46] Wolfgang May, Franz Schenk, and Elke von Lienen. Extending an owl web node with reactive behavior. In *Principles and Practice of Semantic Web Reasoning (PPSWR)*, number 4187 in Lecture Notes in Computer Science. Springer, 2006.

[47] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, pages 267–310, 1983.

[48] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[49] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 1(100):1–77, 1992.

[50] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Intl. Conference on Data Engineering (ICDE)*, pages 251–260, 1995.

[51] George Papamarkos, Alexandra Poulovassilis, and Peter T. Wood. Event-Condition-Action Rule Languages for the Semantic Web. In *Workshop on Semantic Web and Databases (SWDB'03)*, 2003.

[52] George Papamarkos, Alexandra Poulovassilis, and Peter T. Wood. RDFTL: An Event-Condition-Action Rule Language for RDF. In *Hellenic Data Management Symposium (HDMS'04)*, 2004.

[53] Pellet: An OWL DL reasoner. Maryland Information and Network Dynamics Lab, `http://www.mindswap.org/2003/pellet`.

[54] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL 1996)*, pages 295–308. ACM Press, 1996.

[55] Paula Lavinia Pătrânjan. *The Language XChange: A Declarative Approach to Reactivity in the Web*. PhD thesis, Institut für Informatik, Ludwig-Maximilians-Universität München, 2005.

[56] Best Practice Recipes for Publishing RDF Vocabularies. `http://www.w3.org/TR/2006/WD-swbp-vocab-pub-20060314/`, 2006.

[57] Rule markup language (ruleml). `http://www.ruleml.org/`.

[58] Sebastian Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Dissertation, University of Munich, Germany, December 2004.

[59] Daniel Schubert. Development of a prototypical event-condition-action engine for the semantic web. Bachelor Thesis, Univ. Göttingen, 2005.

[60] Peter Sheldrick. Development of a ccs engine. Bachelor Thesis, Univ. Göttingen, 2006.

[61] Sebastian Spautz. Automatenbasierte Detektion von Composite Events gemäss SNOOP in XML-Umgebungen. Diplomarbeit, TU Clausthal (in german), 2006.

[62] Igor Tatarinov, Zachary G. Ives, Alon Halevy, and Daniel Weld. Updating XML. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pages 133–154, 2001.

[63] Simon Thompson. *Haskell: The Art of Functional Programming*. Addison-Wesley, second edition, 1999.

[64] Elke von Lienen. Entwicklung eines RDF-Web-Services mit Trigger-Funktionalität. Diplomarbeit, TU Clausthal (in german), 2006.

[65] XML Syntax for XQuery 1.0 (XQueryX). `http://www.w3.org/TR/xqueryx`, 2001.