I5-D10

Pre-standardization of the language

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/Goettingen/I5-D10/D/PU/a1
Responsible editors:	José Júlio Alferes and Wolfgang May
Reviewers:	
Contributing participants:	Goettingen, Lisbon
Contributing workpackages:	I5
Contractual date of deliverable:	29 February 2008
Actual submission date:	10 March 2008

Abstract
This report describes the final state of the *General Framework for Evolution and Reactivity in the Semantic Web* developed within workpackage I5, as well as the MARS approach. MARS – Modular Active Rules in the Semantic Web is a framework for specifying and executing active rules in the Semantic Web. With MARS, active rules can be specified in a modular way. Usually, rules consist of components, in the style *head←body*. For active rules, Event-Condition-Action Rules are a popular paradigm. Such rules consist of an event specification (on which occurrence the rule should be "fired"), a condition (consisting of obtaining additional information by queries, and applying predicates), and an action which is executed when the condition is satisfied. Event specification, Condition, and Action are called the components of the rule. For each of the components, multiple languages have been proposed in the past (e.g., event algebras and process algebras). As it is by far not necessary to come up with even new proposals for models and formalisms for describing events and actions, MARS aims at supporting arbitrary such languages. For this, MARS is just a framework for active rules where appropriate component languages and services can be embedded.

MARS consists of an ontology for ECA rules together with the language proposal ECA-ML that provides a corresponding lean XML markup, a family of XML data exchange formats, and an open ontology-based service-oriented architecture that implements the framework. The proof-of-concept is completed with sample component ontologies for composite events (derived from the SNOOP event algebra), composite actions (derived from the CSS process algebra), and OWLQ, a language for specifying queries, tests, and atomic event patterns.

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.

Pre-standardization of the language

José Júlio Alferes[1], Ricardo Amador[1], Erik Behrends[2], Oliver Fritzen[2], Wolfgang May[2], and Franz Schenk[2]

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

[2] Institut für Informatik, Universität Göttingen

10 March 2008

Abstract

This report describes the final state of the *General Framework for Evolution and Reactivity in the Semantic Web* developed within workpackage I5, as well as the MARS approach. MARS – Modular Active Rules in the Semantic Web is a framework for specifying and executing active rules in the Semantic Web. With MARS, active rules can be specified in a modular way. Usually, rules consist of components, in the style *head*←*body*. For active rules, Event-Condition-Action Rules are a popular paradigm. Such rules consist of an event specification (on which occurrence the rule should be "fired"), a condition (consisting of obtaining additional information by queries, and applying predicates), and an action which is executed when the condition is satisfied. Event specification, Condition, and Action are called the components of the rule. For each of the components, multiple languages have been proposed in the past (e.g., event algebras and process algebras). As it is by far not necessary to come up with even new proposals for models and formalisms for describing events and actions, MARS aims at supporting arbitrary such languages. For this, MARS is just a framework for active rules where appropriate component languages and services can be embedded.

MARS consists of an ontology for ECA rules together with the language proposal ECA-ML that provides a corresponding lean XML markup, a family of XML data exchange formats, and an open ontology-based service-oriented architecture that implements the framework. The proof-of-concept is completed with sample component ontologies for composite events (derived from the SNOOP event algebra), composite actions (derived from the CSS process algebra), and OWLQ, a language for specifying queries, tests, and atomic event patterns.

Keyword List

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

Contents

Chapter 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* – at 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, cf. the Semantic Web tower [15].

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, 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. The underlying concepts are then events, facts and actions that are provided by the individual application ontologies. Based on this, ECA rules provide 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: MARS is a modular framework for *composing* languages for event specification, 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.

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 are objects of the (Semantic) Web, adhering to an own ontology of rules, and marked up in an (XML) Markup Language of ECA Rules.

Structure of the Presentation. The next chapter, Chapter 2, develops the MARS 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 4. In Chapter 5 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 an event algebra similar derived from SNOOP [25] is mapped into the MARS framework. Then, three chapters deal with domain aspects: Chapter 6 analyzes the notion of domain ontologies wrt. the behavioral notions dealt with in the MARS Framework. The architecture of actual domain nodes is then described in Chapter 7. Domain Brokering is then discussed in Chapter 8. The Web architecture and the formal ontology and metadata of the MARS framework is described in Chapter 9. The language and service metadata contained in this ontology is also used for establishing actual communication between the component services. Chapter 10 gives an overview of the prototype on the XML level.

Chapters 11 – 15 continue the approach to the RDF level: Chapters 11 describes how expressions and languages are seen as resources, and how the RDF level and the XML level are related. Chapter 12 lifts the ECA language and the component languages to the ontology level. Chapter 13 investigates how to describe queries, tests, and atomic event specifications on the RDF level. Chapter 14 gives sample ontologies for composite events and composite actions. Chapter 15 puts these things together to complete rules on the RDF level.

Finally, Chapter 16 lists some issues for further work, and Chapter 17 gives a comparison with related work.

Publications from this report. A preliminary version of the MARS framework described in this chapter has been published in [5], the ontology of rules, rule components and languages, and the service-oriented architecture proposal have been published in [53], and the languages and their markup, communication and rule execution model can be found in [52].

The development of the domain node and domain broker architecture is described in [54], [9], and [39].

A first version of the ECA engine has been described in [12]. The proposal to use CCS in the action part is discussed in [10]; a detailed description of the algebraic handling of composite events and actions is given in [11].

Chapter 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 MARS framework, have to be covered.

We start with a requirements analysis in Section 2.1. We continue with describing the structure of domain ontologies in presence of active notions in Section 2.2. Then we discuss the main abstraction levels of (active) rules in the Semantic Web in Section 2.3. We then discuss the main issue of dealing with the above-mentioned heterogeneity of languages. For this we start, in Chapter 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.

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.

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 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 of data formats and description formalisms 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.

2.1.2 Abstraction Levels

Data Model Abstraction Levels. As described above, the *Semantic Web* is 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 local ECA rules that specify which actions are to be taken upon which events/circumstances 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. In *object-relational* architectures, the physical model is the relational one, and the logical model 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 [66] (based on Adabas) and Excelon [30] (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)* [28] 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 [29].

In the research community, models like OEM [58] or F-Logic [42] (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* of the Semantic Web. 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.
- 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); preferably expressed on the logical (RDF) level. On these lower levels, behavior is expressed by "triggers".

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 2.3.

2.2 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 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.

Figure 2.1 depicts the structure of an ontology and the interferences between its components. Concepts can be divided into classes, relationships and individuals (note that by reification, classes and relationships can also be seen as individuals). The actual languages talk about these things by literals. Actions influence classes, relationships, and individuals and raise events.

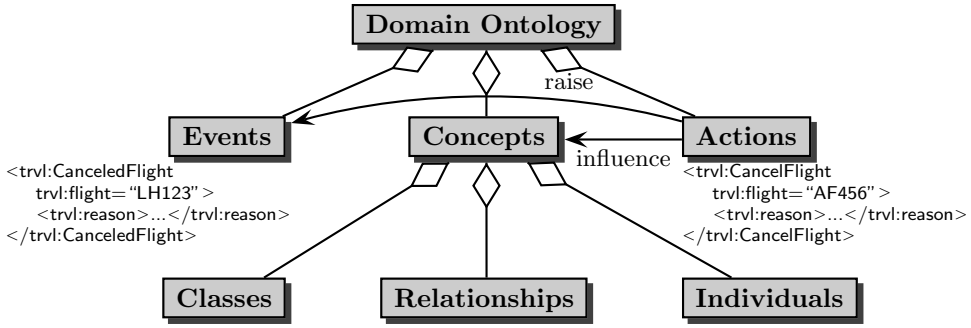


Figure 2.1: Structure and Interference in Ontologies

For designing a Semantic Web application or service, in general ontologies of several domains interfere as shown in Figure 2.2:

- 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 *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.

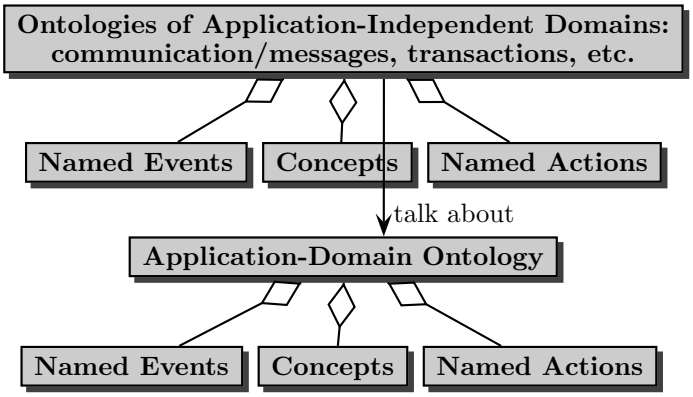


Figure 2.2: Kinds and Components of Ontologies

2.2.1 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:

- Atomic application domain events are the visible happenings in the application domain (e.g., in banking, travel organizing, administration). 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.
- generic parametric events that are not from a 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. Transactional events, i.e., start, commit, and rollback of transactions also belong to this group.

Actions vs. Events. In contrast to simple data-level events on relational (cf. the ON DELETE ... in SQL triggers), 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". Note that there can be multiple derivations for a derived event (for "no more flights ..." e.g., also "flight AF678 on 10.8.2005 is canceled" can be the triggering event). All these events can be used for formulating (business) rules.

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 2.3.

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) events, the Semantic Web must provide *event broker* functionality (integrated within the domain brokering, cf. Section 8).

Temporal Delay and Event Wrapping. Event brokering leads to the effect that the timepoints of actual events and the event detection may differ.

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, any reference to events will usually implicitly have the semantics of "if I get aware of ...".

Composite Events. Composite events, e.g., "if a flight is first delayed and then cancelled", or "from the opening of registration until its end collect all individual registrations, and report them with the end of the registration" are subject of heterogeneity in that there are multiple formalisms and languages for describing them (cf. Section 5.3). As already mentioned, most of them use *event algebras*. MARS supports this heterogeneity.

2.3 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. 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, an event occurrence 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: the occurrence of 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 lists 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 Hanover 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 a 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 [21]). 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 2.3.

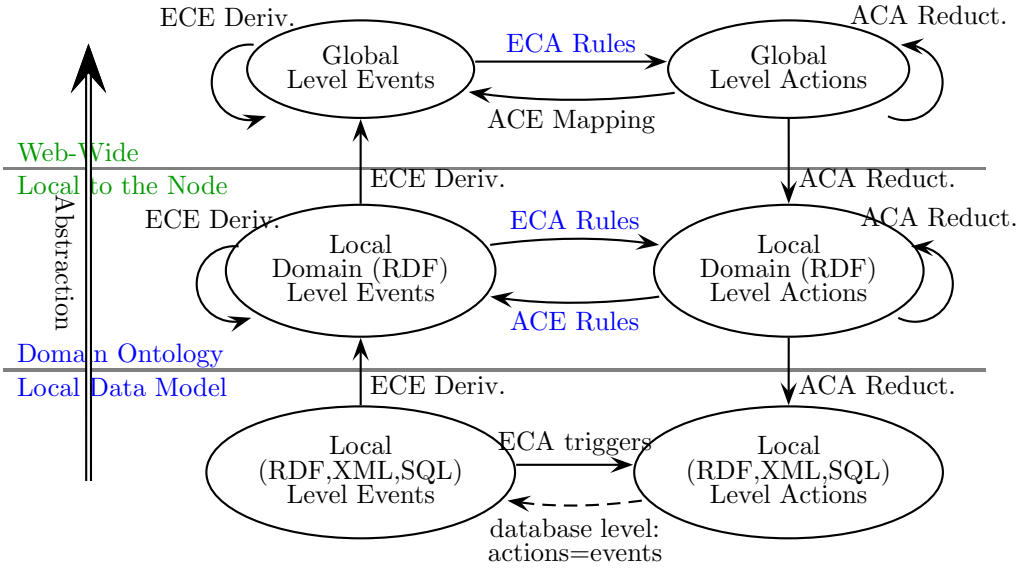


Figure 2.3: Types of Rules

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" also reacts upon the event. 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 travel from X to Y on date D" to a travel agency. The travel agency reacts on incoming messages by searching for connections and booking 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.

Chapter 3

ECA Language Structure

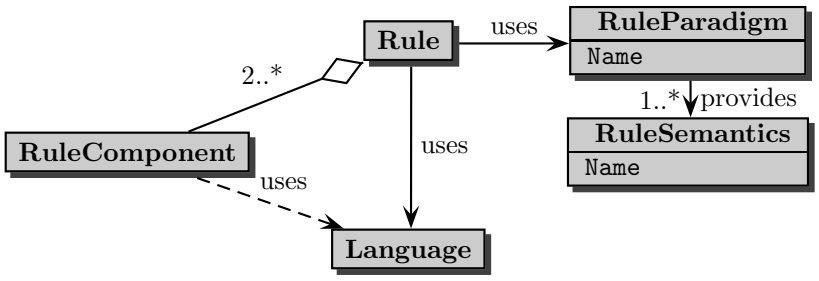
After having discussed and analyzed the setting and requirements for behavior in the Semantic Web by active, ECA-style rules, we develop now the structure of such rules and the required languages.

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 the respective 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 [23], where the event component consists of an event query, the condition component is a test by evaluating an Xcerpt [24] query, and the action component may contain update actions, transactions, or raising of an event. Yet another example is ruleCore [14], where the (different) languages for each component are described. The ECA languages proposed for the Web in ([19, 18, 8, 60]) 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.

For instance, *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 4.



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

Figure 3.1: Rules, Rule Components and Languages

An XML markup agreement for rules must cover the *structure* of rules and rule component languages. Here, the RuleML language [64] provides *general* guidelines that have to 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.).

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 has to 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-style diagram in Figure 3.3, 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 the MARS 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 3.2.

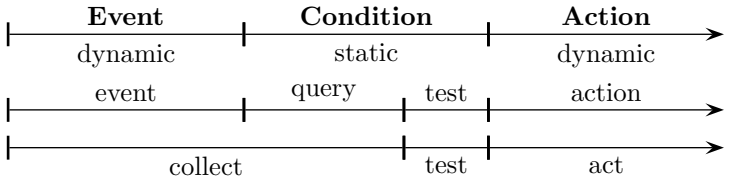


Figure 3.2: Components and Phases of Evaluating an ECA Rule

With this further separation of tasks, the following structure is obtained:

- 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 *all* have 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 3.3.

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 namespaces (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 9; here we investigate the languages and the markup itself.

For an XML representation of ECA rules as shown in Figure 3.3, we propose the below (basic) markup (ECA-ML). This will be refined in the next sections; the full DTD can be found in Appendix 12.5. Later, in Section 11, we develop an *ontology* of rules that defines and relates these concepts. ECA-ML will turn out to be a *canonic* projection of a *canonic* serialization in RDF/XML of the RDF graph of ECA rules. For that,

- class names are capitalized,
- property names are non-capitalized,
- attributes and properties are only associated with instances.

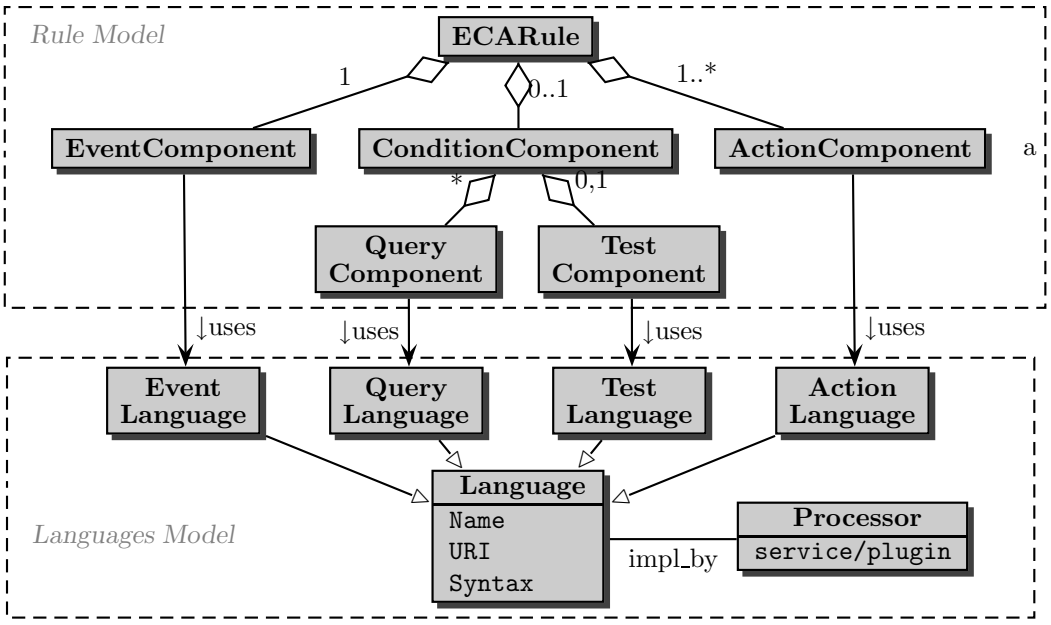


Figure 3.3: ECA Rule Components and Corresponding Languages

```
<!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 >
    event specification, probably binding variables; see Section 4.4.1
  </eca:Event>
  <eca:Query identification of the language >     <!-- there may be several queries -->
    query specification;   using variables, binding others; see Section 4.4.2
  </eca:Query>
  <eca:Test identification of the language >
    condition specification, using variables; see Section 4.4.3
  </eca:Test>
  <eca:Action identification of the language >     <!-- there may be several actions -->
    action specification, using variables, probably binding local ones; see Section 4.4.5
  </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 [19] 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 [22] transaction rules, 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.

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 2.2 and Figure 2.2 to the languages of rules and rule components as shown in Figure 3.4 [here directly associating ontologies with the corresponding languages over the same alphabet]: As described until now, the upper level is formed by the ECA-ML 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 (i.e., event algebras, query languages, boolean tests, process algebras) 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 as described in Section 2.2 , 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 ontologies that provide primitives (with arguments) like general communication, e.g. received_message(M) (where M in turn contains domain-specific content).

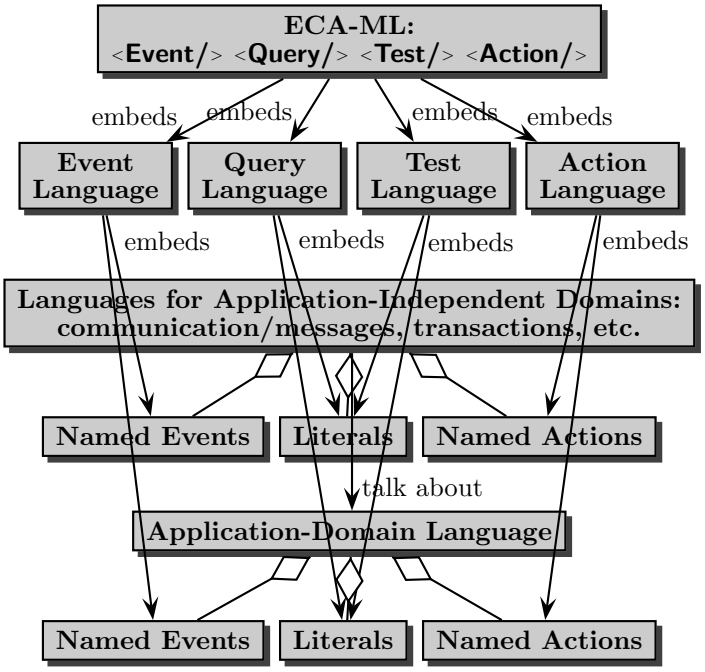


Figure 3.4: Hierarchy of Languages

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 4. Samples of component languages will be discussed in Chapter 5.

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 a common algebraic language structure, although dealing with different notions:

- event languages: every expression describes a (possibly composite) event. Composite expressions are built by composers of an event algebra, and the leaves here are specifications of atomic events of the underlying application domain or an application-independent domain;
- query languages: expressions of an algebraic query language, embedding atomic queries over 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 3.5, 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 3.5 and 3.6).

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 [23], the ruleCore detectors [14], or the SNOOP event algebra of [25]. 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.

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 set of event instances (or sequences) that match the given expression pattern,
- 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.

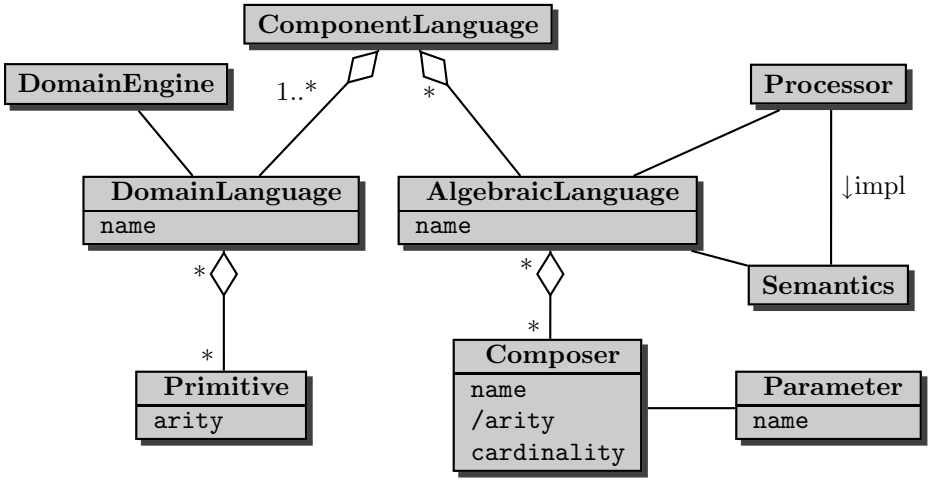


Figure 3.5: Notions of an Algebraic Language

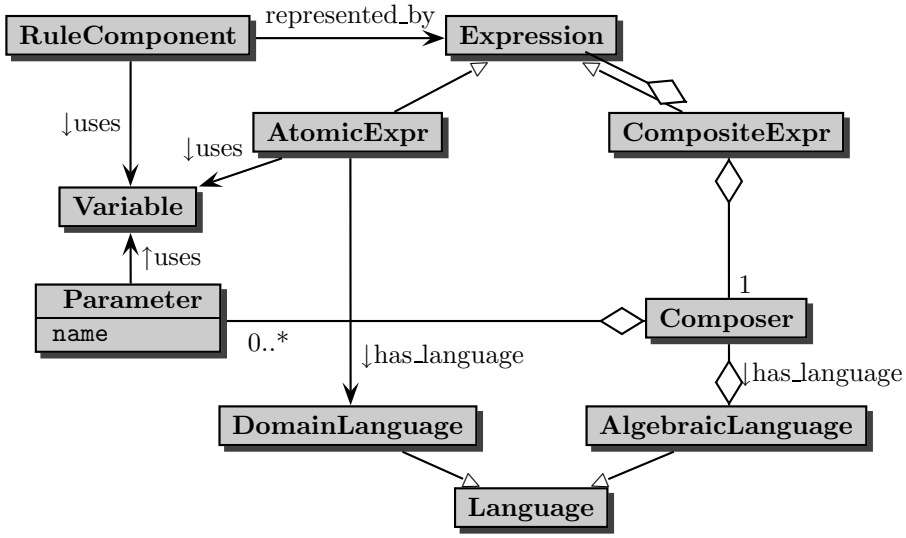


Figure 3.6: Syntactical Structure of Expressions of an Algebraic Language

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 of MARS support this; we show how evaluation also smoothly crosses these language borders in Section 9. Note that Figure 3.6 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 on the *expression* level.

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, 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 3.6, 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).

The language information 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 9 (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). Via the URI, engines that capture the semantics of the (composers of its) language can be found. 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 MARS 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 4.3.1.

3.7 Opaque Rules and Opaque Components

Rules and components can also be given in an "opaque" form. This means that they 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

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 7.2.1 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 3.7.
The ECA-ML language integrates opaque rules by the following markup: The eca:Rule element contains an eca:Opaque element with text content (program code of some rule language). The eca:Rule element also has an attribute language that indicates the URI, or a name alias (e.g. "XQuery" for "http://www.w3.org/XQuery" that must be known to the ECA engine). These names can be handled similar to XML's `NOTATION` concept that associates resources with a program URI.

```
<eca:Rule >
  <eca:Opaque eca: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.

3.7.2 Opaque Components

Analogous to *opaque* rules, MARS allows 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 3.8. In such cases, again <eca:Opaque> elements are used that refer to a URI for that language.
Opaque fragments are marked up as

```
<eca:{Event|Query|Test|Action}>
  <eca:Opaque attributes>
    variable declarations
```

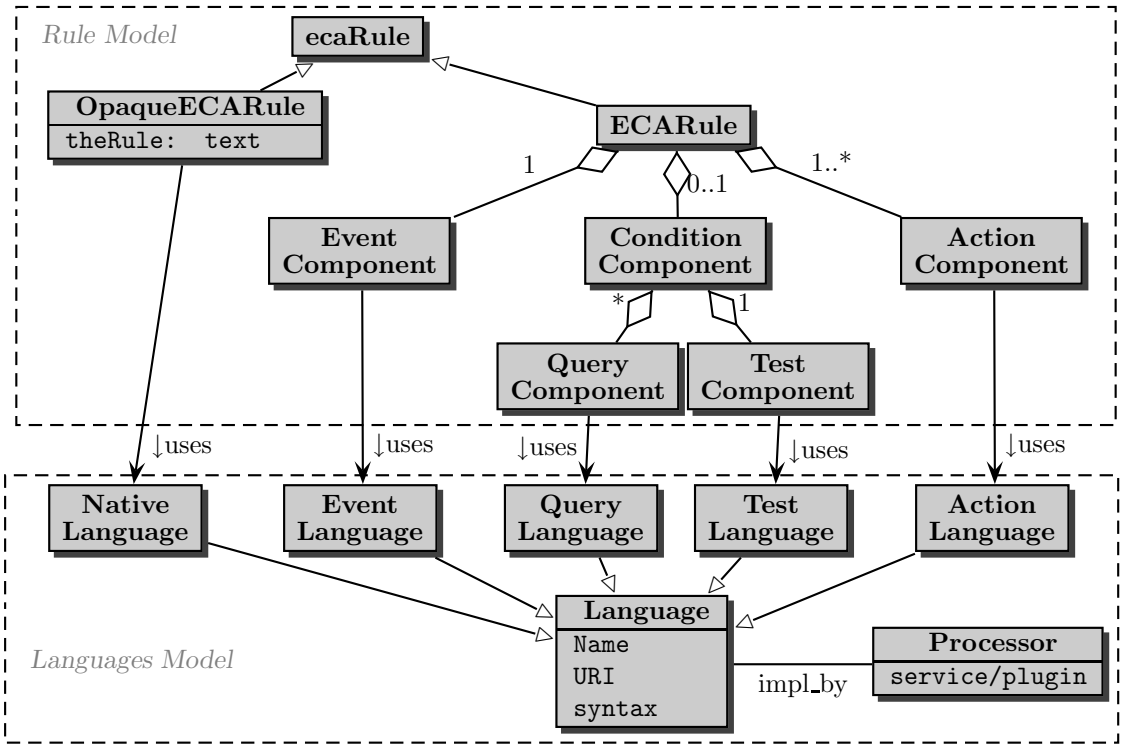


Figure 3.7: ECA Rule Components and Corresponding Languages II

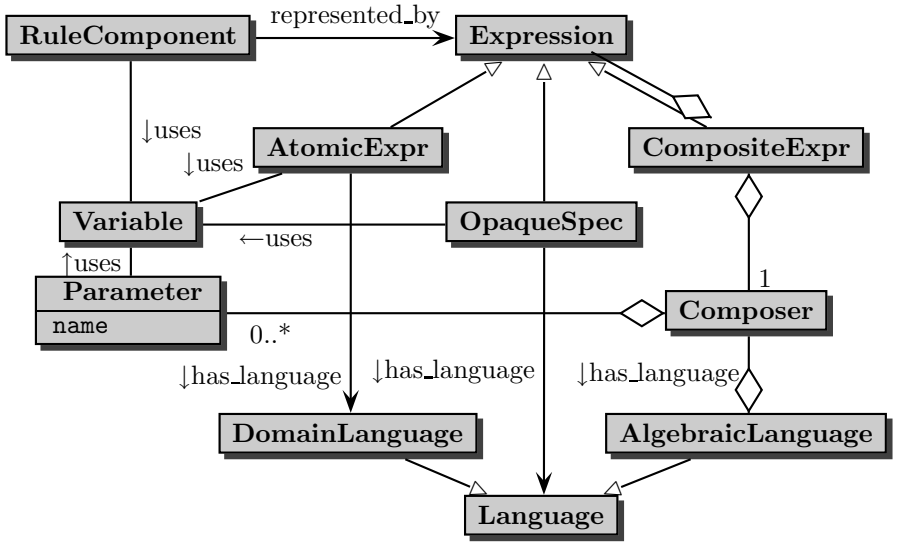


Figure 3.8: Syntactical Structure of Expressions (Algebraic and Opaque)

opaque code[1]
</**eca:Opaque**>
</**eca:**{**Event**|**Query**|**Test**|**Action**}>

[1]Note that if the opaque text contains < or > etc., it must be enclosed into a <`![CDATA[ ... ]]`> sequence. If the opaque contents is an XML fragment itself, the Opaque element must have an attribute "content-type='xml'".

Especially during the development of a prototype, such functionality has been used. We discuss the details of embedding opaque components in more detail in Section 5.2. Wrappers are employed to integrate them into the MARS framework (cf. Section 10.4).

Chapter 4

Rule Level: Abstract Semantics and the ECA-ML Language

This chapter develops the MARS 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 5.

4.1 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 3.2 from the event component via the query components and the test component to the action component.

  For this, we propose to use *logical variables* in the same way as in Logic Programming (cf. Section 4.1.2).

- Vertical communication between the ECA engine and the services 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.

4.1.1 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 [50], Xcerpt [24]; in some sense also the basic forms of SQL and 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:

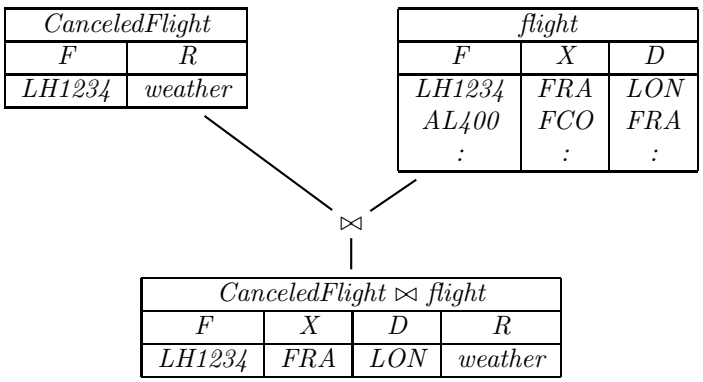
Example 3 (Cancelled Flights) *Consider a rule that should do the following: Whenever a flight is canceled, send a message to the destination airport that the flight will not take place. Assume that events are of the following form*

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

In LP terminology, the event component travel:CanceledFlight(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.*

4.1.2 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 [25] 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 3.2 and 3.3.

- 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 to *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" component (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. 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 [20] that defines its own notions of positive occurrences.

The relationships between rules, rule components and variables on this abstract level are shown in Figure 4.1:

- 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:
  $C.\mathsf{free} = C.\mathsf{positive} \cup C.\mathsf{negative}$.

4.1.3 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: 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 as shown in Figure 4.3:

$$action(X_1, \ldots, X_n) \leftarrow$$
$$\quad event(X_1, \ldots, X_k),\ query(X_1, \ldots, X_k, \ldots X_n),\ test(X_1, \ldots, X_n)$$

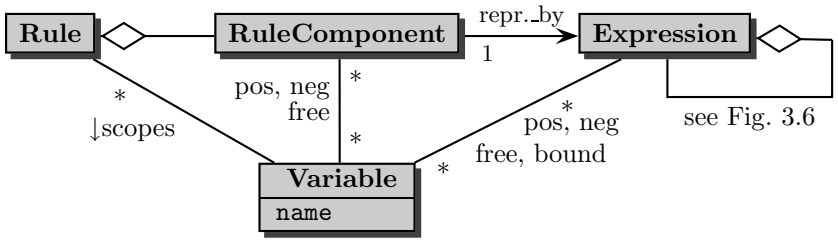


Figure 4.1: Logical Rules and Variables

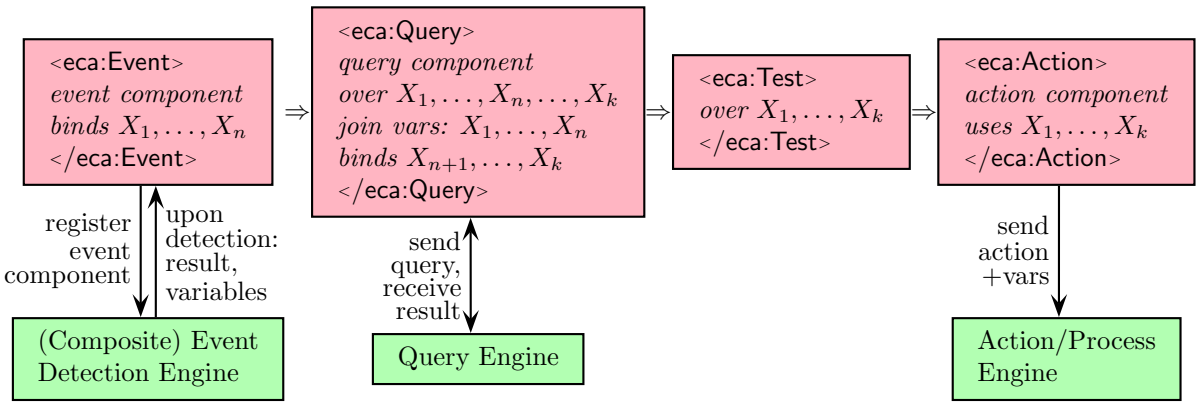


Figure 4.2: Use of Variables in an ECA Rule

Several issues are dealt with in the subsequent sections:

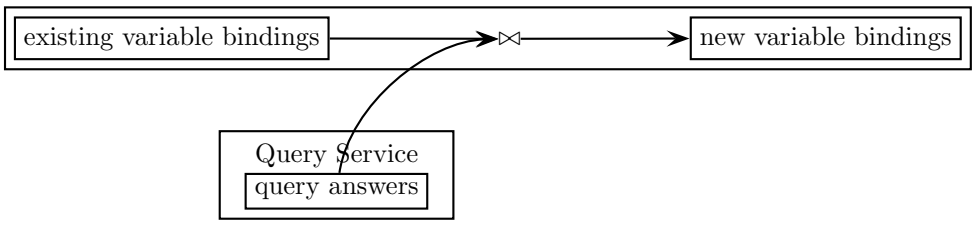
- replacing the abstract predicates by actual event, query, and action components that are specified by sublanguages and evaluated by appropriate services. This requires vertical communication;
- special semantics of events since they are not like usual predicates; expecially, the semantics of event 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;
- optionally: basic transactional functionality.

4.1.4 Vertical Communication

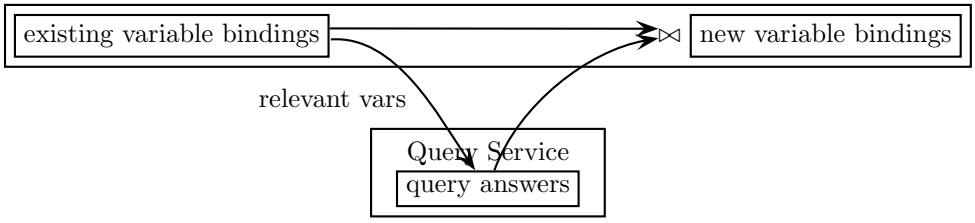
In the same way, the communication with the engines that process the sublanguages is done by sets of tuples of variable bindings. The abstract predicates are now replaced by the actual evaluation. Here, the interplay between the variable bindings on the ECA level and the component services and their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

[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.

- 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 [32]), 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 variables 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.

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

```
available("Frankfurt","Golf", "B", "40").
available("Hanover","Golf", "B", "50").
available("Hanover","Passat", "C", "120").
available("Hanover","S500", "D", "250").
available("Paris","C4", "B", "50").
available("Paris","Golf", "B", "65").
available("Paris","C6", "D", "150").
available("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 one(s) 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.
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, optionally,* 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. Usually, the second case can be expected to be more efficient.*

Consider an extension where a traveler who owns several cars is offered all types 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 4.3 (adapting Figure 4.1) and Figure 4.2. 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. 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 logic-based languages, these are the variables that occur free, but only 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 other languages, 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. In MARS, the above fragment is bound to one variable, and a subsequent query (using local XPath support) extracts the three individual variables.

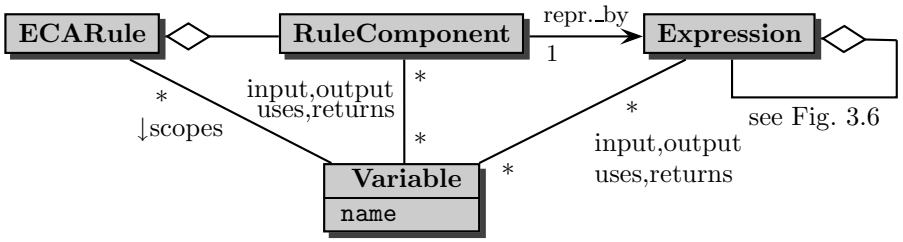


Figure 4.3: Use of Variables in Components of ECA Rules

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 4.1, 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 variables must be taken into consideration.
  - In reality, there are mainly two kinds of services:
    * Logic Programming semantics: outputvars(C) = $\emptyset$ and returned-vars(C) = used-vars(C), and
    * lookup services: output-vars(C) $\neq \emptyset$ and returned-vars(C) = output-vars(C).

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 9).

- 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 HTTP-based Web Service the input and result variables should be defined in a service description.

The markup of components may thus *optionally* contain explicit declarations of the usage of variables. Such 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 Example 4. 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 appropriately with a MARS-aware wrapper (see Section 10.4.2).*

```
<eca:Rule xmlns:eca="http://www.semwebtech.org/languages/2006/eca-ml#"
    xmlns:xpath="http://www.w3.org/XPath"
    xmlns:pseudocode="http://www.pseudocode-actions.nop">
    <eca:Event>
      <eca:has-output-variable eca:name="Person" />
      <eca:has-output-variable eca:name="To" />
      <eca:Opaque eca:language="datalog-match">
        booking(Person,_From,To)
      </eca:Opaque>
    </eca:Event>
    <eca:Query>
      <eca:uses-variable eca:name="Person" />
      <eca:uses-variable eca:name="OwnCar" />
      <eca:Opaque eca:language="datalog-match">
        owns(Person,OwnCar)
      </eca:Opaque>
    </eca:Query>
    <eca:Query>
      <eca:uses-variable eca:name="OwnCar" />
      <eca:uses-variable eca:name="Class" />
      <eca:Opaque eca:language="datalog-match">
        class(OwnCar,Class)
      </eca:Opaque>
    </eca:Query>
    <eca:Query>
      <eca:has-input-variable eca:name="To" />
```

```
    <eca:has-output-variable eca:name="Class" />
    <eca:has-output-variable eca:name="Model" />
    <eca:has-output-variable eca:name="Price" />
    <eca:Opaque eca:uri="http://localhost/lookup-cars">
      ?_place=To
    </eca:Opaque>
  </eca:Query>
  <eca:Action>
    <eca:has-input-variable eca:name="Model" />
    <eca:has-input-variable eca:name="Price" />
    <eca:Opaque eca:uri="localhost">
      show all model-price tuples
    </eca:Opaque>
  </eca:Action>
</eca:Rule>
```

4.2 Logical Variables: Markup for Communication

4.2.1 Basic Interchange of Variable Bindings

We propose the following representation for variable bindings that will be used for interchange (see also Section 4.2.3):

```
<!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/languages/2006/logic#. The complete logvars DTD can be found in Appendix C.1.

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

4.2.2 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 5.4 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 4.1.2). 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 4.2.3) by logvars:answers or logvars:answer at *rule evaluation time* when the event is actually detected (see Section 5.5).

- 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*.

The actual communication format for these identifiers ("Subject") and Reply-To via HTTP messages (headers etc.) is described in Section 5.4.

4.2.3 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 4.2.1 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 [42], XPathLog [50]). Here, the matches can be *literals* (Datalog), literals and structures (e.g., in F-Logic, XPathLog, Xcerpt), or literals and (URI) references (e.g., in SPARQL). 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 [24] – 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, event languages).

To cover all of them, we propose a structure as e.g. used in the Florid system [32] where with *each* result, a set of variable bindings is associated (cf. [46, 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 C.1).

```
<!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).
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 <logvars: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 5.4 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 rule/event specification 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>).

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 4.1.5).
- 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.

$\Rightarrow$ the service descriptions of the language services indicate whether the resulting variable bindings are a superset of the input/used variables (cf. Section 4.1.5 and Section 9.5).

4.3 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.

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-ML language (see also Section 4.3.2), and in the E, Q, T, and A component languages. In the next paragraphs we discuss some design alternatives how to integrate the handling of variables into languages.

4.3.1 Possible Syntaxes

There are several possible constructs for binding variables, borrowed from several language designs:

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

```
<foo:bla foo:bind-to-variable="name">
  content
</foo:bla>
```
or
```
<foo:bla>
  <foo:bind-to-variable foo:name="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 binding variables in its query terms.

The syntax corresponds most directly to RDF, i.e. "annotating" an element with a reference to a variable.

- The use of variables is indicated *inside* of elements where it refers to, but it has to be processed by the *outside*;
- no problem when it is integrated with the DTD (as in ECA-ML);
- when submitting components to services, the attribute/element must be removed;
- not applicable in *patterns* (e.g. for matching something) since it violates the original DTD.

2. Logic Programming/Pattern matching style; (syntax for use borrowed from XQuery, XSLT and XML-QL; syntax and semantics for binding borrowed from XML-QL [26]): (use) variables by {$*var-name*}:

```
<travel:CanceledFlight number=$flight/>
```
or
```
<travel:CanceledFlight number="{$flight}" />
```

matches an event (e.g., <travel:CanceledFlight 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.

- only usable for simple cases (attributes or whole element contents),
- well-suited for use in patterns (XML-QL style). This *atomic event specification language (AESL)* will be considered again in Section 5.3.2.1.

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

```
<foo:variable foo: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 12 and 8 below). Note that the foo:variable elements reside in the component language's namespace (the variable is part of the component language tree).

This interferes with the term/pattern 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.

This alternative is *not recommended*, neither for patterns nor for the design of a component language.

Next, we discuss this issue for ECA-ML. Another discussion can be found in Section 5.3.5 when a markup for a composite event language closely related to the SNOOP event algebra [25] is presented.

4.3.2 Variable Bindings by ECA-ML: Components Results

For making the functional result part of a component (i.e., of the event component or the query component) accessible in the ECA rule, it must immediately be bound to a variable on the rule level. For this, ECA-ML associates the information to which variable the result is to be bound with the component expression.

- eca:bind-to-variable attribute on the ECA component level (general: as an attribute to the respective expression).

```
<eca:Rule...>
  <eca:Event eca:bind-to-variable="result-var">
    contents
  </eca:Event>
    :
</eca:Rule>
```

- eca:bind-to-variable subelement on the ECA component level (general: as a subelement of the respective expression). This syntax fits the RDF ontology that will be presented in Section 12 (note that in RDF, instead of name="*name*", rdf:resource="*uri*" is then used).

```
<eca:Rule...>
  <eca:Event>
    <eca:bind-to-variable eca:name="result-var" />
    contents – an expression in an event markup language
  </eca:Event>
    :
</eca:Rule>
```

The information about the use of a variable is indicated *at* the component element. It is intended to be used by the service that processes the *outside*; from that point of view, the attribute syntax is more intuitive. The actual component is the content of the element. Only this is then sent to the event component service.
With the eca:bind-to-variable directive, the respective variable is bound as follows:

- If the result from evaluating a component contains one or more <logvars:answer> elements, 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 12 and 8 and below).
- If the result is a literal (including an XML fragment), *result-var* is simply bound to that value.

Note that most currently existing tools (e.g. query interfaces) do not return their data in the exchange format defined by logvars. In such cases, wrappers can be used (MARS provides wrappers for XPath and XQuery). 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 10.4).

Usually, event detection returns the relevant event sequence in this way, we illustrate this situation in Section 4.4.1.

4.3.3 Variable Bindings in ECA-ML: Initialization

Additionally, the initialization of variables to values, and the computaion of dependent values is supported in ECA-ML:

- <eca:initialize-variable> elements of the form

  <eca:initialize-variable name="*name*"> *constant value* </eca:initialize-variable>

  can be used for initializing variables with constant values.

- Shortcut for XPath: <eca:initialize-variable> elements of the form

  <eca:initialize-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:initialize-variable** eca:name="y" eca:language="xpath" eca: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:Query eca:bind-to-variable="name"**>
  <**eca:Opaque** eca:language="xpath"> *expr* </**eca:Opaque**>
  </**eca:Query**>

4.4 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 5. We first clarify the canonic operational semantics of the ECA engine.

4.4.1 Firing ECA Rules: the Event Component

The operational semantics of ECA rules differs 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.

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 4.1.4). 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 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.

As shown in the next example, an answer from the event detection module can result in

- one answer, containing several tuples, or
- several answers, containing one tuple.

Example 7 (Multiple Matches in a Single Event) *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 (this example will continued later in Example 38):*

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

Here, a direct matching for "a flight F is delayed by M minutes" will result in multiple matches.

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, the correct (but not always most efficient) semantics is that the ECA engine immediately separates them and fires independent instances of the rule (see Section 4.4.4 and 16.4).

In many formalisms (e.g., in SNOOP [25]), 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, afterwards the values of other variables can be extracted from this one. This is especially relevant in case of *cumulative* events:

Example 8 (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 uni:subject="Databases"/>
<uni:register uni:subject="Databases" uni:name="John Doe"/>
<uni:reg_close uni:subject="Databases"/>
```

An rule 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 [25]. The rule is then fired for this one event; the query part extracts the individual registrants from the event, resulting in multiple tuples of variables (using XPathLog and regular expression syntax in an obvious way):*

```
<eca:Rule ... >
  <eca:uses-variable eca:name="Subj" />
  <eca:Event>
    <eca:Opaque eca:language='xpathlog-events'>
      <eca:bind-to-variable eca:name="regseq" />
      <eca:has-positive-variable eca:name="Subj" />
      uni:reg_open[@uni:subject→Subj], uni:register[@uni:subject→Subj]*, uni:reg_close[@uni:subject→Subj]
    </eca:Opaque>
  </eca:Event>
  <eca:Query>
    bind a variable $Student to each of the students
  </eca:Query>
  <eca:Action>
    insert registered($Student,$Subject) into the database
  </eca:Action>
</eca:Rule>
```

The returned information from the event detection service in the markup proposed in Section 4.2.3 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 uni:subject="Databases" />
    <uni:register uni:subject="Databases" uni:name="John Doe" />
    <uni:register uni:subject="Databases" uni:name="Scott Tiger" />
      :
    <uni:reg_close uni: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 uni:subject="Databases" />
      <uni:register uni:subject="Databases" uni:name="John Doe" />
      <uni:register uni:subject="Databases" uni:name="Scott Tiger" />
        :
```

```
    <uni:reg_close uni:subject="Databases" />
   </logvars:variable>
   <logvars:variable name="Subj">Databases</logvars:variable>
  </logvars:tuple>
</logvars:variable-bindings>
```

4.4.2 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.

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 separate 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
  <*elementname*>{ for ... where ... return ...} </*elementname*> ,
  then the whole result yields a single variable binding.

Example 9 *Consider again Example 8 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:Query>
   <eca:Opaque eca:language='xpath'>
    <eca:bind-to-variable eca:name="Student" />
    <eca:has-input-variable eca:name="regseq" />
```

```
      <eca:has-input-variable eca:name="Subj" />
      $regseq//uni:register[@uni:subject=$Subj]/@uni:name/string()
    </eca:Opaque>
  </eca:Query>
    :
</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:Query>
    <eca:Opaque eca:language='xquery'>
      <eca:bind-to-variable eca:name="Students" />
      <eca:has-input-variable eca:name="regseq" />
      <eca:has-input-variable eca:name="Subj" />
      <students>
        { for $s in $regseq//uni:register[@uni:subject=$Subj]/@uni:name/string()
          return <name> { $s } </name> }
      </students>
    </eca:Opaque>
  </eca:Query>
    :
</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.

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 10 *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/languages/2006/eca-ml#">
  <eca:initialize-variable eca:name="Bookings"> http://localhost/schedule.xml </eca:initialize-variable>
  <eca:uses-variable eca:name="Flight" />
  <eca:Event>
    <eca:Opaque eca:language='datalog-match'>
      <eca:has-positive-variable eca:name="Flight" />
      flight_cancellation(Flight)   <!-- matches Flight against received message -->
    </eca:Opaque>
```

```
</eca:Event>
<eca:Query>
  <eca:Opaque eca:language="xquery">
    <eca:bind-to-variable eca:name="Customers"/>
    <eca:has-positive-variable eca:name="Flight"/>
    <eca:has-positive-variable eca:name="Bookings"/>
    return
      <customers>
      { for $c in document($Bookings)//flight[@id=$Flight]/reservation/customer
        return $c }
      </customers>
  </eca:Opaque>
</eca:Query>
<!-- evaluates XQuery expression and binds the result to the variable 'Customers' -->
<eca:Test>
  <eca:Opaque eca:language='xpath'>
    <eca:has-input-variable eca:name="Customers"/>
    $Customers/customer
  </eca:Opaque>
</eca:Test>
<eca:Action>
  <eca:Opaque eca:language='pseudocode'>
    <eca:has-input-variable eca:name="Customers"/>
    <eca:has-input-variable eca:name="Flight"/>
    send one message with all Customers/customer/name to the hotel,
    and
    for each N in Customers/customer/@phonenr do
      notify_cancellation(Flight, sms:N)
  </eca:Opaque>
</eca:Action>
</eca:Rule>
```

The action part will be reconsidered in Example 32.

Note that there can be query services that return the query answer stepwise and asynchronously (which makes sense if the receiver is then able to continue already for some of the answers). This is considered in the service description of the query service.

4.4.3 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 functions and predicates with boolean combinators and quantifiers. Note that the test component must not access remote data sources (this is done in the query part), but uses only built-in, domain-independent functions and predicates as e.g. provided by the *XQuery and XPath Functions and Operators* [72]. A markup language exists with FOL-RuleML [17]. Since first-order logic is in general undecidable, it is recommended to use suitable fragments. Additionally, we envisage to allow to use simple expressions like XPath of a language that is locally supported in the ECA engine. 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). In general, the test component is a conjunction of literals and is is evaluated locally by the ECA engine.

4.4.4 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.

4.4.5 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 on the rule level, thus they are independent from each other.

4.4.6 Examples

Example 11 (Rental Cars (Revisited)) *Consider again the car rental rule from Example 5. 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 12 *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/languages/2006/eca-ml#"
    xmlns:xpath="http://www.w3.org/XPath"
  <eca:initialize-variable eca:name="Schedule">http://localhost/schedule.xml</eca:initialize-variable>
  <eca:uses-variable eca:name="Flight" />
  <eca:uses-variable eca:name="Destination" />
  <eca:Event>
    <eca:Atomic eca:language="xmlqlmatch">
      <eca:bind-to-variable eca:name="event" />
      <travel:CanceledFlight/>
      <!-- matches any travel:CanceledFlight event, e.g. <travel:CanceledFlight code="LH1234" /> -->
    </eca:Atomic>
  </eca:Event>
    <!-- the matched event is now bound to the variable $event -->
  <eca:initialize-variable eca:name="Flight" eca:language="xpath"
      eca:select="$event/CanceledFlight/string(@code)" />
  <eca:Query>
    <eca:Opaque eca:language='xpath'>
      <eca:bind-to-variable eca:name="Destination" />
      <eca:has-input-variable eca:name="Flight" eca:use="$Flight" />
      <eca:has-input-variable eca:name="Schedule" eca:use="$Schedule" />
      string(document($Schedule)//flight[@id=$Flight]/@to)
    </eca:Opaque>
  </eca:Query>
  <!-- evaluates XPath expression and binds the result to the variable 'Destination' -->
  <eca:Action>
    <eca:Opaque language='pseudocode'>
      <eca:has-input-variable eca:name="Flight" eca:use="$Flight" />
      <eca:has-input-variable eca: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:CanceledFlight 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 is 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. There is no condition component. Next, the pseudocode fragment in the action component is equipped with the flight number and the destination airport and a message is sent.

Chapter 5

Component Languages and Services

In the previous section, the event, query and action components were given by simple opaque expressions. As discussed in Section 3, the components are usually given by specialized languages for describing composite events, complex queries, and processes. The (usually algebraic) component languages provide the glue between atomic notions and the ECA level. Such component languages/services and domain languages/services can be provided by everybody somewhere in the (Semantic) Web.

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).

The communication from the point of view of the *rule semantics*, i.e., transmission of variable bindings has already been discussed in Chapter 4; expecially the formats for downward and upward communication in Section 4.2. In this chapter, this is extended from the point of view of the component services.

First, the general requirements and strategies of designing a markup are analyzed. We then discuss the structure, syntax, and semantics of the event component and its abstract communication issues. Next, we discuss the actual means of communication. The abstract semantics and markup of the query and test components are simpler since they are in general an embedding of query languages in the MARS Framework. The discussion of the action component completes the discussion of the components. A short summary concludes the chapter.

5.1 Design Decisions of the XML Markup

The basic design of the ECA-ML markup has been introduced in Sections 3.3 (ECA-ML) and 4.3 (variables).
The following design decisions have been taken so far:

- tree structure of the rules, components to be embedded,
- variables: binding to variables, and declaration of use of variables is attached *as subelements* of the respective subexpressions.

These decisions are compatible with the RDF model of rules and components as will be described in detail in Sections 11 – 15. Some additional issues when considering the composition of languages are pointed out next.

In RDF, a node can be of many types, e.g., an eca:Event and a snoopy:Sequence, or an eca:Event and an atomic event specification of some language, e.g., aes-xmlql:Query (note that Atomic Event Specifications are actually queries against atomic events). Additionally, such nodes may have

properties (e.g., variable declarations like eca:bind-to-variable) wrt. the surrounding language, and wrt. the local language snoopy:bind-to-variable. In contrast, in XML, every element has only one element name, and a namespace of that element name. A main feature of the MARS processing model wrt. XML data is that *every subtree is processed by the processor that is responsible for the language indicated by the root element of the subtree*. For that, expression nodes at the language border that have properties wrt. both language are divided into an outer and an inner part (this corresponds in some sense to casting into a more specific subclass for the inner part; this will be discussed later in Section 11.4.2 when mapping RDF into RDF/XML markup).

- nested expressions inside a language are immediate subexpressions (cf. algebraic expressions discussed below);
- if a subexpression comes from a different language, a "wrapper" that indicates its role wrt. the surrounding language is added. Actually, the eca:Event element around snoopy:Sequence is such a wrapper.
- this holds especially, when languages on different levels are combined, e.g., an Event Algebra expression whose leaves are expressions in an Atomic Event Specification formalism.
- variable specifications must be associated with the wrapper element if they refer to the surrounding language, or associated with the subtree, when they refer to the embedded language.

This will be made explicit with the presentation of the reference languages for the event component in Section 5.3.

5.2 Opaque Components

Opaque fragments have been introduced in Section 3.7 with eca:Opaque elements. Opaque components do not fully fit in the schema that the namespaces of the embedded elements allow for identifying the languages. 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:Query element has a eca:language attribute whose value may either be a URI, or a namespace prefix or language shortname (that must be known in some way by the *Language and Service Registries (LSRs)* that organize the assignment of processors, see Section 9.5.1).

```
<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 4.3.1).

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

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

  ```
  <eca:Query>
    <eca:Opaque eca:uri="http://exist.dbis.uni-goe.de/exist/servlet/db/mondial.xml"
        eca: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 eca:uri="http://simple-service.dbis.uni-goe.de" eca:method="get">
      ?name='Germany'
    </eca:Opaque>
  </eca:Query>
  ```

  Discussion (including definitions of Section 4.1.5):

  – 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 9 is associated with it) with this part, all other information in the contents.

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 (MARS)
http://...	–	get/post	http with indicated method
ip-address	tcp	–	tcp
ip-address	–	–	native (MARS)

Note that it is possible and useful to use opaque XQuery code to write ad-hoc wrappers for non-MARS-aware services.

5.3 The Event Component: Structure and Languages

The semantics of the event component and its services is separated into two levels. The event component language is embedded in the language hierarchy as shown in Figure 3.4. It is based on the event ontology as discussed in Section 2.2.1. Composite events are defined as a certain combination of atomic events. Thus, the event component of a rule, which is a *specification* of a composite event, consists of the specification of the combination (using a *Composite Event Language (CEL)*), and of specifications of the contributing atomic events (using *Atomic Event Specification Languages (AESLs)*). Thus, languages of two types are needed:

Composite Event Languages (CELs); often Event Algebras: They are used to define composite events, including their specification as algebra terms as shown in Figure 3.6 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 Specification Languages (AESLs); lightweight query languages: 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 atomic events from the domain ontologies.

Thus, the event component uses a combination of one ore more event algebras, and one or more atomic event specification formalisms that express which atomic domain events are relevant.

5.3.1 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 which atomic events are relevant to react upon in the event components of ECA rules.

5.3.1.1 XML Representation of Atomic Events

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

Example 13 (Atomic Events) *Events are data fragments that are available in XML markup or as RDF fragments.*

- *The event*

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

  is an event in the traveling domain that means 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:CanceledFlight travel:flight="LH1234" travel:date="10062005"/>.
  An example for a delayed flight is

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

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

<uni:reg_open uni:subject="Databases"/>
<uni:register uni:subject="Databases" uni: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 (cf. Chapter 15).*

5.3.2 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 5.1:

- 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.

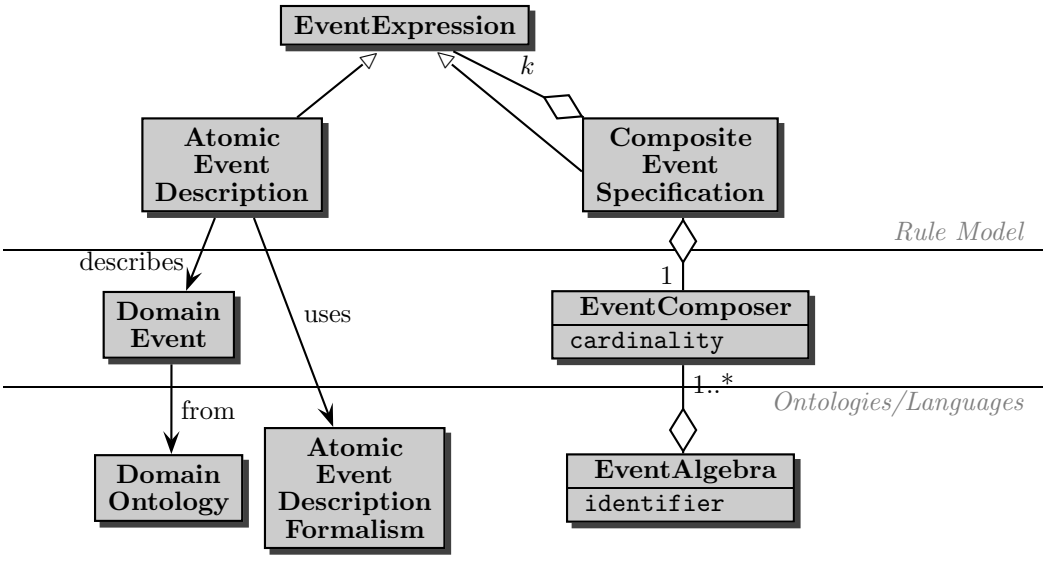


Figure 5.1: Event Expressions: Languages

Since the event is seen as an XML or RDF fragment, the AES formalism is actually a query language that states these conditions against 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 language in which the AES is given, to be able to find an appropriate processor for it.
- Information about the domains that provide the actual events is required to be accessible for the service that then actually processes the AES. This information is usually given by the domain namespace of the event (XML), or by its rdfs:isDefinedBy property (RDF).

- 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 discussed later in Section 5.3.7.

5.3.2.1 Event Specification by XML-QL Style Matching

Pattern-based (e.g. like XML-QL [26]) 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:DelayedFlight xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
    eca:language="xmlqlmatch" travel:flight="{$flight}" travel:time="{$minutes}">
  $content
</travel:DelayedFlight/>
```

specifies that an event is relevant if it is a travel:DelayedFlight event. In case that the variable $flight is already bound, this acts as a (join) condition on the code of the delayed 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:CanceledFlight xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
    xmlns:aes-xmlql="http://www.semwebtech.org/languages/2006/aes-xmlql#"
    travel:flight="{$flight}">
  <aes-xmlql:variable name="reason">
    <travel:reason/>
  </aes-xmlql:variable>
</travel:CanceledFlight>
```

matches any travel:CanceledFlight event concerning a given flight 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 [67] and is available in the MARS prototype.

5.3.2.2 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,

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

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

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

or

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

The AESL has been implemented and is available in the MARS prototype.

Another AESL is based on the OWLQ language, and will be described in Section 13.4.2.

5.3.3 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.

Formally, an event is a predicate $E : T \rightarrow \{\mathsf{true}, \mathsf{false}\}$ where T denotes a set of time instances. With its occurrence, the sequence of all "contributing" atomic events is reported. 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 [25] of the "Sentinel" active database system:

Definition 1 (Semantics of Snoop) *The formal semantics of the temporal operators of the SNOOP event algebra [25] is given in Table 5.1 (as usual, $:\Leftrightarrow$ denotes "defined as").*

(1) $(E_1 \nabla E_2)(t) \quad :\Leftrightarrow \; E_1(t) \vee E_2(t)$,

(2) $(E_1 \triangle E_2)(t) \quad :\Leftrightarrow \; \exists t_1 \leq t \leq t_1 : (E_1(t_1) \wedge E_2(t)) \vee (E_2(t_1) \wedge E_1(t))$.

(3) $(E_1 ;_{\Delta t} E_2)(t) \; :\Leftrightarrow \; \exists t_1 \leq t \leq t_1 + \Delta t : E_1(t_1) \wedge E_2(t)$.

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

(5) $\neg(E_2)[E_1, E_3](t) \; :\Leftrightarrow$
$E_3(t) \wedge \neg E_2(t) \wedge (\exists t_1 : E_1(t_1) \wedge (\forall t_2 : t_1 \leq t_2 < t : \neg(E_2(t_2) \vee E_3(t_2))))$,
when this event occurs, $(E_1; E_3)$ is reported.

(6) $A(E_1, E_2, E_3)(t) \; :\Leftrightarrow \; E_2(t) \wedge (\exists t_1 : E_1(t_1) \wedge (\forall t_2 : t_1 \leq t_2 < t : \neg E_3(t_2)))$,
when this event occurs, $(E_1; E_2)$ is reported.

(7) $A^*(E_1, E_2, E_3)(t) \; :\Leftrightarrow \; E_3(t) \wedge \exists t_1 \leq t : E_1(t_1)$,
when this event occurs, the sequence consisting of E_1, all occurrences of E_2 during the interval, and E_3 is reported.

Table 5.1: Formal Semantics of the SNOOP Event Algebra

Example 14 (Cumulative Event, [25]) *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":

$$E(Acct) :=$$
$$A^*(first\_of\_month(m), (debit(Acct,Am) \nabla deposit(Acct,Am)), first\_of\_month(m+1))$$

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

5.3.4 Variables in Composite Events

Variables can be bound to values, e.g., strings, numbers, XML fragments, or nodes in an RDF model given as URIs. In the MARS model, also *events* are seen as such values (or as resources, when considering the RDF model). In the event algebra extension presented here, variables can be bound to values obtained from matching atomic events, and variables can be bound to (sub)events that match an event algebra (sub)term during detection. If a variable is used twice, the values must coincide. Binding event instances (i.e., their XML markup or URI) to variables allows to

(i) have a closer look at exactly this event later in the query or action component, and

(ii) variables occurring twice in an event term act as join variables for guaranteeing that one (e.g. atomic) event in two subterms is actually the *same* (same occurrence). As shown in the next example, this adds some expressiveness that can be used for synchronization.

Formally, the definition of the SNOOP semantics is extended as follows:

Definition 2 (Definition of Composite Events with Variables) *The definition of the truth (i.e., occurrence of a matching sequence of events) of composite event expressions containing variables at a given timepoint t extends Definition 1: A composite event specification $E(X_1, \ldots, X_n)$ with variables $X_1, \ldots, X_n$ is satisfied under a variable assignment $\beta : X_1, \ldots, X_n \to D$ (where D is the underlying domain) in virtue of an algebraic expression (using only "$\triangle$" and ";") over the contributing events. The formal definition is given in Table 5.2.*

Example 15 (Event Algebra with Join Variables) *The composite event specification*

$$(\neg(E_1(X))[E_2(X) \to V_1, E_3(X) \to V_2]) \triangle (E_2(X) \to V_1; E_4(X); E_3(X) \to V_2)$$

is detected if $E_2(a)$ and $E_3(a)$ occur for some a (binding the variable X to the literal a), and in-between, $E_1(a)$ did not occur, but $E_4(a)$ occured. The variable bindings $E_2(X) \to V_1$ in the subexpressions bind the variable V_1 to the whole event of $E_2(a)$ (at its occurrence, which is different from all earlier or later occurrences of the "same" event; in RDF this is done by associating the event occurrence with a unique "blank" URI, in XML this can be done by attaching a unique attribute in the local namespace) and guarantee that both subterms refer to the same event instance *and not only for two E_2 instances with the same value a: without the variable binding, this would be reported for* any *two pairs of $(E_2(x); E_3(x))$ where between one of them, $E_1(x)$ did not occur, and between the other one, $E_4(x)$ occured.*

$$
\begin{array}{ll}
(1) & (E_1 \nabla E_2, \beta, e)(t) \quad :\Leftrightarrow \ (E_1, \beta, e)(t) \vee (E_2, \beta, e)(t) \ , \\
(2) & (E_1 \triangle E_2, \beta, (e_1 \triangle e_2))(t) \ :\Leftrightarrow \\
 & \quad \exists t_1 \leq t : ((E_1, \beta, e_1)(t_1) \wedge (E_2, \beta, e_2)(t)) \vee ((E_2, \beta, e_2)(t_1) \wedge (E_1, \beta, e_1)(t)) \ , \\
(3) & (E_1; E_2, \beta, (e_1; e_2))(t) \quad :\Leftrightarrow \ \exists t_1 \leq t : (E_1, \beta, e_1)(t_1) \wedge (E_2, \beta, e_2)(t) \ , \\
(4) & (\mathsf{ANY}(m, E_1, \ldots, E_n), \beta, (e_{i_1}; \ldots; e_{i_m}))(t) \ :\Leftrightarrow \ \exists t_1 \leq \ldots \leq t_{m-1} \leq t, \ 1 \leq i_1, \ldots, i_m \leq n \\
 & \quad \text{pairwise distinct s.t. } (E_{i_j}, \beta, e_{i_j})(t_j) \text{ for } 1 \leq j < m \text{ and } (E_{i_m}, \beta, e_{i_m})(t) \ , \\
(5) & (\neg(E_2)[E_1, E_3], \beta, (e_1; e_3))(t) \ :\Leftrightarrow \ (E_3, \beta, e_3)(t) \wedge \neg\exists e : (E_2, \beta, e)(t) \wedge \\
 & \quad \exists t_1 : ((E_1, \beta, e_1)(t_1) \wedge (\forall t_2 : t_1 \leq t_2 < t : \neg\exists e : (E_2, \beta, e)(t_2) \vee (E_3, \beta, e)(t_2))) \ , \\
(6) & (A(E_1, E_2, E_3), \beta, (e_1; e_2))(t) \ :\Leftrightarrow \\
 & \quad (E_2, \beta, e_2)(t) \wedge (\exists t_1 : (E_1, \beta, e_1)(t_1) \wedge (\forall t_2 : t_1 \leq t_2 < t : \neg\exists e : (E_3, \beta, e)(t_2))) \ , \\
(7) & (A^*(E_1, E_2, E_3), \beta, (e_1; e'_1; \ldots; e'_k; e_3))(t) \ :\Leftrightarrow \ (E_3, \beta, e_3)(t) \wedge \\
 & \quad \exists t_1 \leq t : (E_1, \beta, e_1)(t_1) \wedge \exists t'_1, \ldots, t'_k : t < t'_1 \leq \ldots \leq t'_k < t \wedge (E_2, \beta, e'_i)(t'_i), \\
 & \quad (A^*(E_1, E_2, E_3), \beta, (e_1; e_3))(t) \ :\Leftrightarrow \\
 & \quad (E_3, \beta, e_3)(t) \wedge \exists t_1 \leq t : (E_1, \beta, e_1)(t_1) \wedge \forall t_2 : t_1 \leq t_2 < t : \neg\exists e : (E_2, \beta, e)(t_2), \\
(8) & (E \rightarrow \mathit{Var}, \beta)(t) \ :\Leftrightarrow \ (E, \beta, e)(t) \wedge \beta(\mathit{Var}) = e.
\end{array}
$$

Table 5.2: Formal Semantics of the SNOOP Event Algebra with Variables

5.3.5 XML Markup for Composite Events – Example: SNOOP

The <eca:Event> element contains either an atomic event specification, or a subtree according to an event algebra language. Every subexpression is associated by its namespace with the appropriate language – i.e., an event algebra or an AESL. Below, we give an XML markup for a language based on SNOOP [25], extended with logical variables as a reference language for composite events that has been implemented in the prototype. Its handling of leaves (atomic events, opaque specifications, or embedded subtrees of another CEL) follows and illustrates the design decisions discussed in Section 5.1. In general, if an event algebra supports an XML markup, it will define its own ways for dealing with atomic events and variables.

SNOOP XML Markup. Snoop contains the usual simple combinators "and", "or", and "sequence"; additionally "any n out of *alternatives*", different kinds of periodic and aperiodic, optionally cumulative events. Operands can also be atomic event specifications, opaque specifications, or embedded event specifications in another language. The XML markup follows the term tree structure. The SNOOP DTD can be found in Appendix C.2.2.

Wrapper Elements and Language Identification for Nested Subexpressions. Leaves of the SNOOP tree are (i) AESs, (ii) opaque expressions, or (iii) nested subexpressions in other languages. For (ii) opaque elements, the same markup is used –now snoopy:Opaque– as for eca:Opaque. For (i) and (iii), the markup explicitly indicates the language border by the different namespaces. As discussed for the design decisions in Section 5.1, *wrapper* elements must be used in the latter case if the subexpression has any snoopy: properties:

- Sometimes snoopy:language or snoopy:domain for AESL subexpressions are needed.
- snoopy:bind-to-variable binds the result of evaluating a subtree to a variable.

In these cases, a wrapper element snoopy:Atomic or snoopy:External is used. If there are no such snoopy: properties, the wrapper element can optionally be omitted (since the language border is implicit in the namespace).

Variable Bindings in Algebra Expressions. Variables can be bound to literal values or subtrees as discussed above in Sections 5.3.2 (atomic events) and 5.3.4. Furthermore, variables

can be bound to algebraic subexpressions. Our SNOOP-style proposal uses the same mechanisms as presented in the ECA-ML language (but now in the snoopy namespace).

Example 16 *Consider the following event specification that should react if a travel:CanceledFlight event occurs after a travel:DelayedFlight for the same flight number (by using a join variable nr):*

```
<eca:Rule...>
  <eca:Event xmlns:travelns="http://www.semwebtech.org/domains/2006/travel#">
    <snoopy:Sequence>
      <snoopy:Atomic snoopy:language="http://www.semwebtech.org/languages/2006/aes-xmlql#">
        <travel:DelayedFlight travel:flight="{$id}" />
      </snoopy:Atomic>
      <snoopy:Atomic snoopy:language="http://www.semwebtech.org/languages/2006/aes-xmlql#">
        <travel:CanceledFlight travel:flight="{$id}" />
      </snoopy:Atomic>
    </snoopy:Sequence>
  </eca:Event>
    :
</eca:Rule>
```

The atomic events specifications include the necessary information about the used formalism (by the snoopy:language attribute) and the domain (implicit by namespace). The (non-snoop) contents is then sent to the XML-QL-style AEM, while optional snoop contents is considered to be evaluated locally.
Assume now that for every such pair, the complete second event, i.e., the complete structure

```
<travel:CanceledFlight travel:flight="{$id}">
  <travel:reason>bad weather<travel:reason>
</travel:CanceledFlight>
```

should be bound to a variable X. This is done by adding a snoopy:bind-to-variable subelement to the snoopy:Atomic element:

```
<eca:Rule...>
  <eca:Event xmlns:travelns="http://www.semwebtech.org/domains/2006/travel#">
    <snoopy:Sequence>
      <snoopy:Atomic snoopy:language="http://www.semwebtech.org/languages/2006/aes-xmlql#">
        <travel:DelayedFlight travel:flight="{$id}" />
      </snoopy:Atomic>
      <snoopy:Atomic snoopy:language="http://www.semwebtech.org/languages/2006/aes-xmlql#">
        <snoopy:bind-to-variable snoopy:name="X" />
        <travel:CanceledFlight travel:flight="{$id}" />
      </snoopy:Atomic>
    </snoopy:Sequence>
  </eca:Event>
    :
</eca:Rule>
```

Example 17 *The cumulative event from Example 8 (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 xmlns:uni="...">
  <eca:uses-variable eca:name="Subj" />
  <eca:Event>
    <eca:bind-to-variable eca:name="regseq" />
    <snoopy:CumulativeAperiodic xmlns:snoopy="http://www.semwebtech.org/languages/2006/snoopy#">
      <snoopy:Atomic snoopy:language="http://www.semwebtech.org/languages/2006/aes-xmlql#">
        <uni:reg_open uni:subject="{$Subj}" />
      </snoopy:Atomic>
      <snoopy:Atomic snoopy:language="http://www.semwebtech.org/languages/2006/aes-xmlql#">
        <uni:register uni:subject="{$Subj}" />
      </snoopy:Atomic>
      <snoopy:Atomic snoopy:language="http://www.semwebtech.org/languages/2006/aes-xmlql#">
        <uni:reg_close uni:subject="{$Subj}" />
      </snoopy:Atomic>
    </snoopy:CumulativeAperiodic>
  </eca:Event>
    :
</eca:Rule>
```

5.3.6 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) [14]. An rCML expression can be embedded in a SNOOP expression as follows:*

```
<eca:Rule ... >
  <eca:Event xmlns:snoopy="http://www.semwebtech.org/languages/2006/snoopy#">
    <snoopy:CumulativeAperiodic>
      <snoopy:Or> ...</snoopy:Or>
      <rcml:times xmlns:rcml="..."> ... </rcml:times>
      <snoopy:Sequence> ...</snoopy:Sequence>
    </snoopy:CumulativeAperiodic>
  </eca:Event>
  :
</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 4.2.3 to the SNOOP engine.

Above, the language border was implicit. In case that the rCML event should be bound to a variable on the SNOOP level, a snoopy:External wrapper is needed that contains the variable binding directive, and the rCML subevent:

```
<eca:Rule ... >
```

```
<eca:Event xmlns:snoopy="http://www.semwebtech.org/languages/2006/snoopy#">
  <snoopy:CumulativeAperiodic>
    <snoopy:Or> ...</snoopy:Or>
    <snoopy:External>
      <snoopy:bind-to-variable snoopy:name="regseq" />
      <rcml:times xmlns:rcml="..."> ... </rcml:times>
    </snoopy:External>
    <snoopy:Sequence> ...</snoopy:Sequence>
  </snoopy:CumulativeAperiodic>
</eca:Event>
:
</eca:Rule>
```

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.

5.3.7 Embedding Atomic Event Specifications in Composite Events and Rules

For the surrounding language (which can be an event algebra or 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 5.3.2, this was always the domain namespace, but taking e.g. an OWLQ AES, the outer expression is 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 AES. 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 on RDF. In an RDF setting, the administrative information about the AES is as follows:

(*aes*, rdf:type, mars:AtomicEventSpec)
(*aes*, rdf:type, *formalism*:AtomicEventSpec)
(*aes*, mars:domain, *domain*) (multiple domains allowed)
(*aes*, mars:language, *formalism*).

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

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

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

Example 19 (Explicit Namespace Borders) *A specification of an event that reacts upon a DelayedFlight event and extracts some data can be embedded as follows:*

```
<eca:Rule>
  <eca:Event>
```

```
      :
      <snoopy:Atomic
          snoopy:language="http://www.semwebtech.org/languages/2006/aes-xmlql#"
          snoopy:domain="http://www.semwebtech.org/domains/2006/travel#">
        <travel:DelayedFlight xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
          travel:flight="{$flight}" travel:time="{$time}" />
      </snoopy:Atomic>
      :
  </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:AtomicEventSpec
          xmlns:aes-xmlql="http://www.semwebtech.org/languages/2006/aes-xmlql#"
          aes-xmlql:domain="http://www.semwebtech.org/domains/2006/travel#">
        <travel:DelayedFlight xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
          travel:flight="{$flight}" travel:time="{$time}" />
      </aes-xmlql:AtomicEventSpec>
      :
  </eca:Event>
    :
</eca:Rule>
```

No Explicit Element. Explicit wrapper elements are often not necessary since there is a namespace change, and the domain information can be given as an attribute in the subelement. We recommend the designers of such languages to support such explicit elements as *optional* markup.

```
<eca:Rule>
  <eca:Event>
    <aes-xmlql:AtomicEventSpec
        xmlns:aes-xmlql="http://www.semwebtech.org/languages/2006/aes-xmlql#">
      <travel:DelayedFlight xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
          travel:flight="{$flight}" travel:time="{$time}" />
    </aes-xmlql:AtomicEventSpec>
  </eca:Event>
    :
</eca:Rule>
```

5.4 General Communication Patterns

A language becomes known to an ECA engine by being referenced in some rule (with its namespace URI). From that URI, using the Languages and Services Ontology, actual services that implement this language can be obtained (cf. Section 9.5.1).

For the actual 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 a composite event specification) but later no, one or more "answers" referring to the request will be sent,
- 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 MARS Framework is flexible wrt. the actual URLs where the services expect the requests. A service can e.g. receive requests for all tasks 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 9.5.1.

Communication Protocols. The communication is implemented by HTTP.

Message Contents.

- component expressions (i.e., composite events, atomic event specifications, queries, actions): in XML representation as elements in the respective namespaces,
- answers to requests: as <logvars:answers> or <logvars:answer> XML elements (as discussed above in Section 4.2).

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	the answer will refer to this subject (in case of asynchronous communication),
content(1)	the component or a reference to it,
content(2)	optionally: input variable bindings (cf. Section 4.2.2).

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:

Answers:	
Subject	the Subject set by the requester for identifying the answer,
Sender, Reply-To	empty, optional,
content(1)	in the <logvars:answers> format described in Section 4.2.3,
content(2)	optional "comments" [up to now, no elements are specified for this – the model can be extended here]

Communication of (Atomic) Events and Actions. As already discussed, events and actions are XML fragments:

- Reply-To and Subject: none
- content: event or action 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) or as elements in the body. 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 9.5.)
First examples will be given in Section 5.5.1.

5.5 Architecture and Communication: ECA, CED, and AEM

Event processing is done in cooperation of an ECA engine, one or more *Composite Event Detection Engines (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 5.2:

- The ECA engine registers the event component at an appropriate CED (composite component) [or AEM (only atomic event specification)] service,
- the CED registers the AESs at appropriate AEMs that implement the AESLs,
- the AEMs register themselves for the relevant event types at the Event Brokers (details will be described in Section 8.2)
- The domain nodes or event brokers forward relevant atomic events to the AEMs in the format given in Section 5.3.1,
- the AEMs match them against the specification and inform the CEDs,
- the CEDs process them and inform the ECA engine in case that a composite event has been detected.

5.5.1 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. Then all communication ECA→CED, ECA→AEM, CED→CED (nested) and CED→AEM follows the patterns described in Section 5.4:

- Downwards: Reply-To, Subject/identifier of request, fragment, optional: variable bindings (depends whether the service can handle multiple bindings, see Section 9.5).
- Upwards: Subject/identifier, functional result (matched event (sequence)) with optional tuples of variable bindings.

The responses of the event component services match the format for variable bindings given in Section 4.2.3. Moreover, the answer from detecting an atomic event has the same format as when detecting a composite event for the following reasons:

- conceptual cleanliness: both are "detected events",

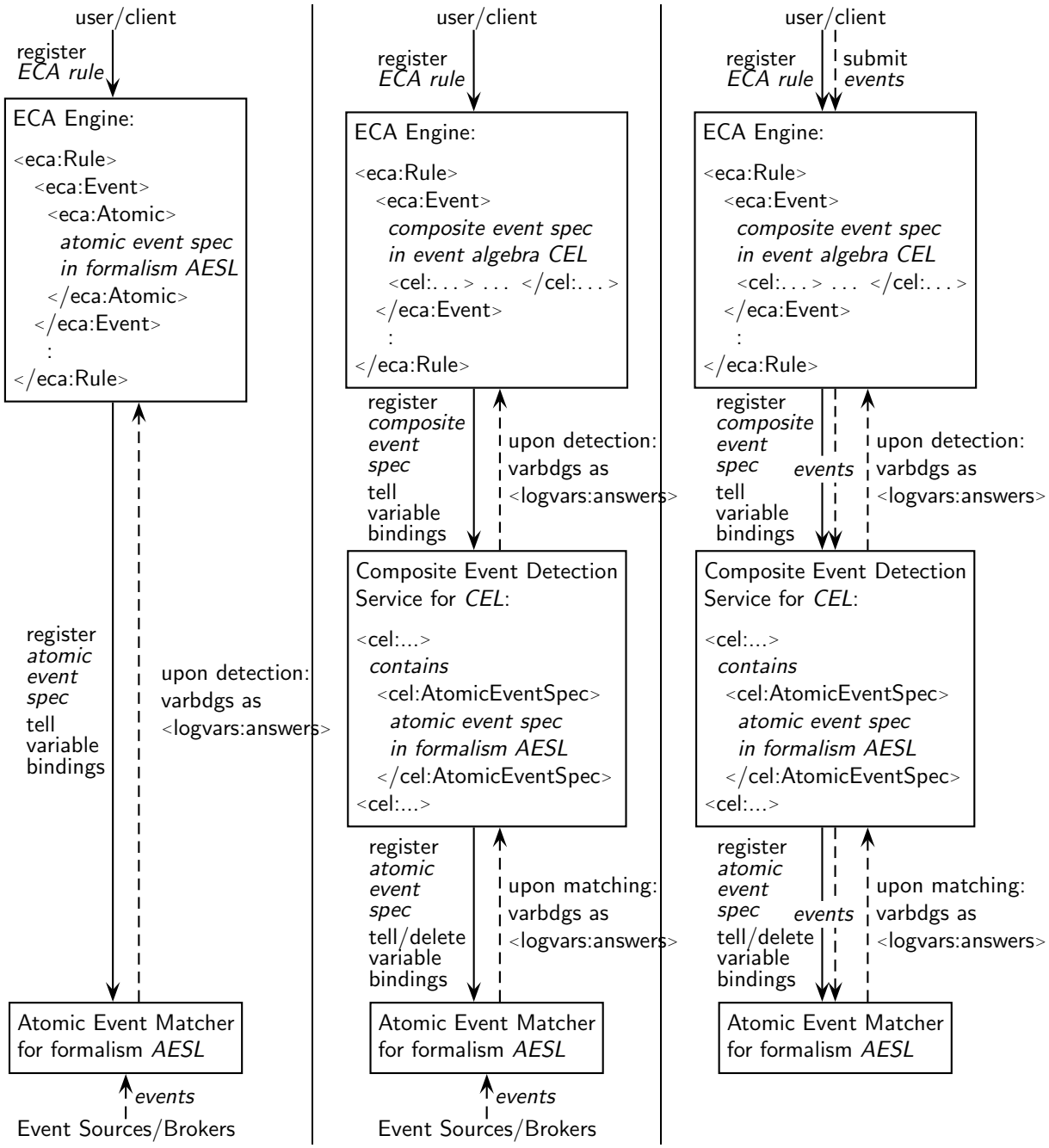


Figure 5.2: Architecture: Processing Event Components and Events (left and middle: independent; right:dependent)

- architecture: the event component of a rule may be either a composite event or an atomic one. 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. Additionally, event algebras are extended with using free logical variables with join semantics. 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).

5.5.2 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 in 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 9 – the result is a URL where the request has to be sent to and some minor directives how it should be sent 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>URI of the service that expects the answer </Reply-To>
  <Subject> identifier of the request </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/languages/2006/logic#">
    variable bindings
  </logvars:variable-bindings>
</register>
```

The upwards communication is even simpler: the result is sent to the appropriate task of the service originally given as Reply-To (cf. Chapter 9 for details). The agreed subject is used, and the <answers> or a single <answer> element is submitted:

```
to: URL of the appropriate task of the service identified by the
    Reply-To-address in the request that is answered
<anyname>
  <Subject>identifier of the request</Subject>
  <!-- the answer bindings -->
  <logvars:answers xmlns:logvars="http://www.semwebtech.org/languages/2006/logic#">
    answers
  </logvars:answers>
</anyname>
```

5.5.3 Communication for Event Matching with AEMs

Similar to the ECA-CED communication, the ECA-AEM/CED-AEM communication 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 4.2.3). 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 8.2). The AEM mathes each potentially relevant event against the registeres AESs 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 AEM 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 a CES at a CED:

```
to: service-URL of the AEM
<register>
  <Reply-To>URI of the service that expects the answer </Reply-To>
  <Subject> identifier of the request </Subject>
  <!-- next the AES -->
  the AES fragment
  <!-- and optionally the existing variable bindings -->
  <logvars:variable-bindings xmlns:logvars="http://www.semwebtech.org/languages/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 4.2.3 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 20 *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:DelayedFlight travel:flight="LH1234" travel:minutes="30" />
    <travel:DelayedFlight travel:flight="AF0815" travel:minutes="90" />
      :
    <travel:CanceledFlight travel:flight="AL4711" />
    :
  </msg:content>
</msg:receive-message>
```

Consider the following rule at an ECA engine:

```
<eca:Rule>
  <eca:Event>
    <eca:Atomic eca:language="http://www.semwebtech.org/languages/2006/aes-xmlql#">
      <travel:DelayedFlight travel:flight="{$flight}" travel:minutes="{$minutes}" />
    </eca:Atomic>
  </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 5.3.

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:DelayedFlight travel:flight="LH1234" travel: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:DelayedFlight travel:flight="LH1234" travel: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 and rule evaluation, variable bindings obtained by "earlier" atomic events can be used for constraining the relevant event occurrences.

Example 21 (Join Variables in Composite Events) *Consider again Example 17 which uses a cumulative SNOOP event with a join variable $Subj:*

```
<snoopy:CumulativeAperiodic
    xmlns:snoopy="http://www.semwebtech.org/languages/2006/snoopy#"
```

```
    xmlns:uni="..." >
  <aes-xmlql:AtomicEventSpec
      xmlns:aes-xmlql="http://www.semwebtech.org/languages/2006/aes-xmlql#" >
    <uni:reg_open subject="{$Subj}" />
  </aes-xmlql:AtomicEventSpec>
  <aes-xmlql:AtomicEventSpec
      xmlns:aes-xmlql="http://www.semwebtech.org/languages/2006/aes-xmlql#" >
    <uni:register subject="{$Subj}" />
  </aes-xmlql:AtomicEventSpec>
  <aes-xmlql:AtomicEventSpec
      xmlns:aes-xmlql="http://www.semwebtech.org/languages/2006/aes-xmlql#" >
    <uni:reg_close subject="{$Subj}" />
  </aes-xmlql:AtomicEventSpec>
</snoopy:CumulativeAperiodic>
```

Here, when the event component is registered at SNOOP, SNOOP 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.)*

5.5.4 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 Registry, see Section 9.5.1). 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.

- If the element is an <Opaque language="*language*" > element, then identify the responsible service.

- 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.

5.5.4.1 Example

Example 22 *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 from 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:Or xmlns:snoopy="http://www.semwebtech.org/languages/2006/snoopy#">
      <snoopy:Atomic snoopy:language="xml-ql-match">
        <msg:receive-message msg:sender="$myTravelAgent">
          <msg:content>
            <travel:delayed-train travel:train="$myTrain" travel:arrivalTime="$newArrival"/>
          </msg:content>
        </msg:receive-message>
      </snoopy:Atomic>
      <snoopy:Atomic>
        <aes-xpath:Event xmlns:aes-xpath="http://www.semwebtech.org/languages/2006/aes-xpath#">
          <aes-xpath:Test cond="$event/name()='travel:DelayedTrain'"/>
          <aes-xpath:Test cond="$event/@train=$myTrain"/>
          <aes-xpath:Variable name="newArrival" select="$event/@arrivalTime"/>
        </aes-xpath:Event>
      </snoopy:Atomic>
    </snoopy:Or>
  </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: URI of the ECA engine*
- *Subject/task identifier: e.g.* the-rule-id/event
- *the whole event component, i.e., the* snoopy:Or *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 first AES, i.e., the* <msg:receive-message> *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 detection, 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 "*http://www.semwebtech.org/languages/2006/aes-xpath#*"; the service URL is looked up in the LSR. 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:DelayedTrain *(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 4.2.3).
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 disjunctive 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 23 *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 uni:subject= "Databases" />
    <uni:register uni:subject= "Databases" uni:name= "John Lennon" />
    <uni:register uni:subject= "Databases" uni:name= "Paul McCartney" />
    <uni:register uni:subject= "Databases" uni:name= "George Harrison" />
    <uni:register uni:subject= "Databases" uni:name= "Ringo Starr" />
    <uni:reg_close uni: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 bound to 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.

5.6 The Query Component

There are a lot of query languages for XML or RDF data around, such as XQuery (where an XML markup proposal has been presented in [73]), 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 of a certain database node can use the query language of this node, often in an opaque way.
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 4.2.3.

In all cases, the result of evaluating queries is communicated in the usual form described above.

5.6.1 Opaque Queries

For the development of a first XML level prototype of the MARS Framework we rely on *opaque* queries; see Section 10.4.

5.6.2 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., in the same style as the relational algebra.

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).

5.6.3 Composite Queries

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 [42, 43] in the Florid system [32] or with some adaptations for XPathLog [50, 51] in the LoPiX system [49].

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 [46, Section 2.2.1]. The (preliminary) DTD is as follows:

```
<!ENTITY % operand "(union | intersect | difference |
    select | project | join | navigation |
    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 language), 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. The markup of queries should support *reasoning* about queries and rules on the RDF level. On the RDF level, OWLQ will provide such a semantics (cf. Section 13.3).

5.7 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. Non-atomic tests can be expressed as formulas e.g. in First-Order logic (where an XML markup is given in [17]).

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. Often, parts of the test 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.

Services in the test component can implement two semantics:

- functional semantics: tuples of variable bindings are communicated downwards, and the test service labels each tuple with true/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. The latter case has the disadvantage that when asynchronous answering is used, "failing" a test never becomes explicit.

5.7.1 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 "( and | or | not |  ANY-from-other-namespace | opaque)">
<!ELEMENT and (%operand;, %operand;)>
<!ELEMENT or (%operand;, %operand;)>
<!ELEMENT not (%operand;)>
```

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 a union (care for same set of bound variables), and not is actually a set difference. Quantifiers are not allowed.

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 24 *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>
      <fn:greater-than>
        <variable>Y</variable>
        100
      </fn:greater-than>
    </all>
    <some>
      <fn:greater-than>
        <variable>Y</variable>
        1000
      </fn:greater-than>
    </some>
  </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.

5.7.2 Atomic Tests: Predicates

A language for atomic tests supports built-in predicates that are evaluated locally. Test languages may e.g. differ in the supported datatypes. As a base, e.g. the comparison predicates of XPath can be used (as will also be done in OWLQ for tests in Section 13.4.4).

Example 25 *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. The rule can be expressed as follows (e.g., using XPath):

```
<eca:Rule xmlns:travel="http://www.semwebtech.org/domains/2006/travel#">
  <eca:Event>
    <eca:Atomic eca:language="xml-ql-match">
      <travel:Booking travel:flight="{$Flight}" travel:name="{$P}" travel:to="{$City}" />
    </eca:Atomic>
  </eca:Event>
  <eca:Query
    <eca:Opaque eca:language= "XPath">
      <eca:bind-to-variable eca:name="OwnCar" />
      <eca:uses-variable eca:name="P" />
      /customers/customer[@name="$P"]/owncar/@type
    </eca:Opaque>
  </eca:Query>
  <eca:Query>
    <eca:Opaque eca:language= "XPath">
      <eca:bind-to-variable eca:name="OwnClass" />
      <eca:uses-variable eca:name="OwnCar" />
      /types/type[@name="$OwnCar"]/@class
    </eca:Opaque>
  </eca:Query>
  <eca:Query>
    <eca:Opaque eca:language= "XPath">
      <eca:bind-to-variable eca:name="MaxPrice" />
      <eca:uses-variable eca:name="P" />
      /customers/customer[@name="$P"]/@maxprice
    </eca:Opaque>
  </eca:Query>
  <eca:Query>
    <eca:Opaque eca:language= "XPathLog">
      <eca:uses-variable eca:name="City" />
      /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/mars/2006/test#"
      xmlns:xpath="http://www.w3.org/XPath">
    <test:and>
      <xpath:equals><test:variable>OwnClass</test:variable>
          <test:variable>Class</test:variable></xpath:equals>
      <xpath:lessThanOrEqual><test:variable>Price</test:variable>
          <test:variable>MaxPrice</test:variable></xpath:lessThanOrEqual>
    </test:and>
  </eca:Test>
  <eca:Action>
    <!-- book the most expensive one -->
  </eca:Action>
</eca:Rule>
```

5.8 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 a *Composite Action Exection Engine (CAE)*. For executing actions, two scenarios can be distinguished:

- Rules can implement *global* behavior (general dynamic integrity constraints like policies in an application domain). Here, it is *not* obvious which nodes should actually execute the action. This is solved by the domain brokers that contact the respective domain nodes; see Sections 8.5.
- 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 that case, the action component can be extended with an attribute eca:execute-at whose value can be either a pseudo-variable "$owner" (the owner of the rule) or a hard-coded 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.

5.8.1 Atomic Actions

Leaf expressions of composite actions can be the following:

- actions from a domain ontology (see Chapter 7),
- nested actions in another CAL,
- opaque actions that invoke arbitrary Web Services (by HTTP or SOAP messages),
- implementation of ECE rules: their action component is usually atomic and just raises an event in an application domain.

Invoking Actions at MARS-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 4.2.2.

The request is sent to an action broker (cf. Section 8.5) 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 26 (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 executed at different application nodes (that can in an RDF world be determined from the association between flight numbers and the corresponding airlines). The action specification looks as follows:

```
<eca:Action>
  <eca:has-input-variable eca:name="flight" />
  <eca:has-input-variable eca:name="reason" />
  <travel:CancelFlight travel:flight="$flight">
    <travel:reason>$reason</travel:reason>
  </travel:CancelFlight>
</eca:Action>
```

The contents of the action component is in the travel namespace which is not a language namespace (like a process language, e.g., CCS), but an application domain. Thus, a domain broker for travel is responsible for processing it. The markup also states that the values of the variables flight and reason must be communicated.

The content of the message sent to the domain broker (wrapped by a <request> element, see Section 9.5 for details about the specification of the communication formats) contains then the (atomic) action pattern together with the variable bindings:

```
<request>
  <travel:CancelFlight travel:flight=$flight>
    <travel:reason>$reason</travel:reason>
  </travel:CancelFlight>
  <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>
```

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 eca:uri="domain-service-url">
        <eca:has-input-variable eca:name="flight" />
        <eca:has-input-variable eca:name="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:has-input-variable eca:name="flight" />
      <eca:has-input-variable eca:name="reason" />
      INSERT INTO cancellations VALUES($flight, $reason)
    </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 eca:language="XQuery+Updates">
      <eca:has-input-variable eca:name="flight" />
      <eca:has-input-variable eca:name="reason" />
      do insert <cancel>$reason</cancel>
      as last into doc("url")/flights/flight[@nr=$flight]
    </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 MARS-aware interface and iterates over the bindings.

Raising Events. ECE rules can be implemented by ECA rules whose action consists of raising an event. For that, MARS provides a built-in action mars:raise-event that raises events in the XML format given in Section 5.3.1 (cf. Section 10.5.3 for a detailed description of its use and syntax).

5.8.2 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, [35]), or as an *operational semantics* by specifying the behavior of every element of the algebra (e.g., CCS – Calculus of Communicating Systems, [55, 56]). 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 is intended. 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 [34] and Hennessy-Milner-Logic [57].
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-from-other-namespace>
```

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* [55, 56]), *CSP – Communicating Sequential Processes* [35]), ACP [13], COSY [36], or the *Box Algebra* [16] 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 following patterns are e.g. available:

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 [55] 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 [35] 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.

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

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

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 27 (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):*

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

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

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

  (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 28 *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 27). 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 29 (Processes with Conditions)

1. *Consider again the scenario from Example 28, 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 27 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 5.3 shows the relationship between the process algebra language and the contributions of the domain languages and the event and test component languages.

XML Markup for the Action Component. According to the above considerations, processes 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) embedded expressions of event languages, conditions, and 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 5.8.5.

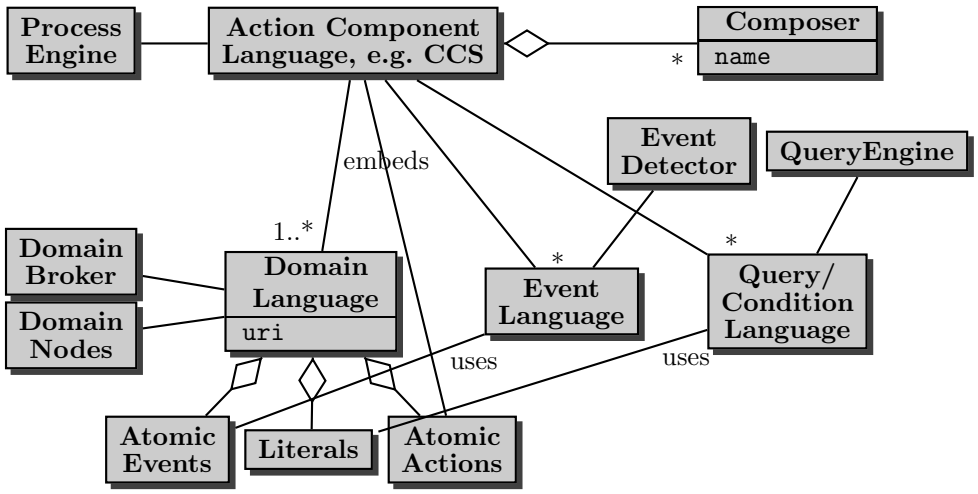


Figure 5.3: Structure of the Action Component as an Algebraic Language using CCS

5.8.3 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. Optionally, ccs:AtomicAction is allowed. 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:BookFlight travel:flight="{$flight}" travel:date="{$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:event-lang="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:cond-lang="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 {ccs:url="*node-url*" | ccs: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).

5.8.4 CCS DTD

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 | ContinueFixpoint |
   Action | Event | Query | Test | AtomicAction | External | 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
     fix-variables IDREFS #REQUIRED
     has-index NMTOKEN #REQUIRED
     local-variables NMTOKENS #IMPLIED>
<!ELEMENT ContinueFixpoint EMPTY>
  <!ATTLIST ContinueFixpoint
     with-variable IDREF #REQUIRED>
<!ELEMENT AtomicAction ANY-from-other-namespace>
<!ELEMENT Event ANY-from-other-namespace>
<!ELEMENT Query ANY-from-other-namespace>
<!ELEMENT Test ANY-from-other-namespace>
<!ELEMENT External ANY-from-other-namespace>
<!ELEMENT Opaque ANY>
  <!ATTLIST Opaque
     language CDATA #IMPLIED
     url CDATA #IMPLIED>
<!-- variable declaration stuff as in ECA-ML -->
```

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:Sequence mode="*mode*">*contents*</ccs:Sequence> 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:Alternative>*contents*</ccs:Alternative> stands for "$\sum$" and "+" (alternatives),
- <ccs:Concurrent mode="*mode*">*contents*</ccs:Concurrent> represents "$\times$" and "|" (parallel),

- The <ccs:Event xmlns:*lang*="*uri*">, <ccs:Query xmlns:*lang*="*uri*">, <ccs:Test xmlns:*lang*="*uri*">, and (optional) <ccs:External xmlns:*lang*="*uri*"> and <ccs:AtomicAction 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 Local Variables in Fixpoint Processes. For integration with the MARS 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 fix-variables="$var_1 \ldots var_n$" has-index="j"
  local-variables="*list of variables*">
  contents, usually containing the process variables
  </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 30 (Variables in Fixpoint Processes) *Consider again Example 29(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.*

5.8.5 Examples in CCS

Example 31 (Basic 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/languages/2006/eca-ml#">
  <eca:Event xmlns:snoopy="http://www.semwebtech.org/languages/2006/snoopy#">
    <snoopy:Sequence>
      <travel:DelayedFlight travel:flight="{$flight}" travel:date="{$date}" />
      <travel:CanceledFlight travel:flight="{$flight}" travel:date="{$date}" />
    </snoopy:Sequence>
  </eca:Event>
  <eca:Query>
    <eca:Opaque eca:language="sparql" eca:domain="travel">
      SELECT $booking, $name WHERE {
      $x rdf:type travel:flight .
      $x travel:flight $flight . $x travel:date $date .
      $x travel:booking $booking .
      $booking travel:name $name }
    </eca:Opaque>
  </eca:Query>
  <eca:Action xmlns:ccs="http://www.semwebtech.org/languages/2006/ccs#">
    <ccs:Concurrent>
      <ccs:AtomicAction>
        <travel:reserve-room travel:hotel="hotel uri" travel:name="$name" />
      </ccs:AtomicAction>
      <ccs:Sequence>
        <ccs:Test >
          <eca:Opaque eca:language="sparql" eca:domain="travel">
            ($booking travel:class travel:business-class)
          </eca:Opaque>
```

```
        </ccs:Test>
        <ccs:Query>
          <eca:Opaque eca:language="sparql" eca:domain="travel">
            SELECT $phone WHERE
            ($booking  travel:contact-phone  $phone)
          </eca:Opaque>
        </ccs:Query>
        <ccs:AtomicAction>
          <comm:send-sms to= "$phone">
            we are very sorry ... and booked a room in ... for you
          </comm:send-sms>
        </ccs:AtomicAction>
      </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.*

Example 32 *Consider again the situation from Example 10 with the following action specification:*

```
<eca:Rule xmlns:eca="http://www.semwebtech.org/languages/2006/eca-ml#">
    :
  Variable Flight contains the flight number,
  Variable Customers contains Customer elements that in turn contain the customer names.
    :
  <eca:Action>
    <eca:has-input-variable eca:name="Customers" />
    <eca:has-input-variable eca:name="Flight" />
    send one message with all Customers/customer/name to the hotel,
    and
    for each N in Customers/customer/@phonenr do
      notify_cancellation(Flight, sms:N)
  </eca:Action>
</eca:Rule>
```

The action part can be expressed in CCS extended with queries as "send the message to the hotel, and in parallel, evaluate a query that binds an additional variable customer (resulting in multiple tuples of bindings) and then send the individual SMSs":

```
<eca:Rule xmlns:eca="http://www.semwebtech.org/languages/2006/eca-ml#">
    :
  Variable Flight contains the flight number,
  Variable Customers contains Customer elements that in turn contain the customer names.
    :
  <eca:Action>
    <eca:has-input-variable eca:name="Customers" />
    <eca:has-input-variable eca:name="Flight" />
    <ccs:Concurrent>
```

```
      <smtp:send-message>
        <smtp:to>the hotel's e-mail</smtp:to>
        <smtp:body> names of customers <smtp:body>
        </smtp:send-message>
        <ccs:Sequence>
          <ccs:initialize-variable name="customer" language="xpath" select="$Customers/customer" />
          <ccs:initialize-variable name="phone" language="xpath" select="string($customer/@phonenr)" />
        <sms:send-message>
        <sms:to>{$phone}</sms:to>
        <sms:body>your flight {$Flight} has been cancelled ... </sms:body>
      </sms:send-message>
    </ccs:Concurrent>
  </eca:Action>
</eca:Rule>
```

Example 33 (CCS with a Fixpoint) *Consider again the scenario of a university from Example 29 whose action component is expressed in CCS was as follows:*

$\mathrm{fix}X.$(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)

The XML Markup of the rule is then as follows:

```
<eca:Rule xmlns:uni="http://www.education.de">
  <eca:Event> failed twice – binds  $student ID and $course  </eca:Event>
  <eca:Query> binds e-mail addresses of the student and the lecturer </eca:Query>
  <eca:Action xmlns:ccs="...">
    <ccs:Sequence>
      <ccs:Fixpoint ccs:fix-variables="X" ccs:has-index="1" ccs:local-variables="$date $time $room">
        <ccs:Sequence>
          <ccs:AtomicAction> send asking mail to lecturer </ccs:AtomicAction>
          <ccs:Event> answer binds $date and $time</ccs:Event>
          <ccs:Query> any room $room at $date $time available? </ccs:Query>
          <ccs:Alternative>
            <ccs:Test> yes </ccs:Test>
            <ccs:Sequence>
              <ccs:Test> no </ccs:Test>
              <ccs:ContinueFixpoint ccs:withVariable="X" /> <!-- the CCS fix variable -->
            </ccs:Sequence>
          </ccs:Alternative>
        </ccs:Sequence>
      </ccs:Fixpoint>
      <ccs:AtomicAction> send message ($date, $time, $room) to student </ccs:AtomicAction>
      <ccs:AtomicAction> send message ($date, $time, $room) to lecturer </ccs:AtomicAction>
    </ccs:Sequence>
  </eca:Action>
</eca:Rule>
```

5.8.6 Grouping

As all variable bindings are atomic, they are usually processed individually. Sometimes, aggregations over the tuples, probably grouped by some variables, are useful.

Example 34 *Consider again Example 8 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* → *'Databases'*, *regseq* → *(as above)*, *Student* → *'John Doe'*$\}$,
   $\beta_2 = \{$*Subj* → *'Databases'*, *regseq* → *(as above)*, *Student* → *'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* → *'Databases'*, *Student* → $\{$*'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,*

$$\beta = \{\ \{\quad \begin{array}{ll} \textit{Subj} \rightarrow \textit{'Databases'}, & \textit{regseq} \rightarrow \textit{(as above)}, \\ \textit{Students} \rightarrow \langle\textit{students}\rangle & \langle\textit{name}\rangle\textit{John Doe}\langle\textit{/name}\rangle \\ & \langle\textit{name}\rangle\textit{Scott Tiger}\langle\textit{/name}\rangle\ \langle\textit{/students}\rangle\} \end{array}$$

   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.*

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). Grouping is indicated by an optional attribute

group-by="*variable list*" .

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.

5.8.7 Processing

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 Chapter 8) 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 9.5.

5.8.8 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 MARS 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.

5.9 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 9. 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 6

Domain Ontologies in the MARS Framework

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 35 *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.*

Currently, (domain) ontologies are usually described in OWL. Most people currently consider an ontology to be the description of the *static* concepts of the domain, but not of its dynamics. The situation is comparable to UML: the static part is the class diagram (which, in contrast to OWL ontologies, at least also describes the signature of the methods), while the dynamic notions are described by other diagrams.

In the same way, ontology descriptions in the MARS Framework have to be extended to cover the behavioral notions of events and actions; preferably also the semantics of actions in terms of changes and raised events must be described to allow for *reasoning* in such an ontology.
In the following sections, we describe:

- generic ontology issues of domains,
- generic interfaces and functionality of domain nodes (Chapter 7); architecture samples are given in Chapter 10,
- communication and *domain brokering* issues (Chapter 8).

6.1 Domain Ontologies

As discussed in Section 2.2, a domain ontology is partitioned into static notions, named actions, and named events, as shown in Figure 6.1. 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).
In the MARS 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.

Example 36 (Definition of the Travel Ontology) *The travel domain ontology provides –among others– the following classes, properties, events and actions:*

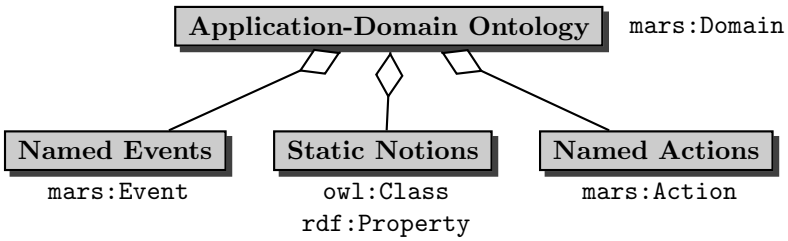


Figure 6.1: Components of Domain Ontologies

```
<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:mars="http://www.semwebtech.org/mars/2006/mars#">
         xml:base="http://www.semwebtech.org/domains/2006/travel">

 <mars:Domain rdf:about="#"/>
 <owl:Class rdf:ID="Airline">
  <rdfs:subClassOf rdf:resource="mars:DomainService"/>
 </owl:Class>
 <owl:Class rdf:ID="Car-Rental">
  <rdfs:subClassOf rdf:resource="mars:DomainService"/>
 </owl:Class>

 <owl:Class rdf:ID="Flight-Event">
  <rdfs:subClassOf rdf:resource="mars:Event"/>
 </owl:Class>
 <owl:Class rdf:ID="Flight-Action">
  <rdfs:subClassOf rdf:resource="mars:Action"/>
 </owl:Class>
 <owl:Class rdf:ID="Cancellation">
  <rdfs:subClassOf rdf:resource="mars:Event"/>
 </owl:Class>
 <owl:Class rdf:ID="CanceledFlight">
  <rdfs:subClassOf rdf:resource="#Cancellation"/>
  <rdfs:subClassOf rdf:resource="#Flight-Event"/>
 </owl:Class>

 <owl:Class rdf:ID="CancelFlight">
  <rdfs:subClassOf rdf:resource="#Flight-Action"/>
 </owl:Class>

 <owl:Class rdf:ID="DelayFlight">
  <rdfs:subClassOf rdf:resource="#Flight-Action"/>
 </owl:Class>

 <owl:Class rdf:ID="FullyBooked">
  <rdfs:subClassOf rdf:resource="#Flight-Event"/>
 </owl:Class>
</rdf:RDF>
```

With such an (event) class hierarchy, there could be a rule that informs about any cancellations of

any means of transportation.

6.2 Rules in Ontologies

Domain ontologies do not only consist of independent static notions, events, and actions, but also contain rule-based *definitions*:

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.

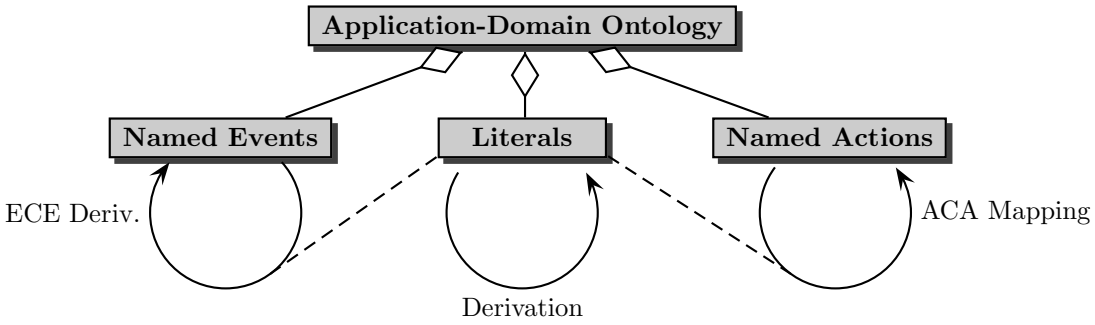


Figure 6.2: Derivation and Mapping Rules for Events, Literals and Actions

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.

6.2.1 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 (restricted),

- combination with external rule reasoners (F-Logic rules, SWRL support in Pellet).

The actual evaluation of such rules can be located in the domain broker and optionally also in the application nodes itself. Note that individual nodes can additionally have own local rules.

6.2.2 ECE Rules

ECE rules define derived, usually composite, events.

Example 37 (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.*

```
<mars:Definition mars:syntax= "xml">
  <mars:defined>
    <!-- pattern of the event to be derived -->
    <travel:HalfBooked travel:flight="$f" travel:date="$d" />
  </mars:defined>
  <mars:defined-as>
    <!-- E/C components how to derive it -->
    <eca:Event>
      <travel:Booking travel:flight="$f" travel:date="$d" />
    </eca:Event>
    <eca:Test>
      <eca:Opaque eca:language="xpath">
        <!-- note: XML schema of the (local) database assumed to be known -->
        count($flight[@date="$d"]/booking) = $flight/id(@aircraft)/@number-of-seats div 2
      </eca:Opaque>
    </eca:Test>
  </mars:defined-as>
</mars: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 mars:Event>.
Another application is to "extract" events from messages:

Example 38 *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:DelayedFlight travel:flight="LH1234" travel:minutes="30" />
    <travel:DelayedFlight travel:flight="AF0815" travel:minutes="90" />
    <travel:DelayedFlight travel:flight="CY42" travel:minutes="60" />
      :
    <travel:CanceledFlight travel:flight="AL4711" />
    :
  </msg:content>
</msg:receive-message>
```

An ECE rule considers the message as an event and raises the event that is contained in the message.

```
<eca:Rule>
  <eca:Event
    <eca:Atomic eca:language="xmlqlmatch"
```

```
      xmlns:travel="http://www.semwebtech.org/domains/2006/travel#" xmlns:mail="...." >
    <eca:bind-to-variable eca:name="e" />
    <mail:receive-message sender="service@fraport.com" >
      <mail:content>
        <travel:delayed travel:flight="$flight" travel:minutes="$minutes" />
      </mail:content>
    </mail:receive-message>
  </eca:Atomic>
 </eca:Event>
 </eca:Test eca:language="xpath" eca:select="$minutes > 25" />
 <eca:Action>
  <mars:raise-event xmlns:mars="http://www.semwebtech.org/mars/2006/mars#" >
    <travel:delayed travel:flight="$flight" travel:minutes="$minutes" />
  </mars:raise-event>
 </eca:Action>
</eca:Rule>
```

The implementation of derived events in course of event brokering will be discussed in Section 8.3.

6.2.3 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.

6.2.3.1 Definitory ACA Rules on the Ontology Level

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.

```
<mars:Definition syntax="xml" >
 <mars:defined>
  <banking:Money-Transfer amount="$amount" from="$from" to="$to" />
 </mars:defined>
 <mars:defined-as>
  <eca:Action>
   <ccs:Concurrent xmlns:ccs="http://www.semwebtech.org/languages/2006/ccs#" >
    <banking:debit amount="$amount" account="$from" />
    <banking:deposit amount="$amount" account="$to" />
   </ccs:Concurrent>
  </eca:Action>
 </mars:defined-as>
</mars: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 mars:Action>.

The implementation of composite actions in course of action brokering and ACA rules will be discussed in Section 8.5.2.

6.2.3.2 Implementational ACA Rules from the Ontology to the Data Model Level

Application Domain nodes receive action invocations that have to be carried out. Usually, such actions are implemented by classical Web Service calls: invoking an action at a node means

invoking a method of a Web Service that starts a Java (or perl or PHP or whatever) execution. The interface and implementation of such application nodes is then very specific to a certain domain.

In the MARS Framework, we target to support a *generic* architecture which is also continued at the domain node level: Atomic actions are represented by XML (or RDF fragments) that are sent to domain nodes that implement these actions. In Section 7.3.2, we show how such a generic domain node architecture supports ACA rules in this lowest abstraction level.

6.2.4 Discussion: Comparison with RuleML Proposal

RuleML proposes a rule markup as ruleml:imp, ruleml:head, ruleml:body for representing rules. We did not 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 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.

Chapter 7

Domain Application Nodes

Domain application nodes are the "leaves" of the Semantic Web architecture. They represent services in the application domain, e.g., airlines, train companies, car rentals, universities. As such, they usually have some kind of database or knowledge base. According to their task as services, they offer to execute actions (e.g., answering queries, booking a flight, or registering as a student). So far, they are closely related to common Web Services. When considering reactive behavior in the Semantic Web for the MARS Framework where *events* are also of central importance, domain nodes should also support this by *emitting* events according to the events defined by the domain ontology (e.g. that a flight is now fully booked). Nodes that do not emit events are "sinks" of behavior – which is often sufficient (e.g., for a car rental). Then, they may have local (reactive) behavior, but nothing reacts externally upon them – at least not directly; they still can be queried and by this participate in processes.

7.1 External Interface of Domain Nodes

7.1.1 Providing Static Data

Nodes serving as databases or knowledge bases can have a rather primitive functionality: providing data for lookup (that can be updated by bulk load, e.g., once per month).

Files. The most simple domain "node" is just a 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 MARS Framework, such queries are handled as opaque queries whose answer are e.g. sets of XML nodes. The ECA engine deals with this by generic wrappers for HTTP, XQuery, SPARQL etc. (cf. Section 10.4).

MARS-Aware Querying. In full exploitation of the MARS Framework, also the query component is stated in structured way by a query language in XML or RDF markup. The respective language proposal OWLQ is described in Section 13.3. The communication is then done using the communication format given in Section 4.1.4.

7.1.2 Executing Action Requests

Behavior for Web nodes means usually to be able to process actions. Full-fledged Semantic Web domain nodes support actions of the corresponding domain ontologies (i.e., things ?X such that (?X, rdf:type, mars:Action) holds) that are communicated according to the downward communication format given in Section 4.1.4, consisting of the action to be executed as an XML fragment, and optionally tuples of variable bindings.

For nodes that are not MARS-aware, actions 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). For such nodes, wrappers can be used for adapting them to the framework level.

The incoming (atomic) action requests must be mapped to updates on the local knowledge base locally (e.g., by ACA rules).

7.1.3 Reporting About Behavior: Providing Atomic Events

The current Web architecture does not consider "events" communicated by *push* communication. For the MARS Framework, nodes are expected to emit atomic events (i.e., things ?X such that (?X, rdf:type, mars:Event) holds). They are communicated in XML format as described in Section 5.3.1, 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 8.3.1).

In the MARS Framework, events are forwarded to domain brokers (as separate services) where filtering and communication to the outside takes place.

The actual emission of events upon changes in the local database must be provided by internal behavior of the node.

7.1.4 Summary

Thus, domain nodes provide a *generic*, application-domain-independent interface:

- receiving queries; answering them,
- receiving action requests, executing them,
- emitting events,
- administrative stuff: registering at domain brokers.

The actual architecture that satiesfies these requirements will be discussed next.

7.2 Domain Node Local Behavior

The internal behavior of nodes can be black-box behavior, implemented in any programming languages, or given by ACA, ECA and ECE rules (often by database-level triggers). The information flow between events and actions is depicted in Figure 7.1 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. ACA mapping: map high-level actions to lower-level (e.g. INSTEAD OF triggers).

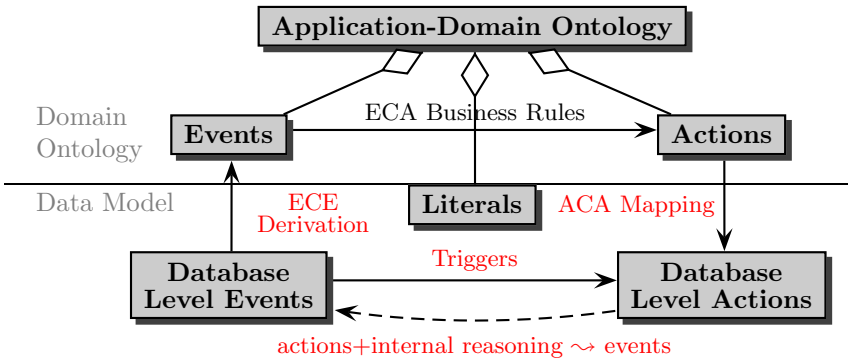


Figure 7.1: Interference of Events, Actions, and Literals

7.2.1 Data Model Triggers

As shown in Figure 2.3, *data model level triggers* inside of the database of a node form the lowest level of rules. Reacting directly to changes in the database, they provide the basic level of behavior. They serve for (i) updating the database, and (ii) raising events.

Triggers are simple rules on the *(database) programming language and data structure level*. They follow a simple ECA pattern where the "events" coincide directly with the update operations of the database, the conditions are given in the database query language and the action component is given in a simple, operational programming language, including the update operations. For instance, 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.

7.2.2 Triggers on XML Data

Work on triggers for XQuery has e.g. been described in [18] with *Active XQuery* (using the same syntax and switches as SQL, with XQuery in the action component) and in [8, 59], 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 [7].

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*.

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

```
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.

7.2.3 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 [59, 60]. The following proposal refines our previous one developed in [7]:

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

- `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` *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).

7.2.4 Triggers on RDFS and OWL Data

For the Semantic Web, triggers on RDFS and OWL data are more relevant. In these cases, *reactivity* must be combined with *reasoning*. A proposal for triggers in such a scenario has been developed and implemented in [54, 70] based on the Jena Framework [37]; see also Section 10.6.

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 issues of communicating it and detecting it by the node that actually processes the business rule have to be dealt with separately.

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 similar 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 is required even for detecting atomic events "somewhere in the Web". The timepoints of actual events and the event detection may differ.

7.3 Domain Node Architecture Proposal

A domain node fitting with the concepts of MARS, i.e., events and actions are full members of the domain, is developed in the SWAN (Semantic Web Application Node) subproject.

The Jena [37] framework provides an API for dealing with RDF and OWL data. In our architecture, Jena uses an external database (e.g. PostgreSQL) for storing base data. If Jena is used for RDFS or OWL data, an appropriate reasoner can be used. For this, Jena provides both a (restricted) internal reasoner and it can use a separate DL reasoner like or Pellet [61] via the DIG interface [27] conventions or via a Java API (the latter provides a more complete functionality). The core SWAN domain node architecture is based on a service using the Jena framework with PostgreSQL and Pellet. The handling of ontology-level actions and mapping them to the RDF model level is located in a wrapper around the core as shown in Figure 7.2.

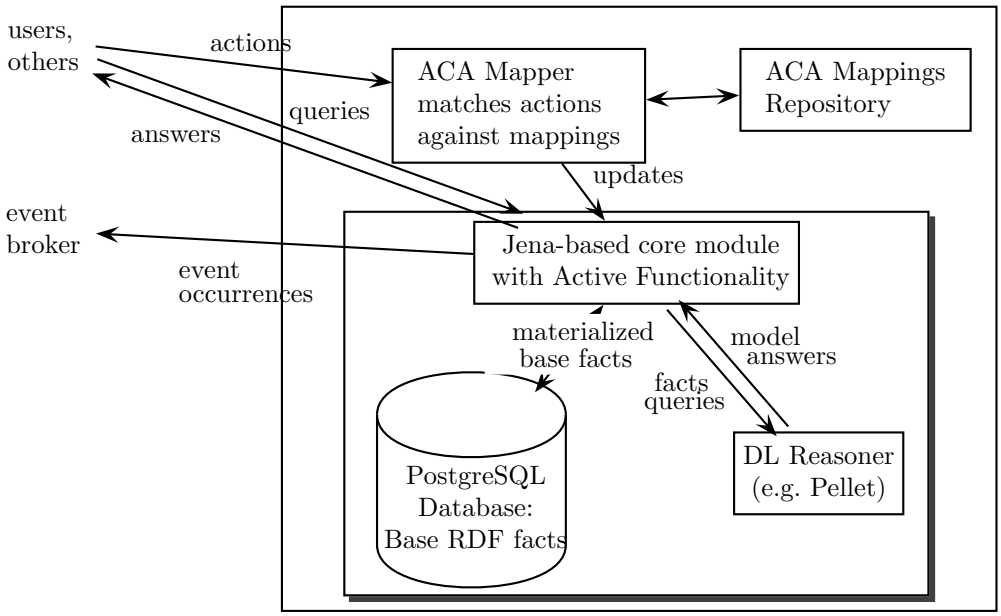


Figure 7.2: Architecture of the Domain Node

7.3.1 Functionality of the Node Core

RDF Database. The base of the node is a pure RDF database, as e.g. provided by the Jena framework [37] with an underlying relational database. It provides storage and SPARQL querying.

OWL Knowledge Base. Using the Jena framework and activating the internal reasoner or connecting to an external OWL reasoner, like e.g. Pellet [61], OWL reasoning over the RDF data is provided.

RDF/OWL Knowledge Base with Updates and Triggers. The next step provides a simple update language for RDF data and supports database triggers reacting upon *database update actions* on the physical level. With this, local ECA rules *inside* the database (that are also required to support actual updates, e.g., when deleting a property that is symmetric) as well as ECE rules raising events can be implemented. An RDF/OWL domain node with such local active behavior based on Jena [37] has been implemented and is described in [54].

7.3.2 Functionality of the Node Wrapper: Mapping of Actions

Ontology-level actions that are requested from the outside arrive in the agreed communication format. Such action requests can e.g. be

```
<travel:DelayFlight travel:flight="LH123" travel:time="30 min" />
```

or, containing tuples of variable bindings,

```
<travel:DelayFlight travel:flight="{$flight} travel:time="{$min}" />
<logvars:variable-bindings>
  <logvars:tuple>
    <logvars:variable name="flight">LH123</logvars:variable>
    <logvars:variable name="time">30 min</logvars:variable>
  </logvars:tuple>
  <logvars:tuple>
    <logvars:variable name="flight">AL400</logvars:variable>
    <logvars:variable name="reason">45 min</logvars:variable>
  </logvars:tuple>
</logvars:variable-bindings>
```

or as a complex XML structure

```
<library:add-book>
  <library:title>Algorithms</library:title>
  <library:author>Thomas Cormen</library:author>
  <library:author>Charles Leiserson</library:author>
  <library:author>Ronald Rivest</library:author>
</library:add-book>
```

In a domain-specific Web Service, such methods are implemented by rules that update the knowledge base accordingly, similar to INSTEAD-triggers that decide if (and optionally how) they are executed. An instance of the generic node architecture is instantiated with a set of such *mappings* which are then applied to the incoming actions.

Example 39 (Domain Node Action Implementation) *Consider the action of delaying a flight. This can be implemented by an SQL-based node as follows:*

```
IMPLEMENT
<travel:DelayFlight travel:flight="{$flight} travel:time="{$min}">
    <travel:reason>$reason</travel:reason>
</travel:DelayFlight>
WHERE EXISTS (SELECT *
                FROM Flights
                WHERE No = getFlightNumber($flight))   % $flight is a URI
```

BY
LET $No = getFlightNumber($flight)
UPDATE Arrivals
SET time:=time+$min WHERE flight=$No
INSERT INTO DELAYED
VALUES($No, $min, $reason)
// *and ignore delays of all other flights*

For an RDF-based node, the implementation could be:

IMPLEMENT
<travel:DelayFlight travel:flight="{$flight} travel:time="{$min}">
  <travel:reason>$reason</travel:reason>
</travel:DelayFlight>
WHERE ($flight-uri, travel:operated-by, *our-url*) % *flight is operated by us*
BY
UPDATE ($flight-uri, travel:leaves, $time, $time+$min)
ASSERT ($flight-uri, travel:flight-is-delayed, $time)
ANNOTATE WITH (travel:reason, $reason)
// *and ignore delays of all other flights*

More complex actions, e.g., adding multiple (*book* has_author *author*) statements in a single action for a book that has multiple authors requires to express more complex actions.

In general, ECA rules (with an action part e.g. using CCS over atomic database updates) could be used. This would require to have local service instances for ECA-ML, an AEM, and CCS.

Prototype Implementation The prototypical SWAN node developed in [54, 70] has been extended for ACA Rules in [71].

The extension defines an XML Markup for RDFUpdate actions. With this, the mapping from actions (that are atomic from the ontology level) to a sequence of database update statements for a given XML input can be seen as a transformation that can –given actions in XML markup – e.g. be expressed in XQuery or XSLT.
The above rule for <library:add-book> can e.g. be expressed as above:

```
let $title := /add-book/title/text()
let $book := library:make-book-uri($title)
return
  INSERT ($book has-title $title)
  for $authorname in /add-book/author/text()
  let $author := library:make-author-uri($authorname)
  INSERT ($book has-author $author)
```

As illustrated above, often the mapping includes to generate URIs for entities. An ontology may provide a function that generates (based on a key attribute) a URI that guarantees that all nodes using this ontology identify a thing by the same URI. In the same way, this function can be local (using e.g. a local prefix), when e.g. a university generates local URIs for students.

7.3.3 Installing and Running a SWAN-based Domain Node

A domain node for a certain domain ontology is set up by instantiating the generic architecture with a set of such ACA mappings. Note that all nodes whose internal database schema is the RDF model of the ontology use the same ACA mapping (that should thus be given when designing the ontology). In this case, also the RDF triggers for raising basic events upon database updates can be given with the ontology.

The domain node provides a method (for local use by its maintainers) where ACA mappings can be submitted. When a mapping is submitted, its language must be indicated or guessed by the node:

<applnode:aca-mapping language="xslt|xquery">
opaque code in the indicated language
</applnode:aca-mapping>

Processing Incoming Actions. For every incoming action, all mappings are applied and the resulting updates are executed on the database. The internal database-level triggers support the actual execution of (intensional) updates and consistency maintenance. They also raise events that are communicated to the registered domain/event brokers.

Outlook: RDF Model. When migrating to a complete RDF environment, events may be exchanged as RDF data. Then, the above mapping can be completely implemented by SPARQL – querying the event and the database and updating the database.

Putting a Domain Node on the Web. Domain nodes need to communicate with other nodes. For that, they must be registered at Domain Brokers, whose functionality is described in the next section.

Chapter 8

Domain Brokering

Domain Brokering is concerned with forming the "Web" out of individual nodes. The Web supports the following tasks

- queries against "the Semantic Web": finding one or more relevant data/knowledge sources, and integration of their data. Example (ICLP 2004): "give me connections from Göttingen to St. Malo". The answer involves finding appropriate flights and train connections – the optimal solution would have been to fly from Hannover to Jersey and to use the afternoon ferry.
- events in the Semantic Web: ECA rules react on events "anywhere". Event Brokering means to bring together event occurrences and rules that should react on them.
- actions in the Semantic Web: while most actions can be considered to be related to a given node, sometimes actions are general in the sense that they can be performed successfully at many nodes; e.g. "if my flight is delayed, book me a hotel room near to the airport".

8.1 Description of Application Services

Application services (e.g., airlines) use (one or more) domain ontologies, e.g., an airline service will support travel and business (for running its business, paying taxes), etc. For supporting the work of domain brokers, 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).

When a "new" service is put on the Semantic Web, it registers itself at one or more domain brokers by sending them its ontology data, which mainly consists of tripels of the form

(*service-uri*, mars:supports, *domain:notion*)

From the ontology of *domain*, the domain broker knows whether *domain:notion* is an owl:Class, an rdf:Property, a mars:Event or a mars:Action, and can handle it appropriately as described in the subsequent sections.

Example 40 (Metadata on Application Services) *Consider an application service* Orafly *in the traveling area (running the respective business and being implemented on some infrastructure):*

```
<!DOCTYPE rdf:RDF [
  <!ENTITY travel "http://www.semwebtech.org/domains/2006/travel#">
  <!ENTITY business "http://www.semwebtech.org/domains/2006/business#"> ]>
<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:mars="http://www.semwebtech.org/mars/2006/mars#"
        xmlns:travel="http://www.semwebtech.org/domains/2006/travel#">
 <mars:Service rdf:about="http://www.semwebtech.org/nodes/2006/orafly">
  <rdf:type rdf:resource="&travel;airline"/>
  <mars:uses-domain rdf:resource="&travel;"/>
  <mars:uses-domain rdf:resource="&business;"/>
  <business:has-taxnumber>eur-0815</business:has-taxnumber>
  <mars:supports rdf:resource="&travel;Airport"/>
  <mars:supports rdf:resource="&travel;FlightConnection"/>
  <mars:supports rdf:resource="&travel;DelayedFlight"/>
  <mars:supports rdf:resource="&travel;DelayFlight"/>
  <mars:supports rdf:resource="&travel;CanceledFlight"/>
  <mars:supports rdf:resource="&travel;CancelFlight"/>
  <mars:supports rdf:resource="&travel;is-delayed"/>
  <mars:supports rdf:resource="&travel;is-canceled"/>
  <mars:supports rdf:resource="&business;has-taxnumber"/>
 </mars:Service>
</rdf:RDF>
```

8.2 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 8.1. The main communication between DNs and EBs is very simple: A DN forwards all events of a domain (as XML fragments according to the domain ontology) to its peer EB(s) for that domain.

Communication of Events between AEM and EB. As described in Section 5.5, 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 *Domain Services Registry (DSR)* (see Section 9.6) what EBs are serving this domain. The AEM then tells an EB to forward it the required events. The specification which events should be forwarded contains the domain name/URI (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 5.4, 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 41 (Registration at the Event Broker) *Assume that an AEM for the sample XML-QL-style event matching formalism gets the following AES to be detected:*

<travel:CanceledFlight xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
    travel:number="{$flight}" />

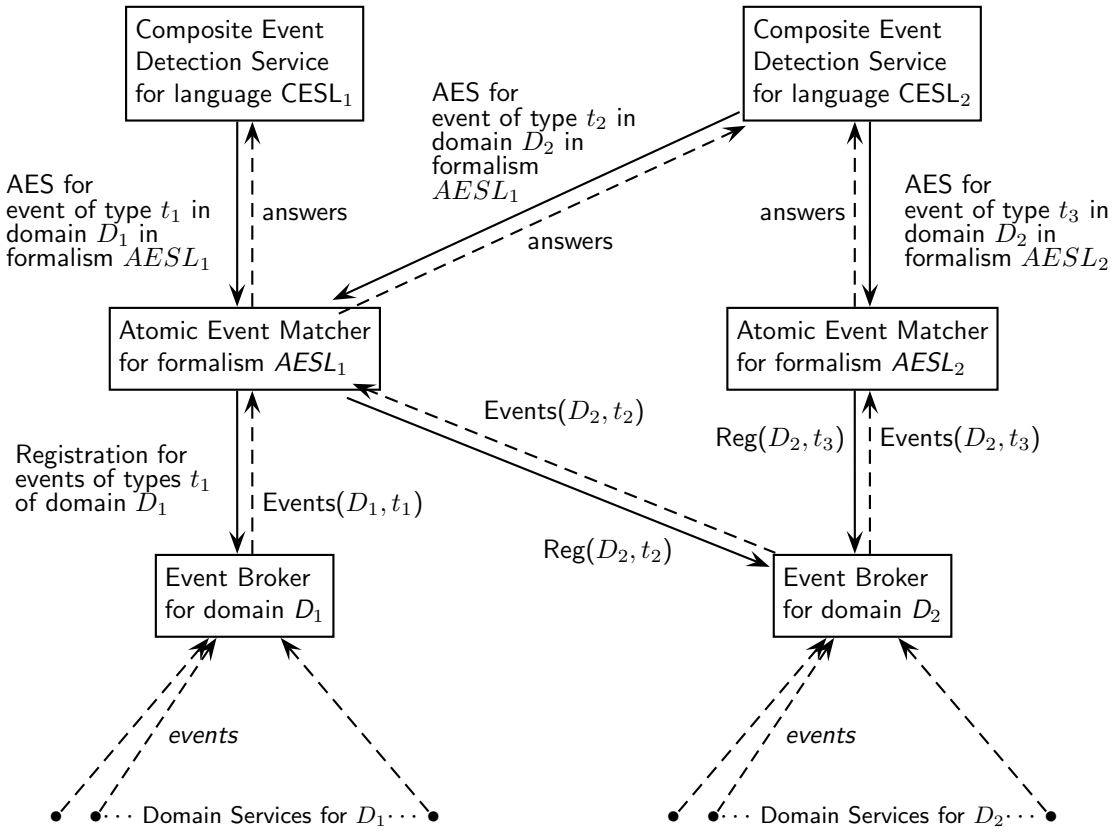


Figure 8.1: Architecture: Communication with Event Brokers

It then explores which EB serves the travel domain (for details, see Chapter 9). Using the fact that the root element of the event is travel:CanceledFlight*, it tells the EB to forward all* travel:CanceledFlight *events 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 the 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 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.

8.3 Brokering of Derived Events

Derived events (cf. Section 6.2.2) are specified as ECE rules whose mars: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>
    <mars:raise-event>
      <!-- head of the ECE rule -->
    </mars: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 rule.

8.3.1 RSS-based Event Brokering

Event Brokering for a given domain may include to raise events obtained from RSS feeds. The broker polls the feed regularly. It first removes the items that have already processed. Then, for each new item, an intermediate 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.*

8.4 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 domain 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 existing algorithms for mediating and integrating information can be applied. To get the infrastructure running, a simple approach is followed first.

Prototype: Simple Approach

Example 42 *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, also declarations of

- owl:inverseOf and
- owl:equivalentProperty

must be considered. Moreover, if the ontology contains rules of the form *head* ← *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 short from in an RDF dictionary as (derivable from RuleML markup):

<*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 mars: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.

8.5 Action Brokering

Clients can request actions of domain ontologies (i.e., which are instances of subclasses of mars:Action) either at certain domain nodes, or at the domain broker:

- Actions that are atomic wrt. the application domain level. These actions are supported by the respective domain nodes, to which they are forwarded by the Domain Broker.
- Actions that are defined by a composition of simpler actions (using a global, ontology-defined ACA rule). The Domain Brokers are responsible for handling the ACA rules.

8.5.1 Required Information for Action Brokering

As described above, the Domain Broker contains information (i) about which domain nodes support which (atomic) actions, and (ii) about the definitions of complex actions.

Atomic Actions. For atomic actions, in the same way as for atomic queries, the domain nodes declare by mars:support which actions they support. When a Domain Node registers at a Domain Broker, the support information is stored at the broker. Atomic actions are then forwarded to relevant nodes, i.e., all services known to the domain broker that mars:support the notion. There, they are processed as described in Section 7.3.2.

Definitions of Complex Actions As described in Section 6.2.3, the domain ontologies contain ACA Rules that define the reduction from complex actions to simpler ones (e.g., by giving a CCS expression). When initializing a Domain Broker with an ontology, these rules are stored in the broker. ACA Rules have a close similarity to ECA rules, they are actually handled by an architectural extension of ECA engines and appropriate communication; see Section 8.5.2.

Data-Dependent Forwarding Actually, most actions do not concern all nodes that mars:support that action type, but depending on some (key) data, only some of them.

Example 43 *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 $flights departing in the next hour </eca:Query>
  <eca:Action>
    <travel:DelayFlight travel:flight="{$flight}" travel:delay="1h">
      <travel:reason>bad weather conditions</travel:reason>
    </travel:DelayFlight>
  </eca:Action>
</eca:Rule>
```

The travel ontology contains a triple

(travel:DelayFlight, rdf:type, mars:Action)

that indicates that this is indeed an action. Consider now the action instance

```
<travel:DelayFlight travel:flight="LH123" travel:delay="1h">
  <travel:reason>bad weather conditions</travel:reason>
</travel:DelayFlight>
```

or, using an RDF/XML event and a URI for the flight,

```
<travel:DelayFlight travel:delay="1h">
  <travel:flight rdf:resource="&travel;/airlines/lufthansa/flights/lh123" />
  <travel:reason>bad weather conditions</travel:reason>
</travel:DelayFlight>
```

The travel:DelayFlight action is mars:supported by Domain Nodes that represent airlines, but e.g. not by the train companies and hotels:

(http://lufthansa.com, mars:supports, travel:DelayFlight)
(http://airfrance.com, mars:supports, travel:DelayFlight) etc.

Thus, the above atomic action is forwarded to these nodes. Actually, the delay of LH123 only concerns the Lufthansa node (the others will ignore it).

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).

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 44 (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.

8.5.2 Handling of Composite Actions by ACA Rules

As discussed in Section 6.2.3, 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 used. 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 deals with an ECA or an ACA rule, 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 9.7.

Chapter 9

The MARS Architecture, Ontology, Language and Service Metadata

In this section, we first describe (in fact, summarize, because its parts have already been described in the previous sections) the Web-Service-based architecture where each language is associated with one or more Web Services that are "responsible" for the language. Afterwards we will then systematically describe how the architecture is realized.

Rules are registered at some *rule evaluation service*, here, an ECA engine. The rule evaluation engine 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 9.5.1), the URLs and communication details of language processors can be found out. In Section 9.1, we describe a general Web-wide architecture based on autonomous Web Services. Section 9.2 describes variants that can especially be applied in "closed" environments.

Section 9.3 provides the formal ontology of concepts in MARS (that actually collects and orders the concepts discussed in the previous sections). Section 9.4 specifies the abstract interfaces of the types of services. Section 9.5 describes how actual services are found, how the concrete communication interfaces are described, and how the actual communication takes place.

9.1 Architecture and Processing: Cooperation between Resources

The overall communication in an open Semantic Web scenario is depicted in Figure 9.1. The MARS infrastructure cares for the choice of appropriate language processors, and for actually detecting the relevant events, collecting the relevant knowledge, and appropriately executing the actions.

A client registers a rule (in the travel domain) at the ECA engine (Step 1.1). The ECA engine submits the event component to the appropriate CED service (1.2), here, a SNOOP service. The SNOOP service inspects the namespaces of the atomic event specifications and registers all atomic event patterns at an appropriate AEM service (1.3). The AEM inspects the namespaces of the used domains, and sees that the travel ontology is relevant. It 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 incoming 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, SNOOP submits the result to the ECA engine (4).

This means the actual "firing" of the rule which then evaluates additional queries and tests (assumed to be empty here). Then, the action component is executed. The ECA engine inspects the used language namespace (ccs:) and forwards it to a CCS service (5.1). CCS executes two actions: first, booking a ticket at LH (5.2a) by sending the request to the travel: domain broker who forwards it the the appropriate domain service at LH (5.3a), and then asks an SMTP service to send a confirmation to the user (5.2b, 5.3b).

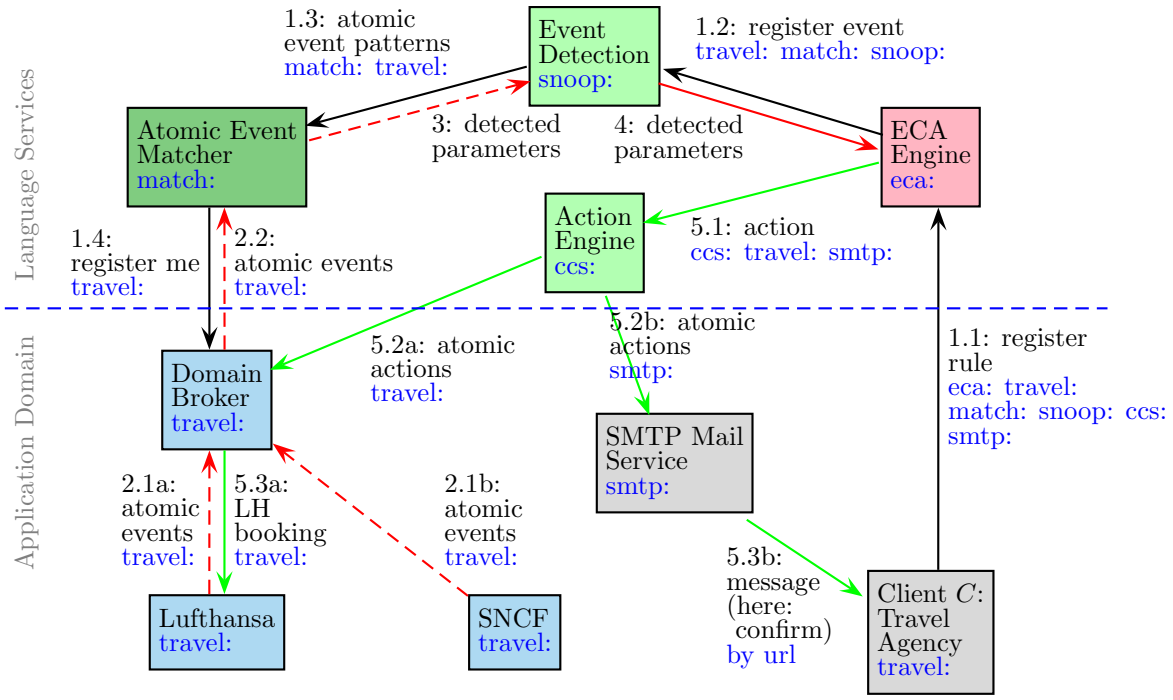


Figure 9.1: Communication: Event Processing

9.2 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 45 (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 node(s) or with the domain broker:

- application services (e.g. for travel:) provide matching functionality (having an AEM for some specification formalism), or
- domain/event brokers (e.g. for travel:) can provide matching functionality (also having an AEM for some specification formalism).

ECA-Engine-Centered (without Domain Broker). In this case, there is no domain broker, but the ECA engine plays the "central" role (see Figure 9.2): clients C register rules to be "supervised" at the ECA 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 signaled 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.

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).

9.3 Ontology of Languages and Services

The Semantic Web as a whole, and also its dynamic aspects as implemented in the MARS Framework combine a multitude of ontologies. Domains, languages and services are themselves resources (identified by their namespace URI). The architectural relationship between these resources provides also the base for managing the actual communication.

9.3.1 The Ontology

The concepts of the MARS Framework are described in an ontology. There are four main kinds of things:

- MARS meta concepts: classes, properties, actions, events, languages, services etc.,
- *domain ontologies*, e.g. travel:, uni: that provide concepts (classes), properties/relationships, atomic events and actions. They have been described in Chapter 6.
- *languages*, that can be partitioned into *language families*: ECA languages (e.g. ECA-ML), event algebras (e.g. SNOOP), query languages (e.g. the relational algebra), and process algebras (e.g. CCS). These language families can also be seen as ontologies. Their details will be discussed again in Section 11.

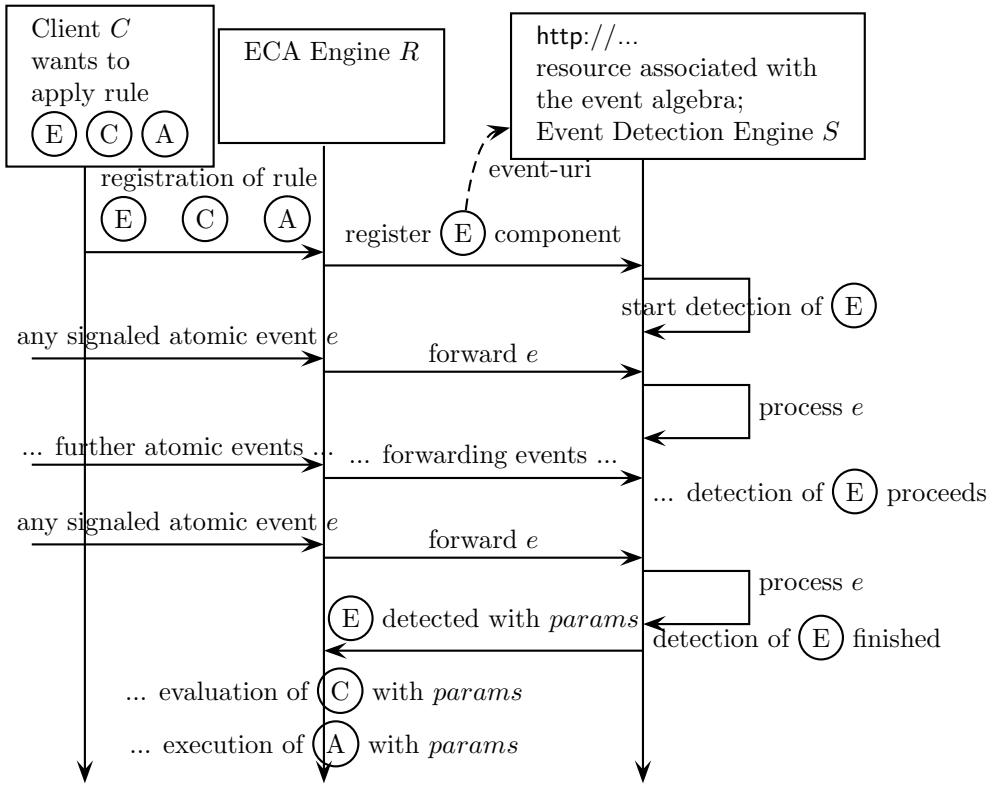


Figure 9.2: Dependent Communication

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 the kind of the language that is implemented by it.

We focus here mainly on the component and domain languages for the MARS Framework, but most considerations hold also for any language in the XML world. The ontology will be given in OWL below.

Languages in the MARS Framework. The MARS framework distinguishes several families of languages (as shown in Figure 3.4):

- the ECA language,
- languages for the event component (split into Composite Event Languages and Atomic Event Specification Languages),
- languages for the query component,
- languages for the test component,
- languages for the action component,
- domain languages.

Apart from the MARS Framework, there may be various other kinds of languages that also form subclasses of mars: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 MARS 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:

- for 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 should be provided. 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 the MARS Framework mainly ECA-ML and the component languages, but the same considerations also hold for other programming languages. When defining such a language, the following information related to the language has to be provided:

- 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, these 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 multiple 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 9.5.1).

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 MARS 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 Section 4.1. Additionally there are several properties of services from the technical point of view that are to be indicated in a service description (see Section 9.5.1).

9.3.2 The MARS Subontology of Domain-Related Notions

Ontological information about the domains, including their *dynamic* concepts, is supported by the domain-related part of the MARS ontology:

- a meta-ontology of static and dynamic notions,
- which actual Semantic Web Nodes support a domain and its concepts, and
- which domain brokers support the domain.

```
<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/mars/2006/mars#"
         xml:base="http://www.semwebtech.org/mars/2006/mars">

 <owl:Class rdf:ID="Domain"/>
 <owl:Class rdf:ID="DomainNotion"/>
 <owl:Class rdf:ID="Class">
   <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
   <rdfs:subClassOf rdf:resource="#DomainNotion"/>
 </owl:Class>
 <owl:Class rdf:ID="Property">
   <rdfs:subClassOf rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"/>
   <rdfs:subClassOf rdf:resource="#DomainNotion"/>
 </owl:Class>
 <owl:Class rdf:ID="Action">
   <rdfs:subClassOf rdf:resource="#DomainNotion"/>
 </owl:Class>
 <owl:Class rdf:ID="Event">
   <rdfs:subClassOf rdf:resource="#DomainNotion"/>
 </owl:Class>

 <rdf:Property rdf:ID="belongs-to-domain">
  <rdfs:domain rdf:resource="#DomainNotion"/>
  <rdfs:range rdf:resource="#Domain"/>
 </rdf:Property>

 <owl:Class rdf:ID="DomainService">
  <rdfs:subClassOf rdf:resource="#Service"/>
 </owl:Class>

 <rdf:Property rdf:ID="uses-domain">
  <rdfs:domain rdf:resource="#DomainService"/>
  <rdfs:range rdf:resource="#Domain"/>
```

```
 <inverseOf rdf:resource="#has-service"/>
</rdf:Property>

<rdf:Property rdf:ID="supports">
 <rdfs:domain  rdf:resource="#DomainService"/>
</rdf:Property>

<owl:Class rdf:ID="DomainBroker">
 <rdfs:subClassOf rdf:resource="#Service"/>
</owl:Class>

<rdf:Property rdf:ID="has-domain-broker">
 <rdfs:domain rdf:resource="#Domain"/>
 <rdfs:range rdf:resource="#DomainBroker"/>
</rdf:Property>
</rdf:RDF>
```

9.3.3 MARS Languages and Services Ontology

The MARS Languages and Services Ontology defines the types of languages and corresponding services that are known in the MARS Framework. Note that these are not exclusive to MARS, but the same classes can also be relevant to other ontologies. For each type of language, the type of appropriate services is given, and the allowed kinds of answers (i.e., a ResultSet (functional style), variable bindings (Datalog Style), Answers (cf. Section 4.2.3; a result set where each result is annotated by a set of tuples of variable bindings), or Boolean) are listed. From the modeling point of view, the latter could be either associated with language types or with service types, but as they belong to the semantics of the language, it is preferable to associate them with the language types. Services provide tasks (that are "typical" for their service class). The actual tasks are defined later in Section 9.4.5.

```
<?xml version="1.0"?>
<!-- filename: mars-language-and-service-types.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/mars/2006/mars#"
         xml:base="http://www.semwebtech.org/mars/2006/mars">

  <owl:Class rdf:ID="LanguageClass"/>
  <owl:Class rdf:ID="Language">
   <owl:disjointWith rdf:resource="#LanguageClass"/>
  </owl:Class>

  <owl:Class rdf:ID="ResultType"/>

  <ResultType rdf:ID="VariableBindings"/>
  <ResultType rdf:ID="Answers"/>
  <ResultType rdf:ID="ResultSet"/>
  <ResultType rdf:ID="Boolean"/>

  <rdf:Property rdf:ID="recommended-result-type">
    <rdfs:domain rdf:resource="#LanguageClass"/>
    <rdfs:range rdf:resource="#ResultType"/>
```

```
</rdf:Property>
<rdf:Property rdf:ID="allowed-result-type">
  <rdfs:domain rdf:resource="#LanguageClass"/>
  <rdfs:range rdf:resource="#ResultType"/>
</rdf:Property>

<LanguageClass rdf:ID="RuleLanguage">
  <rdfs:subClassOf rdf:resource="#Language"/>
</LanguageClass>
<LanguageClass rdf:ID="AlgebraicLanguage">
  <rdfs:subClassOf rdf:resource="#Language"/>
</LanguageClass>
<LanguageClass rdf:ID="ECALanguage">
  <rdfs:subClassOf rdf:resource="#RuleLanguage"/>
  <has-service-type rdf:resource="#ECAService"/>
</LanguageClass>
<LanguageClass rdf:ID="EventAlgebra">
  <rdfs:subClassOf rdf:resource="#AlgebraicLanguage"/>
  <has-service-type rdf:resource="#CompositeEventDetectionEngine"/>
  <recommended-result-type rdf:resource="#Answers"/>
  <allowed-result-type rdf:resource="#VariableBindings"/>
  <allowed-result-type rdf:resource="#ResultSet"/>
</LanguageClass>
<LanguageClass rdf:ID="AtomicEventFormalism">
  <rdfs:subClassOf rdf:resource="#Language"/>
  <has-service-type rdf:resource="#AtomicEventMatcher"/>
  <recommended-result-type rdf:resource="#Answers"/>
  <allowed-result-type rdf:resource="#VariableBindings"/>
  <allowed-result-type rdf:resource="#ResultSet"/>
</LanguageClass>
<LanguageClass rdf:ID="QueryLanguage">
  <rdfs:subClassOf rdf:resource="#Language"/>
  <has-service-type rdf:resource="#QueryService"/>
  <allowed-result-type rdf:resource="#Answers"/>
  <allowed-result-type rdf:resource="#VariableBindings"/>
  <allowed-result-type rdf:resource="#ResultSet"/>
</LanguageClass>
<LanguageClass rdf:ID="ActionLanguage">
  <rdfs:subClassOf rdf:resource="#Language"/>
  <has-service-type rdf:resource="#ActionService"/>
  <allowed-result-type rdf:resource="#VariableBindings"/>
  <allowed-result-type rdf:resource="#Boolean"/>
</LanguageClass>
<LanguageClass rdf:ID="ProcessAlgebra">
  <rdfs:subClassOf rdf:resource="#AlgebraicLanguage"/>
  <rdfs:subClassOf rdf:resource="#ActionLanguage"/>
  <has-service-type rdf:resource="#ActionService"/>
  <allowed-result-type rdf:resource="#VariableBindings"/>
  <allowed-result-type rdf:resource="#Boolean"/>
</LanguageClass>

<owl:DatatypeProperty rdf:ID="name"/>

<rdf:Property rdf:ID="hasMarkup">
```

```
  <rdfs:range rdf:resource="#MarkupDescription"/>
</rdf:Property>
<rdf:Property rdf:ID="hasRDF">
  <rdfs:subPropertyOf rdf:resource="hasMarkup"/>
</rdf:Property>
<rdf:Property rdf:ID="hasDTD">
  <rdfs:subPropertyOf rdf:resource="hasMarkup"/>
  <rdfs:range rdf:resource="#DTD"/>
</rdf:Property>
<rdf:Property rdf:ID="hasStripedDTD">
  <rdfs:subPropertyOf rdf:resource="hasMarkup"/>
  <rdfs:range rdf:resource="#DTD"/>
</rdf:Property>
<rdf:Property rdf:ID="hasStrippedDTD">
  <rdfs:subPropertyOf rdf:resource="hasMarkup"/>
  <rdfs:range rdf:resource="#DTD"/>
</rdf:Property>
<rdf:Property rdf:ID="hasXSD">
  <rdfs:subPropertyOf rdf:resource="hasMarkup"/>
</rdf:Property>

<!-- note that the next also defines "Language-Class" -->

<rdf:Property rdf:ID="has-service-type">
  <rdfs:domain rdf:resource="#LanguageClass"/>
  <rdfs:range rdf:resource="#ServiceClass"/>
</rdf:Property>

<owl:Class rdf:ID="ServiceClass">
 <owl:disjointWith rdf:resource="#LanguageClass"/>
</owl:Class>
<owl:Class rdf:ID="Service">
 <owl:disjointWith rdf:resource="#ServiceClass"/>
 <owl:disjointWith rdf:resource="#Language"/>
</owl:Class>

<ServiceClass rdf:ID="ECAService">
  <rdfs:subClassOf rdf:resource="#Service"/>
</ServiceClass>
<ServiceClass rdf:ID="CompositeEventDetectionEngine">
  <rdfs:subClassOf rdf:resource="#Service"/>
</ServiceClass>
<ServiceClass rdf:ID="AtomicEventMatcher">
  <rdfs:subClassOf rdf:resource="#Service"/>
</ServiceClass>
<ServiceClass rdf:ID="QueryService">
  <rdfs:subClassOf rdf:resource="#Service"/>
</ServiceClass>
<ServiceClass rdf:ID="ProcessAlgebraService">
  <rdfs:subClassOf rdf:resource="#Service"/>
</ServiceClass>
<ServiceClass rdf:ID="DomainBroker">
  <rdfs:subClassOf rdf:resource="#Service"/>
</ServiceClass>
```

```
  <ServiceClass rdf:ID="DomainService">
    <rdfs:subClassOf rdf:resource="#Service"/>
  </ServiceClass>

  <!-- 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>

  <owl:TransitiveProperty rdf:ID="specializedTaskFrom">
    <rdfs:domain rdf:resource="#Task"/>
    <rdfs:range rdf:resource="#Task"/>
  </owl:TransitiveProperty>

  <rdf:Property rdf:ID="meta-provides-task">
    <rdfs:domain rdf:resource="#ServiceClass"/>
    <rdfs:range rdf:resource="#Task"/>
  </rdf:Property>

  <!-- languages are implemented by services of certain
   service types -->

  <rdf:Property rdf:ID="is-implemented-by">
    <rdfs:domain rdf:resource="#Language"/>
    <rdfs:range rdf:resource="#Service"/>
  </rdf:Property>

</rdf:RDF>
```

9.4 Service Interfaces and Functionality

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 9.5.1) which are based on the lists of tasks given next. The complete ontology is given in RDF in Section 9.4.5.

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. Additionally, the answers must be received (note that the respective addresses are communicated with the ReplyTo), but details of the communication format must again 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).

9.4.1 ECA Services

An ECA service must implement the ECA-ML language and adhere to the abstract semantics of ECA rules given in Sections 4.1 and 4.4.

9.4.1.1 Upper Interface:

Registration of Rules: [R] a rule can be registered as 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.

Receiving Events: [R, Optional] Events can be sent directly to the ECA engine that forwards them to the AEMs. This is useful for (i) testing and (ii) in a closed environment where no Domain Broker is available/needed.

9.4.1.2 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 5.5).
- [R] receive answers for detected events.
- [S] Query components: submit query together with bound variables to appropriate service. The answer can be returned immediately and synchronously in the same HTTP communication, or asynchronously.
- [R] receive asynchronous answers to queries (and tests).
- [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 then 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.

9.4.2 (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] receive a statement to be validated,
  - [R, optional] receive a statement and analyze its variable use (used variables, input variables, output variables, and returned variables; based on the declarations of the atomic expressions in the component),
- [S] upper interface: send answers.
- lower interface: mostly communication 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].

9.4.3 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).

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 8.1),
  - 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 registered AEMs,
- [R] receive queries, and [S] evaluate them against domain, receive answers [R], combine them, and [R] return answers,
- [R] receive atomic actions to be executed and [S] forward them to domain nodes.

9.4.4 Domain Services

- provide a certain service in a 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).

9.4.5 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/mars/2006/mars#"
         xml:base="http://www.semwebtech.org/mars/2006/mars">

  <ServiceClass rdf:about="#ECAService">
    <meta-provides-task rdf:resource="./eca-service#register-rule"/>
    <meta-provides-task rdf:resource="./eca-service#deregister-rule"/>
    <meta-provides-task rdf:resource="./eca-service#disable-rule"/> <!-- opt -->
    <meta-provides-task rdf:resource="./eca-service#enable-rule"/> <!-- opt -->
    <meta-provides-task rdf:resource="./eca-service#incoming-events"/> <!-- opt -->

    <meta-provides-task rdf:resource="./eca-service#receive-detected-event"/>
    <meta-provides-task rdf:resource="./eca-service#receive-query-answer"/>
    <meta-provides-task rdf:resource="./eca-service#give-service-description"/> <!-- opt -->
  </ServiceClass>
  <ServiceClass rdf:about="#EventDetectionService">
    <meta-provides-task rdf:resource="./eds#register-event-pattern"/>
    <meta-provides-task rdf:resource="./eds#deregister-event-pattern"/>
    <meta-provides-task rdf:resource="./eds#validate-pattern"/> <!-- opt -->
    <meta-provides-task rdf:resource="./eds#analyze-variables"/> <!-- opt -->
    <meta-provides-task rdf:resource="./eds#give-service-description"/> <!-- opt -->
  </ServiceClass>
  <ServiceClass rdf:about="#CompositeEventDetectionEngine">
    <meta-provides-task>
     <Task rdf:about="./ced#register-event-pattern">
      <specializedTaskFrom rdf:resource="./eds#register-event-pattern"/>
     </Task>
    </meta-provides-task>
    <meta-provides-task>
     <Task rdf:about="./ced#deregister-event-pattern">
      <specializedTaskFrom rdf:resource="./eds#deregister-event-pattern"/>
     </Task>
    </meta-provides-task>
    <meta-provides-task>
     <Task rdf:about="./ced#validate-pattern"> <!-- opt -->
      <specializedTaskFrom rdf:resource="./eds#validate-pattern"/>
     </Task>
    </meta-provides-task>
    <meta-provides-task>
     <Task rdf:about="./ced#analyze-variables"> <!-- opt -->
      <specializedTaskFrom rdf:resource="./eds#analyze-variables"/>
     </Task>
    </meta-provides-task>
    <meta-provides-task rdf:resource="./ced#receive-detected-event"/>
```

```
  <meta-provides-task>
   <Task rdf:about="./ced#give-service-description"> <!-- opt -->
    <specializedTaskFrom rdf:resource="./eds#give-service-description"/>
   </Task>
  </meta-provides-task>
</ServiceClass>
<ServiceClass rdf:about="#AtomicEventMatcher">
  <meta-provides-task>
   <Task rdf:about="./aem#register-event-pattern">
    <specializedTaskFrom rdf:resource="./eds#register-event-pattern"/>
   </Task>
  </meta-provides-task>
  <meta-provides-task>
   <Task rdf:about="./aem#deregister-event-pattern">
    <specializedTaskFrom rdf:resource="./eds#deregister-event-pattern"/>
   </Task>
  </meta-provides-task>
  <meta-provides-task>
   <Task rdf:about="./aem#validate-pattern"> <!-- opt -->
    <specializedTaskFrom rdf:resource="./eds#validate-pattern"/>
   </Task>
  </meta-provides-task>
  <meta-provides-task>
   <Task rdf:about="./aem#analyze-variables"> <!-- opt -->
    <specializedTaskFrom rdf:resource="./eds#analyze-variables"/>
   </Task>
  </meta-provides-task>
  <meta-provides-task rdf:resource="./aem#receive-event"/>
  <meta-provides-task>
   <Task rdf:about="./aem#give-service-description"> <!-- opt -->
    <specializedTaskFrom rdf:resource="./eds#give-service-description"/>
   </Task>
  </meta-provides-task>
</ServiceClass>
<ServiceClass rdf:about="#QueryService">
  <meta-provides-task rdf:resource="./qs#evaluate-query"/>
  <meta-provides-task rdf:resource="./qs#evaluate-test"/> <!-- opt -->
  <meta-provides-task rdf:resource="./qs#register-query"/> <!-- opt -->
  <meta-provides-task rdf:resource="./qs#analyze-variables"/>  <!-- opt -->
  <meta-provides-task rdf:resource="./qs#deregister-query"/> <!-- opt -->
  <meta-provides-task rdf:resource="./qs#invoke-query"/> <!-- opt -->
  <meta-provides-task rdf:resource="./qs#validate-pattern"/> <!-- opt -->
  <meta-provides-task rdf:resource="./qs#analyze-variables"/> <!-- opt -->
  <meta-provides-task rdf:resource="./qs#give-service-description"/> <!-- opt -->
</ServiceClass>
<ServiceClass rdf:about="#ActionService">
  <meta-provides-task rdf:resource="./action-service#execute-action"/>
  <meta-provides-task rdf:resource="./action-service#register-action"/> <!-- opt -->
  <meta-provides-task rdf:resource="./action-service#deregister-action"/> <!-- opt -->
  <meta-provides-task rdf:resource="./action-service#receive-query-answer"/> <!-- opt -->
  <meta-provides-task rdf:resource="./action-service#receive-detected-event"/> <!-- opt -->
  <meta-provides-task rdf:resource="./action-service#validate-pattern"/> <!-- opt -->
  <meta-provides-task rdf:resource="./action-service#analyze-variables"/> <!-- opt -->
  <meta-provides-task rdf:resource="./action-service#give-service-description"/> <!-- opt -->
```

```
  </ServiceClass>
  <ServiceClass rdf:about="#DomainBroker">
    <meta-provides-task rdf:resource="./domain-broker#register-ontology"/> <!-- init -->
    <meta-provides-task rdf:resource="./domain-broker#update-ontology"/> <!-- opt -->
    <meta-provides-task rdf:resource="./domain-broker#receive-support-information"/>
    <meta-provides-task rdf:resource="./domain-broker#update-support-information"/> <!-- opt -->
    <meta-provides-task rdf:resource="./domain-broker#register-for-event"/>
    <meta-provides-task rdf:resource="./domain-broker#receive-event"/>
    <meta-provides-task rdf:resource="./domain-broker#execute-query"/>
    <meta-provides-task rdf:resource="./domain-broker#receive-query-answer"/>
    <meta-provides-task rdf:resource="./domain-broker#execute-action"/>
    <meta-provides-task rdf:resource="./domain-broker#give-service-description"/> <!-- opt -->
  </ServiceClass>
  <ServiceClass rdf:about="#DomainService">
    <meta-provides-task rdf:resource="./domain-node#receive-query"/>
    <meta-provides-task rdf:resource="./domain-node#receive-action"/>
    <meta-provides-task rdf:resource="./domain-node#give-service-description"/> <!-- opt -->
  </ServiceClass>
</rdf:RDF>
```

9.5 Locating and Contacting Language Services

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 "language borders" (which are in XML markup indicated by 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 5.3.2),
- 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 9.5.1. 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 9.5.4) that can be implemented as a standalone service or provided as an instance of a downloadable Java class.

9.5.1 Language & Service Registries

The languages and the actual service instances are handled by *Language & Service Registries (LSRs)*. The current LSR is just an RDF/XML file (i.e., can be processed by XQuery, or by SPARQL) located at the MARS site. In a real environment, Language & Service Registries will 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)",
- answer: the task description in XML or RDF format.

The ontology discussed in Section 9.4.5 specifies 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, the actual communication (which URI, which format) must be determined.

9.5.2 Interface Descriptions of Individual Tasks

MARS-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 9.4.5.
- URLs where the respective input is expected (relative to the service url; items that are not given default to the service url),
- method: POST/GET; default is POST
- 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 asynchronous cases), the message contents itself (which is often a 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 4.2.1 is accepted.
  - `1`: one tuple of variable bindings in the markup given in Section 4.2.1 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 something 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 required `input` variables.
- 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 4.1.5 and requirements on Page 37.
  - `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.
- `mode[1]` [query, test, other service queries]:
  - synchronous: answer is given synchronously (includes one-way communication where only a message is sent)
  - asynchronous: answer will be given asynchronously (in this case, the immediate answer is "OK", and the result is latter returned with appropriate identification (Subject)).
  - Default is synchronous.
- `mode[2]` [query, test]: does the answer contain/consider all tuples, or is it possible that several results are sent?
  - partial: indicates that the service probably returns incomplete answers. Default: no (=complete).

Example 46 (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:mars="http://www.semwebtech.org/mars/2006/mars#"
         xmlns="http://www.semwebtech.org/mars/2006/lsr#"
         xml:base="http://www.semwebtech.org/mars/2006/lsr">

 <mars:QueryService rdf:about="http://foo.nop/fancy-query-engine"
     xml:base="http://foo.nop/fancy-query-engine/">
   <!-- sample: only one task description
        clients can register queries at: -->
   <has-task-description>
     <TaskDescription>
       <!-- reference to the described task according to the Service Ontology -->
       <describes-task rdf:resource="&mars;/query-engine#register-query"/>
       <!-- url of the service where the task can be actually invoked -->
       <provided-at rdf:resource="./register"/>
       <method>POST</method>
       <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 -->
     </TaskDescription>
   </has-task-description>
 </mars:QueryService>
</rdf:RDF>
```

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-MARS-Aware Services. In case that non-MARS-aware services are used in a rule, the LSR must contain appropriate entries.

- the MARS 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.

9.5.3 The Languages and Services Registry RDF Model

Since the metadata structure is complex, it its preferably expressed in RDF/XML. Then it can be processed as RDF, or as XML file.
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 the MARS Framework are located in the `http://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 [63]) and described by RDF/RDFS files accessible at these URLs.

```
<!-- filename: lsr.rdf -->
<!DOCTYPE rdf:RDF [
  <!ENTITY mars "http://www.semwebtech.org/mars/2006/mars"> ]>
<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:mars="http://www.semwebtech.org/mars/2006/mars#"
         xmlns="http://www.semwebtech.org/mars/2006/lsr#"
         xml:base="http://www.semwebtech.org/mars/2006/lsr">

<rdf:Property rdf:ID="has-task-description">
  <rdfs:domain rdf:resource="&mars;#Service"/>
  <rdfs:range rdf:resource="#TaskDescription"/>
</rdf:Property>

<mars:ECALanguage rdf:about="http://www.semwebtech.org/languages/2006/eca-ml#">
 <mars:shortname>eca-ml</mars:shortname>
 <mars:name>ECA Markup Language</mars:name>
 <mars:hasDTD rdf:resource="http://www.semwebtech.org/languages/2006/eca-ml.dtd"/>
 <mars:hasStripedDTD
    rdf:resource="http://www.semwebtech.org/languages/2006/eca-ml-striped.dtd"/>
 <mars:hasStrippedDTD rdf:resource="http://www.semwebtech.org/languages/2006/eca-ml.dtd"/>
 <mars:hasRDF rdf:resource="http://www.semwebtech.org/languages/2006/eca-ml.rdf"/>
 <mars:is-implemented-by>
  <mars:ECAService
     rdf:about="http://www.semwebtech.org/services/2007/eca-engine"
     xml:base="http://www.semwebtech.org/services/2007/eca-engine/">
   <has-task-description>
```

```
 <TaskDescription>
  <describes-task rdf:resource="&mars;/eca-service#register-rule"/>
  <provided-at rdf:resource="register"/>
  <method>POST</method>
  <Reply-To>none</Reply-To>
  <Subject>body</Subject>
  <input>element register</input>
  <variables>no</variables>
 </TaskDescription>
</has-task-description>
<has-task-description>
 <TaskDescription>
  <describes-task rdf:resource="&mars;/eca-service#deregister-rule"/>
  <provided-at rdf:resource="deregister"/>
  <method>POST</method>
  <Reply-To>none</Reply-To>
  <Subject>body</Subject>
  <input>element deregister</input>
  <variables>n.a.</variables>
 </TaskDescription>
</has-task-description>
<has-task-description>
 <TaskDescription>
  <describes-task rdf:resource="&mars;/eca-service#incoming-events"/>
  <provided-at rdf:resource="incoming-events"/>
  <method>POST</method>
  <Reply-To>none</Reply-To>
  <Subject>none</Subject>
  <input>item *</input>
  <variables>no</variables>
 </TaskDescription>
</has-task-description>
<has-task-description>
 <TaskDescription>
  <describes-task rdf:resource="&mars;/eca-service#receive-detected-event"/>
  <provided-at rdf:resource="notifications"/>
  <method>POST</method>
  <Reply-To>none</Reply-To>
  <Subject>body</Subject>
  <input>element response</input>
  <variables>n.a.</variables>
 </TaskDescription>
</has-task-description>
<has-task-description>
 <TaskDescription>
  <describes-task rdf:resource="&mars;/eca-service#receive-query-answer"/>
  <provided-at rdf:resource="query-answer"/>
  <method>POST</method>
  <Reply-To>none</Reply-To>
  <Subject>body</Subject>
  <input>element response</input>
  <variables>n.a.</variables>
 </TaskDescription>
</has-task-description>
```

```
  </mars:ECAService>
 </mars:is-implemented-by>
</mars:ECALanguage>

<mars:EventAlgebra
  rdf:about="http://www.semwebtech.org/languages/2006/snoopy#">
 <mars:name>SNOOP (from the Sentinel Database System)</mars:name>
 <mars:shortname>snoop</mars:shortname>
 <mars:hasRDF rdf:resource="http://www.semwebtech.org/languages/2006/snoopy.rdf"/>
 <mars:hasDTD rdf:resource="http://www.semwebtech.org/languages/2006/snoopy.dtd"/>
 <mars:hasStripedDTD
    rdf:resource="http://www.semwebtech.org/languages/2006/snoopy-striped.dtd"/>
 <mars:hasStrippedDTD rdf:resource="http://www.semwebtech.org/languages/2006/snoopy.dtd"/>
 <mars:is-implemented-by>
  <mars:CompositeEventDetectionEngine
       rdf:about="http://www.semwebtech.org/services/2007/snoopy"
       xml:base="http://www.semwebtech.org/services/2007/snoopy/">
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/ced#register-event-pattern"/>
     <provided-at rdf:resource="register"/>
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
     <input>element register</input>
     <variables>none</variables>
    </TaskDescription>
   </has-task-description>
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/ced#deregister-event-pattern"/>
     <provided-at rdf:resource="deregister"/>
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
     <input>element deregister</input>
    </TaskDescription>
   </has-task-description>
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/ced#receive-detected-event"/>
     <provided-at rdf:resource="answers"/>
     <Subject>body</Subject>
     <input>element response</input>
     <variables>n.a.</variables>
    </TaskDescription>
   </has-task-description>
  </mars:CompositeEventDetectionEngine>
 </mars:is-implemented-by>
</mars:EventAlgebra>

<mars:AtomicEventFormalism
  rdf:about="http://www.semwebtech.org/languages/2006/aes-xmlql#">
 <mars:name>Atomic Event Detection by XML-QL</mars:name>
 <mars:shortname>xmlqlmatch</mars:shortname>
 <mars:is-implemented-by>
```

```
  <mars:AtomicEventMatcher
     rdf:about="http://www.semwebtech.org/services/2007/aem-xmlql"
     xml:base="http://www.semwebtech.org/services/2007/aem-xmlql/">
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/aem#register-event-pattern"/>
     <provided-at rdf:resource="register"/>
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
     <input>element register</input>
    </TaskDescription>
   </has-task-description>
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/aem#deregister-event-pattern"/>
     <provided-at rdf:resource="deregister"/>
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
     <input>element deregister</input>
    </TaskDescription>
   </has-task-description>
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/aem#receive-event"/>
     <provided-at rdf:resource="events"/>
     <!-- other parameters are not given. Events are simply sent. -->
    </TaskDescription>
   </has-task-description>
  </mars:AtomicEventMatcher>
 </mars:is-implemented-by>
</mars:AtomicEventFormalism>

<mars:AtomicEventFormalism
  rdf:about="http://www.semwebtech.org/languages/2006/aes-xpath#">
 <mars:name>Atomic Event Detection by XPath</mars:name>
 <mars:shortname>aes-xpath</mars:shortname>
 <mars:is-implemented-by>
  <mars:AtomicEventMatcher
     rdf:about="http://www.semwebtech.org/services/2007/aem-xpath"
     xml:base="http://www.semwebtech.org/services/2007/aem-xpath/">
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/aem#register-event-pattern"/>
     <provided-at rdf:resource="register"/>
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
     <input>element register</input>
     <variables>1</variables>
    </TaskDescription>
   </has-task-description>
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/aem#deregister-event-pattern"/>
     <provided-at rdf:resource="deregister"/>
```

```
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
     <input>element deregister</input>
    </TaskDescription>
   </has-task-description>
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/aem#receive-event"/>
     <provided-at rdf:resource="events"/>
     <!-- other parameters are not given. Events are simply sent. -->
    </TaskDescription>
   </has-task-description>
  </mars:AtomicEventMatcher>
 </mars:is-implemented-by>
</mars:AtomicEventFormalism>

<mars:Language
  rdf:about="http://www.semwebtech.org/languages/2006/owlq#">
 <mars:isa rdf:resource="&mars;#AtomicEventFormalism"/>
 <mars:isa rdf:resource="&mars;#QueryLanguage"/>
 <mars:name>The OWLQ AEM and Query Service</mars:name>
 <mars:shortname>OWLQ</mars:shortname>
 <mars:hasRDF rdf:resource="http://www.semwebtech.org/languages/2006/owlq.rdf"/>
 <mars:hasDTD rdf:resource="http://www.semwebtech.org/languages/2006/owlq.dtd"/>
 <mars:hasStripedDTD rdf:resource="http://www.semwebtech.org/languages/2006/owlq.dtd"/>
 <!-- OWLQ does not have a separate stripped DTD -->
 <mars:is-implemented-by>
  <mars:Service
     rdf:about="http://www.semwebtech.org/services/2007/owlq"
     xml:base="http://www.semwebtech.org/services/2007/owlq/">
   <mars:isa rdf:resource="&mars;#AtomicEventMatcher"/>
   <mars:isa rdf:resource="&mars;#QueryService"/>
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/aem#register-event-pattern"/>
     <provided-at rdf:resource="register"/>
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
     <input>element register</input>
    </TaskDescription>
   </has-task-description>
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/aem#deregister-event-pattern"/>
     <provided-at rdf:resource="deregister"/>
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
     <input>element deregister</input>
    </TaskDescription>
   </has-task-description>
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/aem#receive-event"/>
     <provided-at rdf:resource="events"/>
```

```
     <!-- other parameters are not given. Events are simply sent. -->
    </TaskDescription>
   </has-task-description>
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/query-engine#evaluate-query"/>
     <provided-at rdf:resource="eval-query"/>
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
     <input>element request</input>
     <variables>*</variables>
     <mode>asynchronous</mode>
    </TaskDescription>
   </has-task-description>
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/query-engine#evaluate-test"/>
     <provided-at rdf:resource="eval-test"/>
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
     <input>element request</input>
     <variables>*</variables>
     <mode>asynchronous</mode>
    </TaskDescription>
   </has-task-description>
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/query-engine#analyze-variables"/>
     <provided-at rdf:resource="analyze-variables"/>
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
     <input>element request</input>
     <variables>n.a.</variables>
    </TaskDescription>
   </has-task-description>
  </mars:Service>
 </mars:is-implemented-by>
</mars:Language>

<mars:QueryLanguage
  rdf:about="http://www.semwebtech.org/languages/2006/owlq-novars#">
 <!-- for testing only -->
 <mars:name>Simple Query Service</mars:name>
 <mars:shortname>OWLQ-novars</mars:shortname>
 <mars:hasRDF rdf:resource="http://www.semwebtech.org/languages/2006/owlq-novars.rdf"/>
 <mars:hasDTD rdf:resource="http://www.semwebtech.org/languages/2006/owlq-novars.dtd"/>
 <mars:hasStripedDTD rdf:resource="http://www.semwebtech.org/languages/2006/owlq-novars.dtd"/>
 <mars:hasStrippedDTD rdf:resource="http://www.semwebtech.org/languages/2006/owlq-novars.dtd"/>
 <mars:is-implemented-by>
  <mars:Service rdf:about="http://www.semwebtech.org/services/2007/owlq-novars">
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/query-engine#evaluate-query"/>
     <provided-at rdf:resource="http://www.semwebtech.org/services/2007/owlq/novars/eval-query"/>
```

```
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
     <input>element request</input>
     <variables>no</variables>
     <mode>asynchronous</mode>
     <mode>partial</mode>
    </TaskDescription>
   </has-task-description>
  </mars:Service>
 </mars:is-implemented-by>
</mars:QueryLanguage>


<mars:QueryLanguage rdf:about="http://www.w3.org/XQuery">
 <mars:shortname>XQuery</mars:shortname>
 <mars:name>XQuery (Opaque)</mars:name>
 <mars:is-implemented-by>
  <mars:QueryService rdf:about="http://www.semwebtech.org/services/2007/xquery">
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/query-engine#evaluate-query"/>
     <provided-at rdf:resource="http://www.semwebtech.org/services/2007/eca-engine/xquery"/>
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
     <input>element request</input>
     <variables>1</variables>
    </TaskDescription>
   </has-task-description>
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/query-engine#evaluate-test"/>
     <provided-at rdf:resource="http://www.semwebtech.org/services/2007/eca-engine/xquery"/>
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
     <input>element request</input>
     <variables>1</variables>
    </TaskDescription>
   </has-task-description>
  </mars:QueryService>
 </mars:is-implemented-by>
</mars:QueryLanguage>

<mars:QueryLanguage rdf:about="http://www.w3.org/XPath">
 <mars:shortname>XPath</mars:shortname>
 <mars:name>XPath (Opaque)</mars:name>
 <mars:is-implemented-by>
  <mars:QueryService rdf:about="http://www.semwebtech.org/services/2007/xpath">
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/query-engine#evaluate-query"/>
     <provided-at
       rdf:resource="http://www.semwebtech.org/services/2007/eca-engine/xpath"/>
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
```

```
     <input>element request</input>
     <variables>1</variables>
    </TaskDescription>
   </has-task-description>
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/query-engine#evaluate-test"/>
     <provided-at
       rdf:resource="http://www.semwebtech.org/services/2007/eca-engine/xpath"/>
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
     <input>element request</input>
     <variables>1</variables>
    </TaskDescription>
   </has-task-description>
  </mars:QueryService>
 </mars:is-implemented-by>
</mars:QueryLanguage>

<mars:QueryLanguage rdf:about="http://exist.sourceforge.net/NS/exist">
 <mars:shortname>XQuery</mars:shortname>
 <mars:name>XQuery (Opaque)</mars:name>
 <mars:is-implemented-by>
  <mars:QueryService
       rdf:about="http://www.semwebtech.org/util/local/exist-wrapper/direct">
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/query-engine#evaluate-query"/>
     <provided-at rdf:resource="http://www.semwebtech.org/util/local/exist-wrapper/direct"/>
     <input>element request</input>
     <variables>1</variables>
    </TaskDescription>
   </has-task-description>
  </mars:QueryService>
 </mars:is-implemented-by>
</mars:QueryLanguage>

<mars:QueryLanguage rdf:about="http://exist.sourceforge.net/NS/exist">
 <mars:shortname>XQuery</mars:shortname>
 <mars:name>XQuery (Opaque)</mars:name>
 <mars:is-implemented-by>
  <mars:QueryService
       rdf:about="http://www.semwebtech.org/util/local/exist-wrapper/general">
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/query-engine#evaluate-query"/>
     <provided-at rdf:resource="http://www.semwebtech.org/util/local/exist-wrapper/general"/>
     <input>element request</input>
     <variables>1</variables>
    </TaskDescription>
   </has-task-description>
  </mars:QueryService>
 </mars:is-implemented-by>
</mars:QueryLanguage>
```

```
<mars:ProcessAlgebra
  rdf:about="http://www.semwebtech.org/languages/2006/ccs#">
 <mars:name>CCS</mars:name>
 <mars:shortname>CCS</mars:shortname>
 <mars:hasRDF rdf:resource="http://www.semwebtech.org/languages/2006/ccs.rdf"/>
 <mars:hasDTD rdf:resource="http://www.semwebtech.org/languages/2006/ccs.dtd"/>
 <mars:hasStripedDTD
    rdf:resource="http://www.semwebtech.org/languages/2006/ccs-striped.dtd"/>
 <mars:hasStrippedDTD rdf:resource="http://www.semwebtech.org/languages/2006/ccs.dtd"/>
 <mars:is-implemented-by>
  <mars:ActionService
       rdf:about="http://www.semwebtech.org/services/2007/ccs"
       xml:base="http://www.semwebtech.org/services/2007/ccs/">
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/action-service#execute-action"/>
     <provided-at rdf:resource="????"/>
     <Reply-To>????</Reply-To>
     <Subject>????</Subject>
     <input>????</input>
     <variables>*</variables>
    </TaskDescription>
   </has-task-description>
  </mars:ActionService>
 </mars:is-implemented-by>
</mars:ProcessAlgebra>

<!-- contact the r3 prototype at Lisbon -->
<mars:Language
  rdf:about="http://rewerse.net/I5/NS/2006/r3#">
 <mars:isa rdf:resource="&mars;#AtomicEventFormalism"/>
 <mars:isa rdf:resource="&mars;#EventAlgebra"/>
 <mars:isa rdf:resource="&mars;#QueryLanguage"/>
 <mars:isa rdf:resource="&mars;#ActionLanguage"/>
 <mars:name>The r3 engine for everything</mars:name>
 <mars:shortname>r3</mars:shortname>
 <mars:hasRDF rdf:resource="http://rewerse.net/I5/NS/2006/r3#"/>
 <mars:is-implemented-by>
  <mars:Service rdf:about="http://di150.di.fct.unl.pt:15080/r3/service/eca_engine">
   <mars:isa rdf:resource="&mars;#AtomicEventMatcher"/>
   <mars:isa rdf:resource="&mars;#CompositeEventDetectionEngine"/>
   <mars:isa rdf:resource="&mars;#QueryService"/>
   <!-- r3: input element name is always the same as the fragment part
            of the URI given at describes-task -->
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/eds#register-event-pattern"/>
     <provided-at rdf:resource="http://di150.di.fct.unl.pt:15080/r3/service/eca_engine"/>
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
     <input>element register-event-pattern</input>
    </TaskDescription>
   </has-task-description>
```

```
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/eds#deregister-event-pattern"/>
     <provided-at rdf:resource="http://di150.di.fct.unl.pt:15080/r3/service/eca_engine"/>
     <Reply-To>n.a.</Reply-To>
      <Subject>body</Subject>
      <input>element deregister-event-pattern</input>
    </TaskDescription>
   </has-task-description>
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/query-engine#evaluate-query"/>
     <provided-at rdf:resource="http://di150.di.fct.unl.pt:15080/r3/service/eca_engine"/>
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
     <input>element evaluate-query</input>
     <variables>*</variables>
    </TaskDescription>
   </has-task-description>
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/query-engine#evaluate-test"/>
     <provided-at rdf:resource="http://di150.di.fct.unl.pt:15080/r3/service/eca_engine"/>
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
     <input>element evaluate-test</input>
     <variables>*</variables>
    </TaskDescription>
   </has-task-description>
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/action-service#execute-action"/>
     <provided-at rdf:resource="http://di150.di.fct.unl.pt:15080/r3/service/eca_engine"/>
     <Reply-To>body</Reply-To>
     <Subject>body</Subject>
     <input>element execute-action</input>
     <variables>*</variables>
    </TaskDescription>
   </has-task-description>
  </mars:Service>
 </mars:is-implemented-by>
</mars:Language>

<!-- when contacted by the r3 prototype at Lisbon -->
<mars:Language
  rdf:about="http://rewerse.net/I5/NS/2006/r3/r3tomars#">
 <mars:name>r3tomars</mars:name>
 <mars:shortname>r3tomars</mars:shortname>
 <mars:is-implemented-by>
  <mars:Service rdf:about="http://di150.di.fct.unl.pt:15080/r3-mars/service">
   <!-- this service provides only tasks that process responses -->
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/eca-service#receive-detected-event"/>
```

```
     <provided-at rdf:resource="http://di150.di.fct.unl.pt:15080/r3-mars/service"/>
     <Reply-To>n.a.</Reply-To>
     <Subject>body</Subject>
     <input>element receive-detected-event</input>
     <variables>*</variables>
    </TaskDescription>
   </has-task-description>
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/eca-service#receive-query-answer"/>
     <provided-at rdf:resource="http://di150.di.fct.unl.pt:15080/r3-mars/service"/>
     <Reply-To>n.a.</Reply-To>
     <Subject>body</Subject>
     <input>element receive-query-answer</input>
     <variables>*</variables>
    </TaskDescription>
   </has-task-description>
  </mars:Service>
 </mars:is-implemented-by>
</mars:Language>

<mars:Language
  rdf:about="http://www.semwebtech.org/mars/2006/mars#">
 <mars:name>MARS</mars:name>
 <mars:shortname>MARS</mars:shortname>
</mars:Language>

<mars:Domain
   rdf:about="http://www.semwebtech.org/mars/2006/mars#">
 <mars:name>MARS Built-Ins</mars:name>
 <mars:is-implemented-by>
  <mars:Service
       rdf:about="http://www.semwebtech.org/services/2007/eca-engine/"
       xml:base="http://www.semwebtech.org/services/2007/eca-engine/">
   <has-task-description>
    <TaskDescription>
     <describes-task rdf:resource="&mars;/action-service#execute-action"/>
     <provided-at rdf:resource="mars-builtins"/>
     <input>item</input>
     <variables>no</variables>
    </TaskDescription>
   </has-task-description>
  </mars:Service>
 </mars:is-implemented-by>
</mars:Domain>

<mars:Domain rdf:about="http://www.semwebtech.org/domains/2006/travel#">
 <mars:name>Traveling</mars:name>
</mars:Domain>

<mars:Domain rdf:about="http://www.semwebtech.org/domains/2006/uni#">
 <mars:name>University</mars:name>
</mars:Domain>
```

```
<mars:Domain rdf:about="http://foo.nop"/>

<!-- 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) -->

<mars:DomainBroker
   rdf:about="http://www.semwebtech.org/services/2007/domain-broker">
  <mars:name>All Domains Broker</mars:name>
  <has-task-description>
   <TaskDescription>
    <describes-task rdf:resource="&mars;/domain-broker#register-for-event"/>
    <provided-at rdf:resource="http://www.semwebtech.org/services/2007/domain-broker/register-for-ev
    <Reply-To>body</Reply-To>
    <Subject>n.a.</Subject>
    <input>element register</input>
    <variables>no</variables>
   </TaskDescription>
  </has-task-description>
  <has-task-description>
   <TaskDescription>
    <describes-task rdf:resource="&mars;/domain-broker#deregister-for-event"/>
    <provided-at rdf:resource="http://www.semwebtech.org/services/2007/domain-broker/deregister-for-
    <Reply-To>body</Reply-To>
    <Subject>n.a.</Subject>
    <input>element deregister</input>
    <variables>no</variables>
   </TaskDescription>
  </has-task-description>
  <has-task-description>
   <TaskDescription>
    <describes-task rdf:resource="&mars;/domain-broker#handle-event"/>
    <provided-at rdf:resource="http://www.semwebtech.org/services/2007/domain-broker/receive-event"/
    <input>item</input>
    <variables>no</variables>
   </TaskDescription>
  </has-task-description>
  <has-task-description>
   <TaskDescription>
    <describes-task rdf:resource="&mars;/domain-broker#execute-action"/>
    <!--
    <provided-at rdf:resource="http://www.semwebtech.org/domains/travel/rdfserver/actions"/>
    -->
    <provided-at rdf:resource="http://www.semwebtech.org/services/2007/domain-broker/execute-action"
    <input>item</input>
    <variables>no</variables>
   </TaskDescription>
  </has-task-description>
  <mars:supports-domain>http://www.semwebtech.org/domains/2006/travel#</mars:supports-domain>
  <mars:supports-domain>http://www.semwebtech.org/domains/2006/uni#</mars:supports-domain>
  <mars:supports-domain>http://foo.nop</mars:supports-domain>
   <!-- to be extended -->
</mars:DomainBroker>
```

```
<mars:ApplicationNode rdf:about="http://www.semwebtech.org/nodes/2007/mars-railway">
  <mars:name>MARS Worldwide Railway Services</mars:name>
  <mars:uses-domain>http://www.semwebtech.org/domains/2006/travel</mars:uses-domain>
  <!-- list of all names in the domain (events, actions, concepts, properties)
      that are supported by that application node -->
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#RailStation"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#RailConnection"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#from"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#to"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#duration"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#distance"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#railDelay"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#railDelayed"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#railBook"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#railBooked"/>
</mars:ApplicationNode>

<mars:ApplicationNode rdf:about="http://www.semwebtech.org/nodes/2007/mars-airlines">
  <mars:name>MARS Worldwide Airline Services</mars:name>
  <mars:uses-domain>http://www.semwebtech.org/domains/2006/travel</mars:uses-domain>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#Airport"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#FlightConnection"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#from"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#to"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#duration"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#distance"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#flightDelay"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#flightDelayed"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#flightBook"/>
  <mars:supports rdf:resource="http://www.semwebtech.org/domains/2006/travel#flightBooked"/>
</mars:ApplicationNode>
</rdf:RDF>
```

The following SPARQL query illustrates the combination of the RDF information described in this chapter.

```
# call
# jena -q -pellet -il RDF/XML -qf lsr-query.sparql

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 mars: <http://www.semwebtech.org/mars/2006/mars#>
PREFIX lsr: <http://www.semwebtech.org/mars/2006/lsr#>

SELECT ?L ?S ?T ?URL
FROM <file:mars-language-and-service-types.rdf>
FROM <file:services-ontology.rdf>
FROM <file:lsr.rdf>
WHERE {
 ?L rdf:type mars:Language .
 ?L mars:is-implemented-by ?S .
```

```
?S lsr:has-task-description ?TD .
?TD lsr:describes-task ?T .
?TD lsr:provided-at ?URL
}
```

9.5.4 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 integrating 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 9.3.

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-MARS-aware services).

GRH: Interface

- most generic call situation: requests from the ECA engine (component in XML, "name" of the task to be executed with this fragment). Then, the GRH may slect *any* service that supports the respective language. It returns the URI of the service.

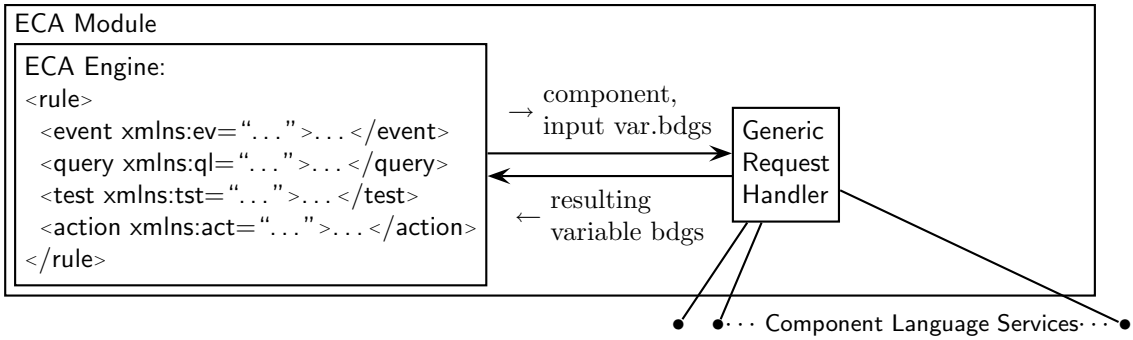


Figure 9.3: Architecture of an ECA Module with own GRH

- note that 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 that case, the GRH is called with a concrete service URI, a content (or only the subject referred to) and optionally variable bindings.
- out to below (autonomous services): requests to MARS-aware and non-MARS-aware (HTTP) services,
- in from below: results in the <logvars:answers> format from MARS-aware component services and unwrapped results from non-MARS-aware services (HTTP answers).

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 Section 5. 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 9.4 illustrates this by an ECA engine and an action engine (CCS).

Example 47 *Consider again the sample rule given in Section 5.8.5. 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 query engine which in turn contacts a domain broker that supports the travel domain. The query engine returns a set of extended answer tuples (flight, date, booking, name) to the ECA engine. 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.*

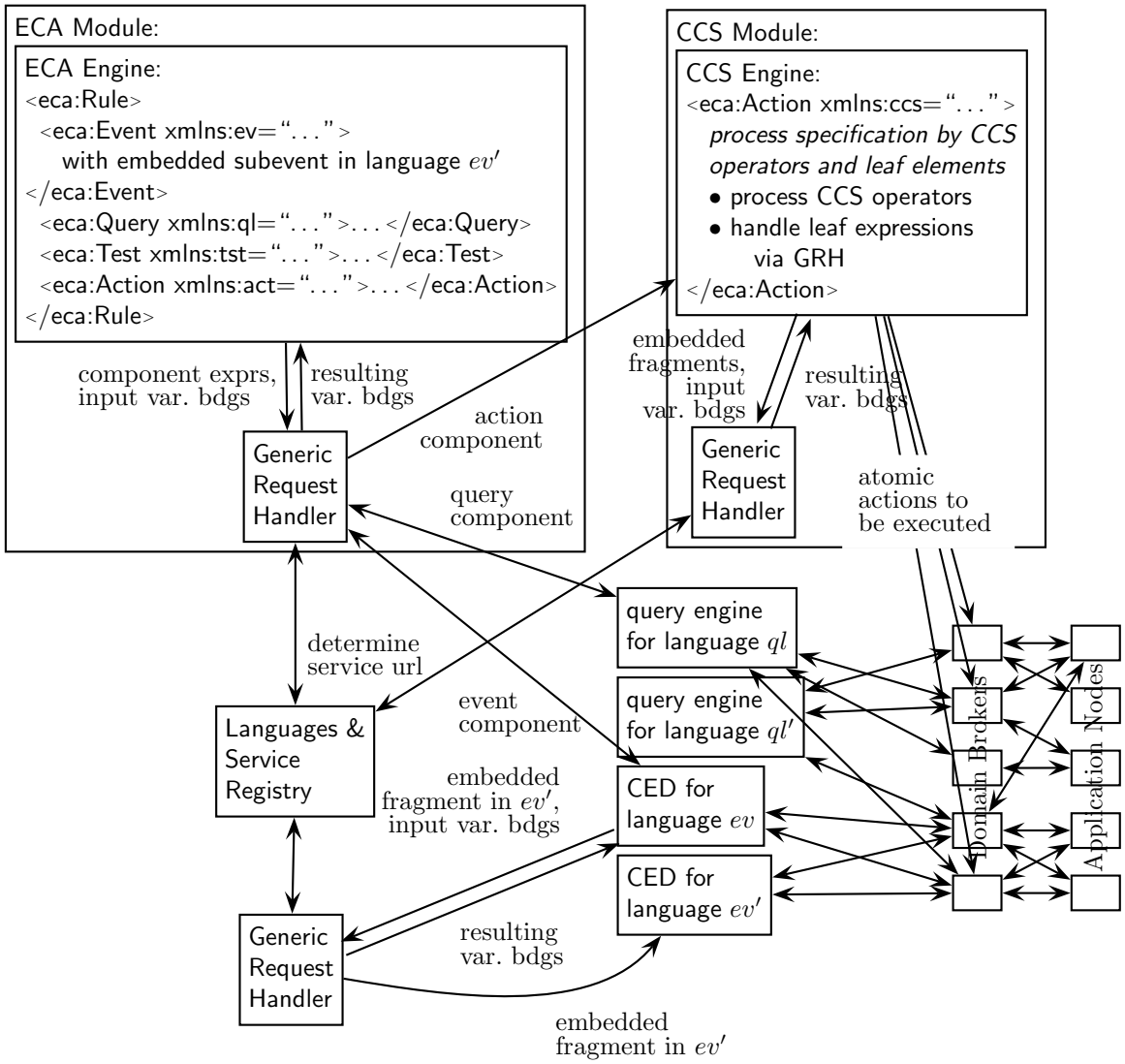


Figure 9.4: Global Coooperation using GRHs

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. When a client's request that used communication bookkeeping fails, the client should ask the LSR if the service description of that service changed.

9.6 Contacting Domain Services

As described in Section 8, the communication with domain services is done via Domain Brokers. Thus, Atomic Event Matchers and Query Engines do not contact the domain nodes directly, but communicate with an appropriate Domain Broker. For that, *Domain Service Registries (DSRs)* are used. For the current prototype, the DSR functionality is integrated withing the LSR.

9.7 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 10

Implementation and Prototype: The XML Level

This chapter describes the current state of the XML level of the prototype implementation at Göttingen University (the RDF level will be added stepwise to the public access). A demo+testing version on the XML level is available since early 2006 at `http://www.semwebtech.org/eca/frontend`.
As pointed out in the previous section about global architectures, the implementation of the MARS 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 of 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.

10.1 The ECA Engine

The prototype is implemented incrementally. The first module was an ECA Engine prototype whose first version has been developed using very simple component services [65] (first presentation in late 2005).

The ECA engine implements the functionality described in Sections 4.1-4.4: 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 of 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 9.5.4) 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-MARS-aware services). The architecture is as shown in Figure 9.3.

ECA: Functionality

- ECA semantics: handling variable bindings, joins; available also for other modules in the package eca-common,
- registration and storage of rules,
- control flow in ECA rules.

ECA: Interface

The ECA engine provides tasks according to the MARS Language and Services Ontology (cf. Section 9.3.3). The actual interface data is accessible in the LSR (cf. Section 9.5.1).

- Reference URL: `http://www.semwebtech.org/services/2007/eca-engine`
- HTTP in from above: registration and deregistration of rules (at `./register` and `./deregister`). Messages Formats:
  - Registration:

    ```
    <register>
      <subject>x://rule7</subject>
    </register>
    ```

    returns a URI (if the rule already had an URI, return it)
  - Deregistration:

    ```
    <deregister>
     uri
    </deregister>
    ```

- HTTP in from below: asynchronous notification about occurrences of the specified event patterns. (at `./notifications`)
  Message Format:

  ```
  <response>
    <subject>rule2#event</subject>
    <log:answer xmlns:log="http://www.semwebtech.org/languages/2006/logic#">
      ...
    </log:answer>
  </response>
  ```

- Java out to below: requests to GRH as instances of *class GRHRequest*: components in XML Markup of the individual languages, and which task should be executed.
- in from below: answers in the <logvars:variable-bindings> format. The answers can be synchronous (answers to HTTP POST), or asynchronous (HTTP POST to `./query-answer`).

Additional Functionality

Events can be sent to the ECA engine at `./incoming-events`. By default, they are then forwarded to the domain broker. This can be changed into hardcoded forwarding to specific AEMs if a domain broker is not needed/not available.

ECA: Classes

- ECAEngine, ECAEngineServlet, RuleManager, RuleInstance, IRuleDao (and implementations RuleFileDao, RuleExistDao);
- Plugins: XPathEngine and XQueryEngine (wrapping saxon).

ECA Common: Rules

- Rule (structural level), RuleComponent (and subclasses), LanguageElement.

ECA Common: GRH & Friends

- GenericRequestHandler
- NativeLanguageProcessor: deals with requests to services that support the MARS communication (and are listed in the LSR),
- OpaqueHTTPProcessor: deals with requests to services that are dealt by plain HTTP GET/POST.
- ServiceRegistry: provides the actual functionality to query the LSR RDF/XML database (the current version works on the XML level).

ECA Common: Utilities

- VariableBindings, Tuple, VariableBinding (and subclasses);
- XMLUtils, HTTPHelper, Logger, some more;
- Messages: Answer, Request, GRHRequest, some more.

10.2 Composite Event Detection and Atomic Event Matchers

The following modules for event detection according to the general interfaces have been developed:

- a SNOOP service (2005/06, [67]);
- two AEM services: XML-QL style matching (2005/06, [67]), and XPath-based mini-queries (ad hoc, 2006);
- RDF level OWLQ (2007/08, cf. Section 13).

A first integrated CED+AEM: Snoopy. A CED for a language closely related to the SNOOP event algebra [25] of the "Sentinel" active database system has been implemented in [67]. The CED uses the SNOOP XML markup presented in Section 5.3.5.

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 5.3.2.1. 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.

10.2.1 Snoopy CED

The part that implements the SNOOP language is installed as a separate service.

- Reference URL: `http://www.semwebtech.org/services/2007/snoopy`
- HTTP in from above: registration and deregistration of event specifications (at `./register` and `./deregister`).
  Message Formats:

```
<register>
 <reply-to>http://.../eca-engine/notifications</reply-to>
 <subject>x://rule4/event</subject>
  <snoop:And xmlns:snoop="http://www.semwebtech.org/languages/2006/snoopy#">
    :
  </snoop:And>
</register>
```

  and

```
<deregister>
 <subject>x://rule4/event</subject>
</deregister>
```

- HTTP in from below: asynchronous notification about occurrences of the specified event patterns (at `./notifications`)
  Message Format:

```
<response>
  <subject>x://rule4#event</subject>
  <log:answer xmlns:log="http://www.semwebtech.org/languages/2006/logic#">
    :
  </log:answer>
</response>
```

Internal

Registered CESs are submitted to the XSL transformer that creates the automaton:

```
<register>
  <snoop:And xmlns:snoop="http://www.semwebtech.org/languages/2006/snoopy#">
  :
  </snoop:And>
</register>
```

Atomic Patterns in an AES language to be registered are written into a separate file by the stylesheet which is then processed separately. For each AES, a message to an appropriate AEM is sent. For being able to embed arbitrary AES languages, Snoopy uses the same technology as the ECA engine: an instance of the GRH that accesses the LSR.

10.2.2 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.

- Language: XML patterns. Variables are enclosed in {`$varname`}.
- Reference URL: `http://www.semwebtech.org/services/2007/aem-xmlql`
- Registration (from ECA or CED; at `./register`):

  ```
  <register>
    <reply-to> ... </reply-to>
    <subject>x://rule3/event/_2</subject>
    <travel:Booking xmlns:travel="http://www.travel.nop"
          person="{$Person}" to="{$To}" />
  </register>
  ```

- Deregistration (from ECA or CED; at `./register`): uses the ID (i.e., URI) of the event spec:

  ```
  <deregister>
    <subject>x://rule3/event/_2</subject>
  </deregister>
  ```

- Events in from "below" (i.e., outside or domain broker; at `./events`).
  Message Format: Events as XML fragments.

Internal

The (XSLT-based) XPath matching engine keeps a patterns file of the following form:

```
<patterns>
  <info>
    <reply-to>http://www.semwebtech.org/services/2007/eca-engine</reply-to>
    <subject>x://rule1#event</subject>
    <registered-at>http://www.semwebtech.org/services/2007/domain-broker</registered-at>
    <pattern>
      <travel:Booking
         xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
         person="{$Person}" to="{$To}" />
    </pattern>
  </info>
</patterns>
```

10.2.3 XPath-based AEM

Another AEM using the XPath-navigation based formalism described in Section 5.3.2.2 has also been implemented as an XSL stylesheet.
The AES-XPath language is as follows (see registration example below):

- a surrounding element that is the "element hull" of the expected XML element.
- aes-xpath:bind-variable elements that have a name (variable to be bound) attribute and a select (XPath expression as in XSLT) attribute that selects a node from the element.
- Reference URL: `http://www.semwebtech.org/services/2007/aem-xpath`
- Registration (from ECA or CED; at `./register`):

```
<register>
  <reply-to> ... </reply-to>
  <subject>x://rule7/event/_2</subject>
  <travel:Booking xmlns:travel="http://www.travel.nop"
      xmlns:xpm="http://www.semwebtech.org/languages/2006/aes-xpath#">
    <xpm:bind-variable name="Person" select="$event/@person" />
    <xpm:bind-variable name="To" select="$event/@to" />
  </travel:Booking>
</register>
```

- Deregistration (from ECA or CED; at `./register`): uses the ID (i.e., URI) of the event spec:

```
<deregister>
  <subject>x://rule7/event/_2</subject>
</deregister>
```

- Events in from "below" (i.e., outside or domain broker; at `./events`).
  Message Format: Events as XML fragments.

Internal

The (XSLT-based) XPath matching engine keeps a patterns file of the following form:

```
<em:event-matcher
   xmlns:em="http://www.semwebtech.org/languages/2006/aes-xpath#">
 <em:registered>
  <em:event reply-to="http://www.semwebtech.org/services/2007/eca-engine"
     eds-event-id="x://rule5#event"
     registered-at="http://www.semwebtech.org/services/2007/domain-broker">
   <travel:Booking
     xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
     xmlns:xpm="http://www.semwebtech.org/languages/2006/aes-xpath#">
     <xpm:bind-variable name="Person" select="$event/@person" />
     <xpm:bind-variable name="To" select="$event/@to" />
   </travel:Booking>
  </em:event>
 </em:registered>
</em:event-matcher>
```

10.3 Queries and Updates

On the XML level, existing XML query languages are used by *opaque* query components; in the same way SPARQL can be treated as an opaque query language for RDF. Existing implementations of these languages have been wrapped according to the general interface. Pure language implementations for stating queries on independent XML instances are distinguished from XML databases.

Saxon. Saxon [40] is an XQuery implementation that allows to state queries against XML sources on the Web. A MARS-aware 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 [29] is supported that embeds XPath into SQL. Updates are implemented via transformations; thus triggers on the XML level as described in Section 7.2.2 are currently not supported.

eXist. eXist [31] 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 [68, 45]. A MARS-aware wrapper around an eXist database that supports queries and XUpdate has been implemented. On the XML level, an eXist database is used as a substitute stub for queries against XML domain nodes.

Jena/SPARQL. Jena [37] is an open-source framework that provides functionality for handling RDF data; also OWL reasoners can be integrated. Similar to XQuery, SPARQL queries can be evaluated against a local knowledge base (select ... where ...), or against a remote knowledge (select ... from ... where ...).

Opaque SPARQL would have been the canonic next step after the above opaque XPath/XQuery handling. But, as SPARQL itself is not ontology-based, it is not supported by an own wrapper. We come back to queries against RDF and OWL databases on the RDF level, especially, with OWLQ in Section 13.3.

10.4 Handling Opaque Fragments

As discussed above, the conceptually simplest step (making a first ECA prototype run) is to investigate opaque fragments for queries and actions. Since many repositories in the Web are non-semantic, the handling of opaque queries in XPath or XQuery is also required for the full MARS Framework.

10.4.1 Communicating with Primitive Services

When using opaque components, the most basic assumption is that the service is not aware of anything in the MARS 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 4.1.2).
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, 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 basic HTTP GET Web Services. Applications are here mainly application services, but also generic (query) language services can be used.

10.4.2 MARS-Aware Wrappers

Non-MARS-aware services can be wrapped into MARS-aware ones. With the latter, communication can take place as described in Section 4.2. 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 53, MARS-aware wrappers for non-MARS-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 4.2.2 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 4.2.3.

10.5 Actions and Processes

10.5.1 Application Domain Actions

Actions in the application domain are full members of the data model, i.e., (volatile) instances of the class mars:Action. A node of the SWAN architecture (cf. Section 7.3) that "receives" an action (at the appropriate HTTP address) will execute it. Actions are submitted as XML (or RDF/XML) fragments. Rules can trigger actions in two ways:

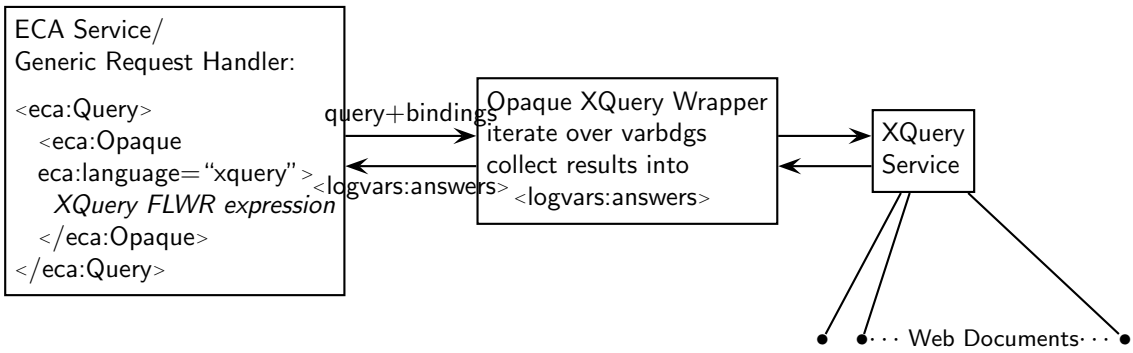


Figure 10.1: Architecture: Generic Opaque Prototype Pattern (e.g., XQuery)

- addressing them directly to a domain node,
- submitting them to a domain broker (cf. Section 8). The broker then submits them to appropriate domain nodes for actual execution.

10.5.2 Raising Events by Opaque Atomic Actions

As described in Section 3.6 for the language binding for opaque components, an explicit service URL 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 uni:subject="Databases" uni:name="John Doe"/>
  </eca:Opaque>
</eca:Action>
```

"sends" the event representation <uni:register uni:subject="Databases" uni:name="John Doe"/> to the exam administration system of the CS department.

10.5.3 MARS builtin: mars:raise-event

Event raising is provided as a MARS builtin action mars:raise-event that takes the event als XML pattern (variables embedded in XQuery style by {$varname}). Raising events is implemented by sending an HTTP message with the raised event to the domain brokers that support the respective domain.

```
<eca:Rule>
    :
  <eca:Action>
    <mars:raise-event xmlns:mars="http://www.semwebtech.org/mars/2006/mars#">
      <eca:has-input-variable eca:name="varname"/>
        :
      event as XML fragment
      variables embedded by {$varname}
    </mars:raise-event>
  </eca:Action>
</eca:Rule>
```

the *event as XML fragment* is forwarded (=raised) to all event brokers known to the ECA engine.

10.5.4 Composite Actions and Processes

A CCS-based action engine with the functionality described in Section 5.8 is under implementation. It shares the GRH functionality with the ECA engine.

10.6 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.

SQL: The use of the Oracle Rule Engine for SQL-based node (sample scenario: car rental company) has been evaluated in [33].

XML: An eXist database is used as a substitute stub for queries and updates against XML domain nodes.

RDF: The functionality of the SWAN node architecture described in Section 7 provides for a "native" Semantic Web node architecture consisting of an RDF/OWL node that additionally emits events and executes actions of the domain ontology.

10.7 Infrastructure: LSR and GRH

The prototype prototype uses an LSR as an XML/RDF file (see Section 9.5.3) which can be found at `http://www.semwebtech.org/mars/2006/lsr.rdf`. In the current prototype, the access and (XML) processing of the LSR is implemented by the Java class *ServiceRegistry*. The GRH functionality is implemented by the Java class *GenericRequestHandler*. Both classes are available in the eca-common package; they have been successfully used for integrating MARS with r^3 (see Section 17.2).

The *ServiceRegistry* class and the LSR does not provide registration functionality; currently the LSR is maintaind manually. The LSR functionality can be separated into an own service without problems.

10.8 Infrastructure: Domain Broker

A domain Broker has been developed in [44], and has been integrated functionally [9]. Further tests and developments first require to have nontrivial domain nodes.

10.9 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) atomic events to the Domain Broker; they are forwarded to the registered AEMS where they are be processed.
- 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.

Chapter 11

The RDF Level: Language Elements and their Instances as Resources

Rules on the semantic level, i.e., RDF or OWL, lift ECA functionality wrt. two (independent) aspects: first, the events, conditions and actions refer to the ontology level as described above. On an even higher level, the above rule ontology and event, condition, and action subontologies regard rules themselves as objects of the Semantic Web. Together with the languages and their processors, this leads directly to a resource-based approach: every rule, rule component, event, subevent etc. becomes a resource, which is related to a language which in turn is related to other resources.
Describing rules in RDF will allow to reason about rules. This will support several things:

- validation and support for execution (e.g., use of variables; cf. Section 15.2);
- actual reasoning about the behavior of a node, including correctness issues cf. Section 16.2;
- expressing rules in abstract terms instead of wrt. concrete languages. The services can then e.g. choose which concrete languages support the expressiveness required by the rule's components.

Events and Actions on the RDF Level. For the RDF/OWL level, we assume that not only the data itself is in RDF, but also events and actions are given as XML/RDF fragments (using the same URIs for entities and properties as in the static data).

11.1 Example and Motivation

We give first an impression what rules in RDF look like. The examples use the notions already known from the XML level, and expand them to the RDF level. Details will be discussed afterwards.

Example 48 *Figure 11.1 shows an excerpt of the rule given in Example 31, now in full RDF: "If a flight is first delayed and then canceled (note: use of a join variable), then ..."*

```
<!DOCTYPE rdf:RDF [
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <!ENTITY owlq "http://www.semwebtech.org/languages/2006/owlq#">
```

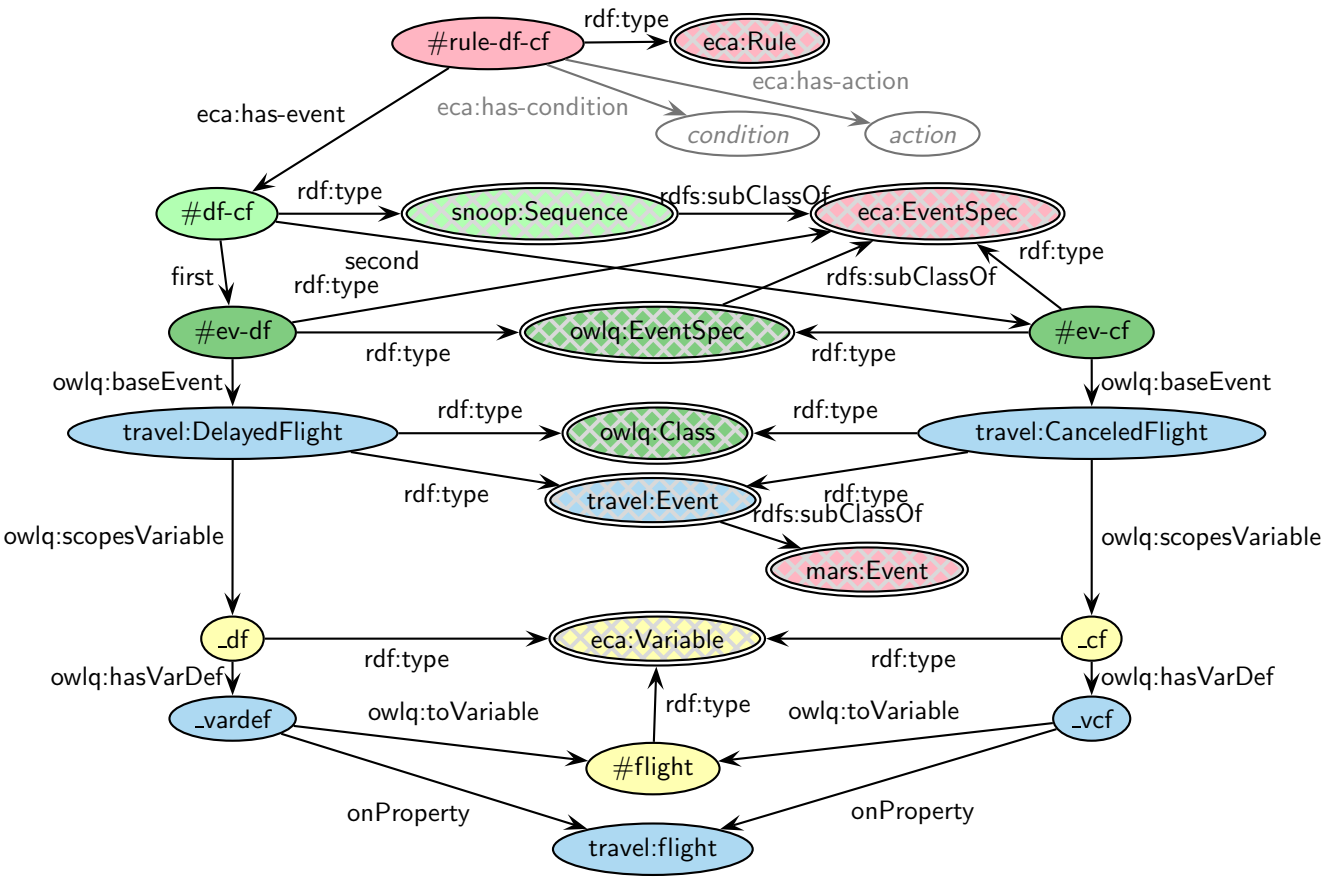


Figure 11.1: Example Rule and Event Component as Resources

```
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <!ENTITY travel "http://www.semwebtech.org/domains/2006/travel#"> ]>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:eca="http://www.semwebtech.org/languages/2006/eca-ml#"
   xmlns:snoop="http://www.semwebtech.org/languages/2006/snoopy#"
   xmlns:owlq="http://www.semwebtech.org/languages/2006/owlq#"
   xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
   xml:base="foo:rule">

<eca:Rule rdf:ID="rule-df-cf">
 <eca:uses-variable rdf:resource="#flight"/>
 <eca:has-event>
  <snoop:Sequence>
   <snoop:first>
    <owlq:EventSpec rdf:ID="ev-df">
     <owlq:baseEvent rdf:resource="&travel;DelayedFlight"/>
     <owlq:scopesVariable>
      <owlq:Variable>
       <owlq:hasVariableDefinition>
        <owlq:VariableDefinition>
         <owlq:onProperty rdf:resource="&travel;flight"/>
```

```
        <owlq:toVariable rdf:resource="#flight"/>
       <owlq:VariableDefinition>
      </owlq:hasVariableDefinition>
     </owlq:Variable>
    </owlq:scopesVariable>
   </owlq:EventSpec>
  </snoop:first>
  <snoop:second>
   <owlq:EventSpec rdf:ID="ev-cf">
    <owlq:baseEvent rdf:resource="&travel;CanceledFlight"/>
    <owlq:scopesVariable>
     <owlq:Variable>
      <owlq:hasVariableDefinition>
       <owlq:VariableDefinition>
        <owlq:onProperty rdf:resource="&travel;flight"/>
        <owlq:toVariable rdf:resource="#flight"/>
       <owlq:VariableDefinition>
      </owlq:hasVariableDefinition>
     </owlq:Variable>
    </owlq:scopesVariable>
   </owlq:EventSpec>
  </snoop:second>
 </snoop:Sequence>
</eca:has-event>
<!-- ... query and action ... -->
</eca:Rule>
```

11.2 Ontology of Expressions

Languages in general define *expressions* and their semantics. The expressions' ontology is divided into three parts:

General Expressions are either CompositeExpressions, AtomicExpressions (wrt. that language; this can be composite expressions according to an *embedded* language), or Variables.

Algebraic expressions are a special subclass of expressions that adheres to a term structure, i.e., in addition to subexpressions they are related to an operator.

Variables: Expressions *use* variables in certain ways. Note that variables are actually *resources* that are referenced. Usually, they have URIs relative to the rule; this allows e.g. reasoning about dependencies between rules that trigger each other.

The below ontologies provide the required concepts. They are included e.g. from the ECA Ontology in Chapter 12, from the Snoopy Ontology in Section 14.1 and from the CCS Ontology in Section 14.2.

11.2.1 Subontology of General Expressions

General expressions cover algebraic and non-algebraic expressions.

```
<!DOCTYPE rdf:RDF [
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
  <!ENTITY mars "http://www.semwebtech.org/mars/2006/mars#"> ]>
<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:expr="http://www.semwebtech.org/mars/2006/expressions#"
   xml:base="http://www.semwebtech.org/mars/2006/expressions">

<owl:Ontology rdf:about="#"/>

<!-- general languages and expressions -->
<owl:Class rdf:about="Expression"/>

<!-- note: not functional since the same node
   can e.g. be an ECA-ML and a SNOOP-Node -->
<owl:Property rdf:about="#language">
 <rdfs:domain rdf:resource="#Expression"/>
 <rdfs:range rdf:resource="&mars;Language"/>
 <owl:maxCardinality>2</owl:maxCardinality>
</owl:Property>

<rdfs:Class rdf:about="#Expression">
 <owl:unionOf rdf:parseType="Collection">
  <rdfs:Class rdf:about="#Variable"/>
  <rdfs:Class rdf:about="#AtomicExpression">
   <owl:disjointWith rdf:resource="#Variable"/>
  </rdfs:Class>
  <rdfs:Class rdf:about="#CompositeExpression">
   <owl:disjointWith rdf:resource="#Variable"/>
   <owl:disjointWith rdf:resource="#AtomicExpression"/>
  </rdfs:Class>
 </owl:unionOf>
</rdfs:Class>

<rdf:Property rdf:about="#has-subexpression">
 <rdfs:domain rdf:resource="#CompositeExpression"/>
 <rdfs:range rdf:resource="#Expression"/>
</rdf:Property>
</rdf:RDF>
```

Note: expr:has-subexpression is most generic – it allows for ordered/unordered and named/unnamed relationships.

11.2.2 Subontology of Algebraic Expressions

```
<!DOCTYPE rdf:RDF [
  <!ENTITY mars "http://www.semwebtech.org/mars/2006/mars#"> ]>
<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:expr="http://www.semwebtech.org/mars/2006/expressions#"
   xml:base="http://www.semwebtech.org/mars/2006/expressions">

<owl:Ontology rdf:about="#"/>
```

```
<owl:Class rdf:about="#AlgebraicExpression">
 <rdfs:subClassOf rdf:resource="#Expression"/>
 <owl:equivalentClass>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#language"/>
   <owl:someValuesFrom rdf:resource="&mars;#AlgebraicLanguage"/>
  </owl:Restriction>
 </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:about="#Operator"/>
<owl:Class rdf:about="#Arities"/>
 <!-- should be defined as xsd:nonNegativeInteger + positive Infinity -->

<rdf:Property rdf:about="#arity">
 <rdfs:domain rdf:resource="#Operator"/>
 <rdfs:range rdf:resource="#Arities"/>
</rdf:Property>

<rdf:Property rdf:ID="provides-operator">
  <rdfs:domain rdf:resource="#AlgebraicLanguage"/>
  <rdfs:range rdf:resource="#Operator"/>
</rdf:Property>

<rdf:Property rdf:about="#arity">
 <rdfs:domain rdf:resource="#Operator"/>
 <rdfs:range rdf:resource="#Arities"/>
</rdf:Property>

<owl:Class rdf:about="#AlgebraicCompositeExpression"/>
 <owl:EquivalentClass>
  <owl:intersectionOf rdf:parseType="Collection">
   <owl:Class rdf:about="#AlgebraicExpression"/>
   <owl:Class rdf:about="#CompositeExpression"/>
  </owl:intersectionOf>
 </owl:EquivalentClass>
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#has-operator"/>
   <owl:cardinality>1</owl:cardinality>
  </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>

<owl:FunctionalProperty rdf:about="#has-operator"/>
 <rdfs:domain rdf:resource="#AlgebraicCompositeExpression"/>
 <rdfs:range rdf:resource="#Operator"/>
</owl:FunctionalProperty>
</rdf:RDF>
```

11.2.3 Subontology of Variable Usage

Variables are resources, usually associated with a rule, or in case of local variables, by a fragment (see examples in later sections). Expressions *use* variables in several ways. The use and handling of variables has already been discussed in Sections 4.1.5 and 4.3. The following relationships can hold between an expression and a variable:

uses-variable: a variable occurs in any way in (the context of) an expression.

free variables: a variable occurs free. Such variables belong to the "outer interface" of an expression.

bound variables: a variable occurs bound (i.e., inside in the scope of a quantifier). Such variables are not relevant to the outside of an expression.

positive variables, negative variables: used in logic-based frameworks (Datalog, F-Logic).

scoping of variables: a quantifier scopes a variable, that (usually) occurs *free* in a subexpression, and is then *bound* by the quantifier.

input variables: these variables must be bound when evaluating an expression. In logical frameworks, *negative* variables are input variables.

output variables: these variables are bound to values when evaluating an expression (e.g., OUT variables in procedures). If such a variable is already bound before evaluating an expression, it *may* be used as a join variable. For logical frameworks, it is recommended to use the notion of *positive* variables instead.

An additional side-effect that is used in many languages is to bind the results of subexpressions to variables (e.g., in F-Logic). They are to be considered as "used", "free" and "positive" by the expression.

The actual values of these properties for a given expression can usually be determined by an appropriate service that knows about the syntax of the language. Note that if such languages use an own ontology, an ontology mapping is required. This is usually defined by (deductive) rules.

```
<!DOCTYPE rdf:RDF [
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
  <!ENTITY mars "http://www.semwebtech.org/mars/2006/mars#">
  <!ENTITY expr "http://www.semwebtech.org/mars/2006/expressions#"> ]>
<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:expr="http://www.semwebtech.org/mars/2006/expressions#"
   xml:base="http://www.semwebtech.org/mars/2006/expressions">

<owl:Ontology rdf:about="#"/>

<!-- notions around variables -->
<!-- The class "Variable" has been defined above -->

<rdf:Property rdf:about="#uses-variable">
 <!-- variable occurs in some way in this expression -->
 <owl:inverseOf rdf:resource="#variable-occurs-in"/>
 <rdfs:domain rdf:resource="#Expression"/>
 <rdfs:range rdf:resource="#Variable"/>
```

```
</rdf:Property>

<rdf:Property rdf:about="#has-free-variable">
 <rdfs:domain rdf:resource="#Expression"/>
 <rdfs:range rdf:resource="#Variable"/>
 <rdfs:subPropertyOf rdf:resource="#uses-variable"/>
</rdf:Property>

<rdf:Property rdf:about="#has-bound-variable">
 <rdfs:domain rdf:resource="#Expression"/>
 <rdfs:range rdf:resource="#Variable"/>
 <rdfs:subPropertyOf rdf:resource="#uses-variable"/>
</rdf:Property>

<rdf:Property rdf:about="#has-positive-variable">
 <rdfs:domain rdf:resource="#Expression"/>
 <rdfs:range rdf:resource="#Variable"/>
 <rdfs:subPropertyOf rdf:resource="#uses-variable"/>
</rdf:Property>

<rdf:Property rdf:about="#has-negative-variable">
 <rdfs:domain rdf:resource="#Expression"/>
 <rdfs:range rdf:resource="#Variable"/>
 <rdfs:subPropertyOf rdf:resource="#has-input-variable"/>
 <rdfs:subPropertyOf rdf:resource="#uses-variable"/>
</rdf:Property>

<rdf:Property rdf:about="#scopes-variable">
 <rdfs:domain rdf:resource="#Expression"/>
 <rdfs:range rdf:resource="#Variable"/>
 <rdfs:subPropertyOf rdf:resource="#uses-variable"/>
</rdf:Property>

<rdf:Property rdf:about="#has-input-variable">
 <rdfs:domain rdf:resource="#Expression"/>
 <rdfs:range rdf:resource="#Variable"/>
 <rdfs:subPropertyOf rdf:resource="#uses-variable"/>
</rdf:Property>

<rdf:Property rdf:about="#has-output-variable">
 <!-- mainly for non-logical things -->
 <rdfs:domain rdf:resource="#Expression"/>
 <rdfs:range rdf:resource="#Variable"/>
 <rdfs:subPropertyOf rdf:resource="#uses-variable"/>
</rdf:Property>

<rdf:Property rdf:about="#bind-to-variable">
 <rdfs:domain rdf:resource="#Expression"/>
 <rdfs:range rdf:resource="#Variable"/>
 <rdfs:subPropertyOf rdf:resource="#uses-variable"/>
</rdf:Property>
</rdf:RDF>
```

Note that the mapping of variables to an XML Markup can be done in many ways as discussed in Section 4.3 for "native XML markup".

11.3 Language Ontologies vs. Markup Languages

MARS assumes that each language (e.g. ECA-ML, SNOOP, OWLQ, and CCS) defines an own, autonomous set of element names (DTD/XML Schema, XML world) and/or ontology (OWL world):

- the OWL ontology defines *all* notions of the language. This is used when representing things in RDF, e.g., for reasoning.
- an XML Markup is given by a DTD that consists of a subset of the ontology notions. The XML markup is a *stripped* version of a *specific* RDF/XML serialization of the RDF graph. It is used e.g., for data exchange between services and when services process a fragment. The (stripped) ECA-ML XML markup is e.g. the one that has been presented in Section 3.3.

Services may use each of the representations. Thus, there must be translations in both directions (cf. Section 11.4.1).

We will first discuss some generic aspects that lead the design of the OWL ontologies and the XML markup (and the relationship between both). Later we describe the ECA-ML OWL ontology in Section 12. The SNOOP and CCS ontologies follow then in Sections 14.1 and 14.2.

11.3.1 Design of Ontologies, Languages and Namespaces

When looking at existing language proposals, there is not yet a homogeneous handling of namespaces and ontology descriptions by OWL files behind them. For the RDF language, the design is as follows:

- The URL `http://www.w3.org/1999/02/22-rdf-syntax-ns#` is given as RDF namespace. There, an actual RDF/RDFS/OWL document (with name `22-rdf-syntax-ns.htm` can be found.
- The URL `http://www.w3.org/2000/01/rdf-schema#` returns a document `rdf-schema.htm`.

  Note that nevertheless, the URI `http://www.w3.org/1999/02/22-rdf-syntax-ns#Property` as a URL does not *address* the rdf:Property resource (which actually does not exist):

  ```
  <!-- excerpt from 22-rdf-syntax-ns.htm -->
  <rdfs:Class rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property">
    <rdfs:isDefinedBy rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
    <rdfs:label>Property</rdfs:label>
    <rdfs:comment>The class of RDF properties.</rdfs:comment>
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
  </rdfs:Class>
  ```

  If rdf:ID would have been used instead of rdf:about, the Class element rdfs:Property would have been the resource itself.
- Namespaces can either be defined as hash-namespace or as slash-namespace (cf. [63]). Most namespaces in MARS are hash-namespaces. Recall that XML markup allows only for qnames as identifiers within the namespaces.

The current decision is as follows:

- The namespace of ECA-ML is `http://www.semwebtech.org/languages/2006/eca-ml#`.

- This gives two possibilities:
  - `http://www.semwebtech.org/languages/2006/eca-ml` is a directory that contains the ECA-ML OWL file and the server delivers this file when the directory is accessed (independent whether with or without hash). This is currently done by linking the file to index.html.
  - `http://www.semwebtech.org/languages/2006/eca-ml` is the name of the OWL file. Then, no server mapping is required.

Analogously for the other language namespaces.

11.3.2 Language Identification

In the RDF graph, everything, including the language itself is a resource (cf. the MARS Languages & Services Ontology in Section 9.3.3). In the XML representation, the XML tree represents only the elements of the languages as a nested structure.

Language Elements and Element Names. While in the RDF world, every class is represented by an arbitrary URI, in the XML representation, only *namespace*:*elementname* where *elementname* is a qname (cf. XML terminology) is allowed. For that, it is strongly recommended that languages do not use hierarchical namespaces, but flat *hash* namespaces or *slash* namespaces.

Language Indicators. For both the RDF and the XML representation, each relevant item has one or more language indicators:

- for every class or property, the rdfs:isDefinedBy property refers to the language where it belongs to.

  It is strongly recommended that the value of rdfs:isDefinedBy is the prefix of the class or property up to the rightmost "#" or "/", e.g.

  <http://www.semwebtech.org/languages/2006/eca-ml#Rule>
  rdfs:isDefinedBy <http://www.semwebtech.org/languages/2006/eca-ml>.

  Then, the language can be guessed from the nodes rdf:type (which will be important when mapping nested expressions from RDF to XML in Section 11.4.1).

- for every expression, the mars:language property indicates to which language(s) it belongs (note that a same node "at the language border" can e.g. be both an ECA-ML and a SNOOP-Node).

  The property is usually derived by mars:language $\equiv$ rdf:type $\circ$ rdfs:isDefinedBy.

- In the XML representation, the language is encoded in the namespace (if the above recommendation to have flat namespaces is followed, the namespace of an element is the language URI).

11.3.3 (Sub)Components of Other Languages

Nested subexpressions belong to some other language or formalism (e.g., an Event Algebra or an AESL). They are also instances of concepts of the surrounding language (e.g., a snoopy:Sequence is an eca:Event).

A sample Snoop event specification graph is shown in Figure 11.2.

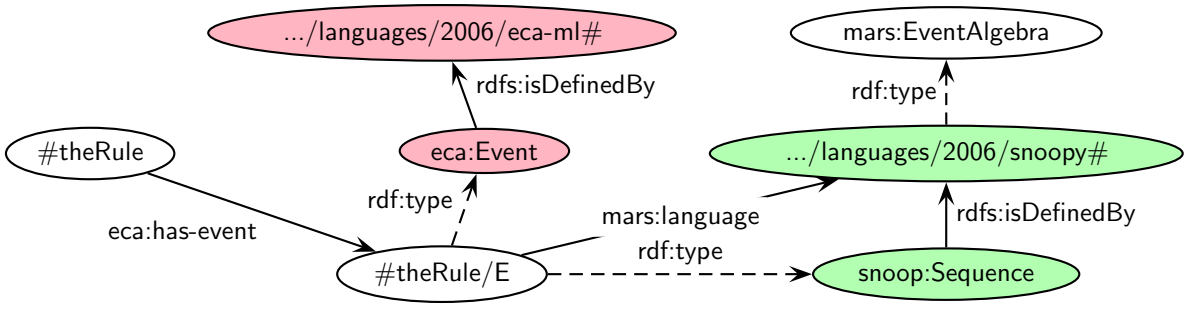


Figure 11.2: Structure of an Nested Sublanguage Element

11.3.4 Opaque (Sub)Components

The handling of opaque fragments differs from the above XML design since OWL allows for subclassing: eca:OpaqueQuery is defined as a subclass of eca:Query; similar for the other concepts. All are subclasses of eca:Opaque (which is not a subclass of any of the other concepts – being eca:Opaque does only tell the ECA engine that something has to be handled specifically). Opaque items have properties mars:language, or mars:uri and mars:method (e.g., saying that the query is answered by sending a HTTP GET request to uri *service-uri*). A sample OWL opaque query graph is shown in Figure 11.3.

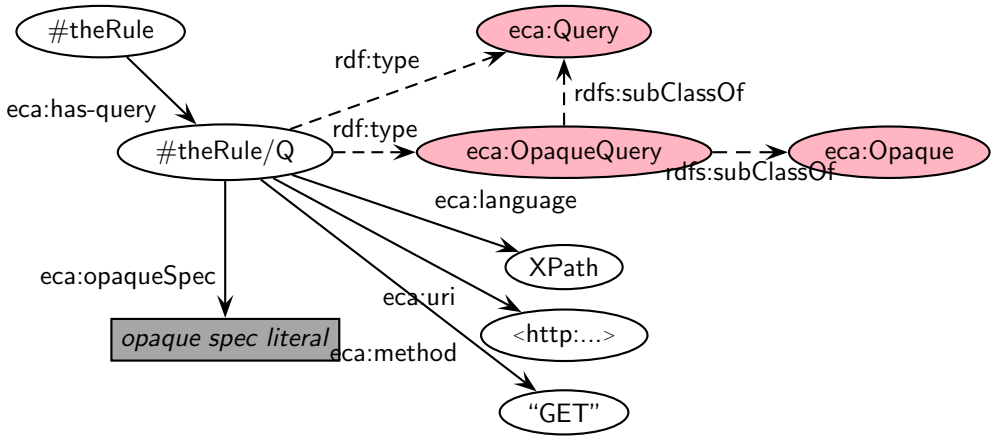


Figure 11.3: Structure of an Opaque Fragment

11.4 Processing ECA Rules in RDF and XML

The processing of ECA rules on the in XML level has been discussed in Chapters 4 – 10. For processing on the RDF level, two alternatives are possible

1. Working directly on the RDF graph. In this case only the URI of the root node of an expression has to be communicated to the processor.
2. Mapping RDF rules to XML.

(1) would require a shared repository, or services that accept RDF input. The existing services described above, and prospectively in the reality usually use XML markup of "their" languages. Even more, the notion of an "RDF rule graph" is not clearly delimited: where does the rule end and where do other fragments (domain definitions etc.) start? The processors would then also be

required to extract the actual components from the Web using the URIs. Thus, the alternative (2) has been chosen for MARS.

11.4.1 Fragment Translation between RDF and XML Representations

The RDF graph of a rule and its components is transformed into XML markup as illustrated in Figure 11.4. The XML markup is based on one of the RDF/XML serializations of the RDF graph:

- The RDF graph is serialized as striped RDF/XML: a depth-first traversal is applied that is controlled by a given *Striped DTD*. Note that this also "cuts" the relevant fragment that represents the rule out of the large Semantic Web RDF graph. This first step generates the underlying tree structure of the target XML markup.
- a tree projection is applied to the striped XML tree. It is based on a tree traversal whose output is a tree projection, controlled by a given *Stripped DTD*. This second step removes e.g. intermediate property subelements, rdf attributes, turns subelements into attributes, and generates ID/IDREF references.
- optionally, a final XSLT transformation can be applied to obtain a certain (in most cases legacy) markup of an already existing XML language.

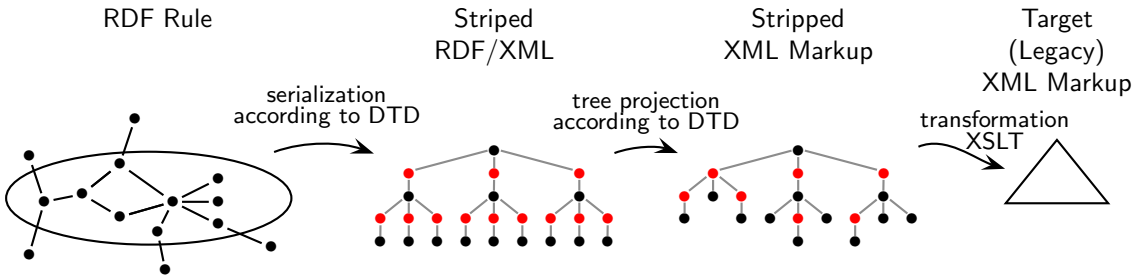


Figure 11.4: Obtaining XML markup from RDF

Since the goal of both DTDs is to use them for controlling a depth-first traversal of the RDF graph for RDF/XML serialization, these have a simpler structure (only sequence and alternative) than common DTDs. The normative DTD of the language is usually more detailed.

A Java class that transforms RDF into XML by using Striped/Stripped DTDs has been implemented in [69] and belongs to the MARS distribution.

The translation between the RDF and XML representations is furthermore concerned with glueing between languages/ontologies (i.e., how to compose a rule from ECA-ML, SNOOP and CCS notions).

11.4.2 Language Changes in XML and RDF Representations

One of the underlying concepts of MARS is that rules and expressions are made up from several, heterogeneous languages, and the overall design is open to integrate with nearly arbitrary component languages. For that, in the RDF and XML representation, there are *language changes* where the nested subexpressions are embedded. Elements (in XML) or notions (in RDF) representation must be glued together between bordering languages or ontologies.

Glue in RDF/OWL: Glue Axioms. In the RDF/OWL representation, the glue is provided by axioms that identify notions of the ontologies to be connected (using rdfs:subClassOf and rdfs:subPropertyOf). For instance, the SNOOP ontology's central class is snoopy:Event, whereas the ECA-ML ontology defines eca:Event. For that, *glue axioms* are added that state e.g.

snoopy:Event rdfs:subClassOf eca:Event ,
ccs:Action rdfs:subClassOf eca:Action ,
ccs:Event rdfs:subClassOf eca:Event .

Note that for the instance-based *processing*, these glue axioms are not necessary if the individual ontologies specify rdfs:domain and rdfs:range of the tree edges accordingly. By this, for the bordering nodes, class membership wrt. both bordering ontologies can be derived.

Example 49 (Derivation of class membership at language borders) *Consider the following RDF/N3 fragment:*

```
<foo:rule1> a eca:Rule;
  eca:has-event <foo:rule1/event>.
<foo:rule1/event> a snoopy:Sequence;
  snoopy:first <foo:rule1/event/first>;
  snoopy:second <foo:rule1/event/second>.
```

Together with the ECA-ML axiom

eca:has-event a rdf:Property; rdfs:domain eca:Rule; rdfs:range eca:Event.

the fact <*foo:rule1/event*> *a eca:Event can be derived. Thus, the node*

<foo:rule1/event> a eca:Event; a snoopy:Sequence.

represents a "shared" border object that is an eca:Event from the ECA-ML point of view, and a snoopy:Sequence from the SNOOP point of view as shown in Figure 11.2. Any processing can change there from the ECA-ML context to the SNOOP context.

Language Changes in pure XML: Embedding. In the XML representation, fragments of the embedded language are embedded into elements of the outside language. For instance, SNOOP expressions are embedded inside eca:Event elements as described in Section 5.3.6/Example 18:

```
<eca:Rule ... >
  <eca:Event>
    <snoopy:Sequence
            xmlns:snoopy="http://www.semwebtech.org/languages/2006/snoopy#">
      :
    </snoopy:Sequence>
  </eca:Event>
  :
</eca:Rule>
```

Having this kind of "double-sided" borders, the "outer part" ends with the <eca:Event> element, and the inner starts with the <snoopy:Sequence> element. For processing, only the inner subtree is submitted to the SNOOP processor.

Language Changes in Striped XML: Embedding or Explicit. The above XML fragment contains two *nested* elements that describe the event. Since the XML markup is intended to be obtained via a certain RDF/XML serialization (lead by the Striped DTD) and a tree projection (lead by the Stripped DTD), the striped format must also provide two nodes. This can be achieved by an explicit owl:sameAs edge.

```
<eca:Rule ... >
  <eca:has-vent>
    <eca:Event>
      <owl:sameAs>
```

```
        <snoopy:Sequence xmlns:snoopy="http://www.semwebtech.org/languages/2006/snoopy#">
          :
        </snoopy:Sequence>
      </owl:sameAs>
    </eca:Event>
  </eca:has-vent>
  :
</eca:Rule>
```

Here, the owl:sameAs has some similarity to *casting* in programming languages. For the mapping to the pure XML markup, the owl:sameAs is usually removed by the projection defined by the Striped DTD. The Striped and Stripped ECA-ML DTDs given in Section 12.5 will show how the DTDs are designed to enforce this effect.

Chapter 12

The ECA-ML Ontology

The ECA-ML language induces an ontology of rules. The complete OWL ontology can be found in Section 12.4. Obviously, the main classes are known from the already presented XML markup: Rule, Event (which actually means "event specification"), Query, Test, and Action (which actually means "action specification").

12.1 ECA-ML RDF Ontology

We give a short overview of the ontology; i.e., its classes, properties of the classes (with cardinalities) and some comments. The complete OWL specification is given in Section 12.4.

- eca:Rule: eca:has-event (1), eca:has-condition (0..n), eca:has-query (0..n), eca:has-test (0..n), eca:has-action (1). For a discussion about conditions vs. queries/tests, see Section 12.2.
- eca:Event: usually an expression of an event specification language. E.g., an event consisting of a conjunction of two atomic subevents is an eca:Event, a snoop:Event, and a snoop:And.
- eca:Condition, eca:Query, eca:Test, eca:Action: similar to eca:Event with according sublanguages.
- eca:Event, eca:Condition, eca:Query, eca:Test, eca:Action can also be eca:Opaque; then the sublanguage is not an RDF language but an XML or string literal that contains a fragment of some actual language.
- eca:Opaque instances have the properties eca:language (0,1), eca:uri (0,1), eca:method (0,1 – post or get), and eca:opaqueSpec (1), which is the actual literal.
- all language constructs have properties that describe their variable use (cf. Section 11.2.3).

Obtaining the ECA-ML XML Markup from ECA-ML RDF. The XML representation of rules not containing opaque stuff is obtained from the striped RDF by omitting the has-event, has-query, has-test, and has-action elements. For the condition component, only the sequence of queries and tests is given. Additionally, for refinement by subclasses that add attributes, i.e., Opaque and Atomic etc., subelements have to be inserted. The DTDs that describe this mapping are given in Section 12.5.

Analogously, striped RDF can be obtained from the XML representation by extension with the above elements, and mapping the AtomicCondition fragments back as a list or conjunction.

12.2 Conditions, Queries, Tests: XML Markup vs. OWL Ontology

12.2.1 Problem Statement

In the XML markup developed above, a rule is an EQ*TA *sequence*. The ordering of evaluation of queries is actually mainly a computational (often even optimization) issue. From the modeling point of view, the

$$\text{Condition} \equiv \text{Queries} + \text{Tests}$$

component is not a sequence, but an expression, in general a conjunction of subexpressions (which are serialized in the XML markup).
From that point of view,

- the XML markup is seen as *executable code*, either written by the user, or generated from an *OWL-based Rule Engine* which cares for a semantically correct serialization (usage of variables) and optimization;
- the OWL ontology is the *model* of the rule that is used for rule analysis.

Thus, on the OWL level, a rule consists of

- an event component: eca:has-event, eca:Event,
- a condition component: eca:has-condition, eca:Condition, and
- an action component: eca:has-action, eca:Action.

12.2.2 Structure of eca:Condition in OWL

The eca:Condition component can be of three types (one simple and two composite):

- an eca:AtomicCondition; such a condition is *atomic from the ECA point of view*: it has a language which is different from the ECA language, and is submitted as a whole to another service.

  ```
  eca:AtomicCondition rdfs:subClassOf eca:Condition.
  eca:Query rdfs:subClassOf eca:AtomicCondition.
  eca:Test rdfs:subClassOf eca:AtomicCondition.
  ```

- a conjunctive expression eca:ConjunctiveCondition which consists of eca:AtomicConditions:

  ```
  eca:ConjunctiveCondition rdfs:subClassOf eca:Condition.
  eca:has-atomicCondition a rdf:Property;
                  rdfs:domain eca:ConjunctiveCondition;
                  rdfs:range eca:AtomicCondition.
  ```

  Amongst the individual eca:AtomicConditions (which are then expressions in some sublanguage, including opaque ones) of an eca:ConjunctiveCondition, there is no inherent ordering. Only if they are actually organized in a list, they get the ordering.

- or a list eca:ListCondition, which also consists of eca:AtomicConditions. A eca:ListCondition is an rdf:List, and its constructors are based on rdf:first and rdf:rest.

```
eca:ListCondition rdfs:subClassOf eca:Condition.
eca:ListCondition rdfs:subClassOf rdf:List.
eca:firstInConditionList a rdf:Property;
        rdfs:subPropertyOf rdf:first;
        rdfs:domain eca:ConditionList;
        rdfs:range eca:AtomicCondition.
eca:restConditionList a rdf:Property;
        rdfs:subPropertyOf rdf:rest;
        rdfs:domain eca:ConditionList;
        rdfs:range eca:ConditionList.
```

Atomic Queries vs. Tests. Queries use properties defined in any of the domain ontologies while tests use comparison predicates like $<$ or substring(_,_).

Usage: intended user format and internal format Note that a composite condition can be both an eca:ConjunctiveCondition and an eca:ListCondition (associated with a list structure). The intention is that

- the user gives a condition as eca:ConjunctiveCondition, and
- the rule engine or the condition engine can turn it (internally) into a eca:ListCondition which is then executed. This is expected to happen *inside the RDF graph*, on the same nodes. (Note that more sophisticated engines can also use parallel evaluation.)

Example 50 *Figure 12.1 shows a schematic ECA OWL rule graph. Note that the condition <#theRule/C> is both an eca:ConjunctiveCondition and an eca:ListCondition. <#theRule/C2> is eca:Opaque. The variable usage is not shown – it would be indicated (or derived) with each of the eca:AtomicConditions.*
Example 51 shows later how this graph is exported according to the striped ECA-ML DTD.

12.2.3 Assertions

The RDF level of rules provides the base for reasoning about rules. As it is also preferable to allow for opaque legacy components, these can be accompanied by *assertions* that formalize the relationship between the variables. For the XML representation, eca:Assertion elements can be added after each component. Apart from using assertions for reasoning, rule execution can use them as constraints to be verified – and raising an exception if the assertion is not satisfied. While a test simply discards the variable bindings that do not pass (if no bindings remain, the firing of the rule ends), for an assertion failure of a tuple must cause a serious exception.

12.3 Variables

Variables of the rule are resources of the whole rule (e.g. of the form the uri *rule-uri*/vars/*varname*), or more concretely foo:bla/rule123/vars#N. Both slash namespaces and hash namespaces are allowed. On the RDF level, variables are instances of eca:Variable.

12.3.1 Variable Handling

For controlling the exchange of variable bindings between the rule and its components, the ontology properties

- eca:uses-variable
- eca:has-free-variable

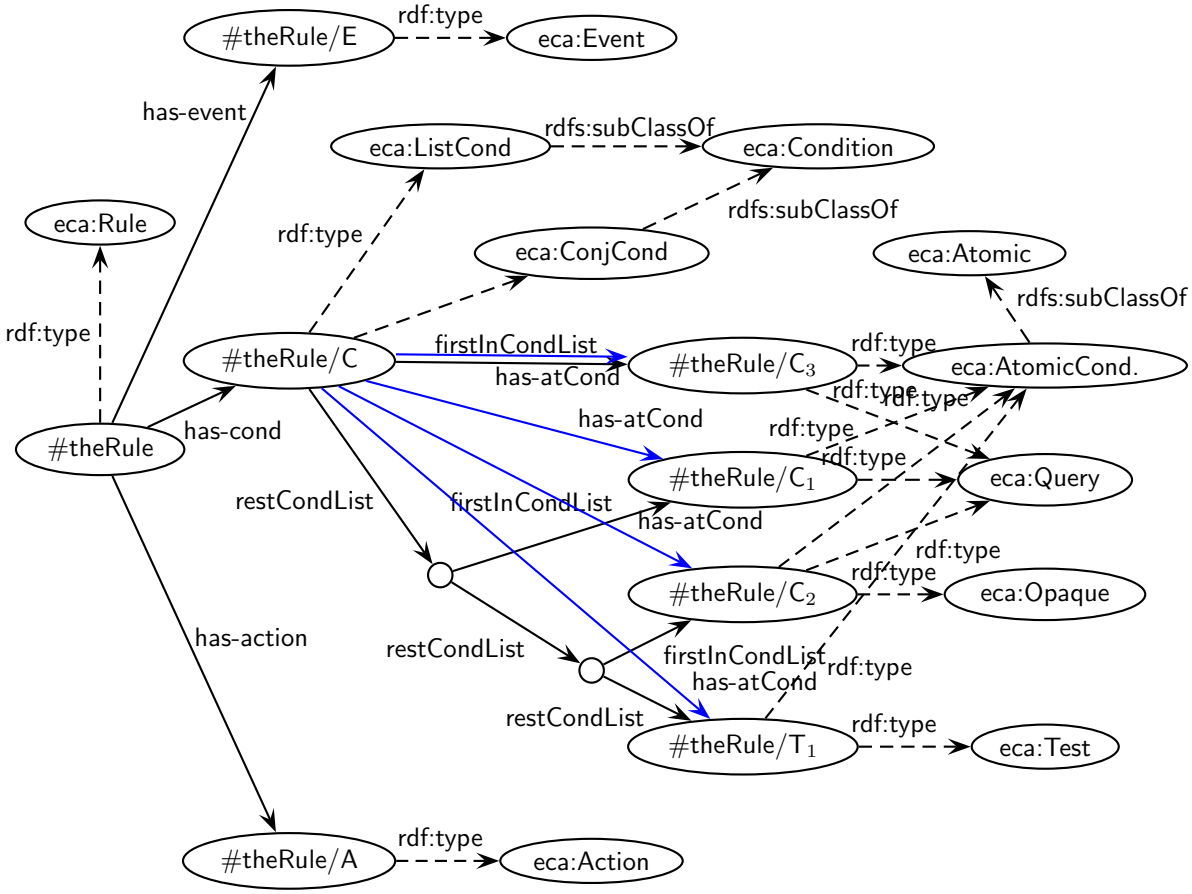


Figure 12.1: Structure of an ECA OWL Rule graph (using abbreviated names)

- eca:has-bound-variable
- eca:has-positive-variable
- eca:has-negative-variable
- eca:has-input-variable
- eca:has-output-variable

are used.

- uses-variable, has-free-variable, has-bound-variable, has-positive-variable, and has-negative-variable are defined as usual. This information is sufficient if the ECA-engine can derive by itself which variable-bindings must be communicated to evaluate a component,
- The specification of eca:has-input-variable must be given explicitly if the engine needs it. Input variables can be all variables that occur inside and outside the fragment, and have a positive occurrence before the respective component (note that this has some interferences when query reordering for optimization is considered).
- The specification of eca:has-result-variable can especially be used for opaque components.
- If a variable has a binding/positive occurrence before a component, and it is not declared to be an input variable, the join after return will result in the correct semantics.
  If all variables that the inner and outer part have in common and that have a binding/positive

occurrence before a component are declared to be input, then, the join afterwards can be omitted.

12.3.2 Fully MARS-Enabled Services

Fully MARS-Enabled Services are expected to provide a task analyze-variables that analyzes their variable usage:

- input: the language fragment; optionally the set of variables bound when this fragment is called,
- output: variables that occur positively/negatively.

The actual output format is RDF/XML, providing the variable usage triples of the fragment. It uses the "expressions" ontology (cf. Section 11.2.3):

```
<rdf:RDF xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
         xmlns:expr='http://www.semwebtech.org/mars/2006/expressions#'>
<rdf:Description rdf:about='foo://bla/theRule/Query'>
         <expr:has-positive-variable rdf:resource='foo:x#N'/>
         <expr:has-positive-variable rdf:resource='foo:x#M'/>
         :
</rdf:Description>
```

Note that instead of <rdf:Description>, also the respective class of the fragment is allowed, e.g. <owlq:Query>.

12.3.3 The Implicit Event Variable

Every rule has an implicit variable $Event (its URI is *uri-of-the-rule/vars/Event*) which can optionally also be declared explicitly in the rule as

```
<eca:Variable rdf:about="vars/Event" /> ,
```

and which is bound to the event instance that fired the rule. Note that the event itself is *volatile* data, thus, at the evaluation time it has to be added as a local graph, including all volatile RDF nodes, to the rule.

12.3.4 Mixed XML and RDF Level

When XML (value/string-oriented) and RDF (URI-oriented) representation are mixed (e.g., when mapping the RDF to the XML representation of a rule), the variable names are projected as follows:

```
<eca:Rule>
  <eca:uses-variable rdf:resource="path-uri/N" />
  <eca:has-query>
    <eca:Query>
      <eca:has-input-variable rdf:resource="path-uri/N" />
        :
    </eca:Query>
  </eca:has-query>
    :
</eca:Rule>
```

is mapped to

```
<eca:Rule>
  <eca:uses-variable name="N" />
  <eca:Query>
    <eca:has-input-variable eca:name="N" />
      :
  </eca:Query>
    :
</eca:Rule>
```

By convention, the *local part* of the URI is used as string *variable name*. This also occurs when an OWLQ component (whose variables are URI references) is used in a rule in XML markup.

12.4 The ECA Ontology and OWL File

For all ECA notions, rdfs:isDefinedBy points to http://www.semwebtech.org/languages/2006/eca-ml. Note that we define eca:Event as equivalent to eca:EventSpec (which is actually the more correct naming, as the thing is not an event, but an event specification); analogous for eca:ActionSpec.

```
<!DOCTYPE rdf:RDF [
  <!ENTITY mars "http://www.semwebtech.org/mars/2006/mars#">
  <!ENTITY expr "http://www.semwebtech.org/mars/2006/expressions#">
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#"> ]>
<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#"
   xml:base="http://www.semwebtech.org/languages/2006/eca-ml">

<owl:Ontology rdf:about="#"/>

<owl:Class rdf:about="#Rule">
 <rdfs:isDefinedBy rdf:resource="#"/>
</owl:Class>

<owl:Class rdf:about="#Event">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <owl:disjointWith rdf:resource="#Rule"/>
 <owl:equivalentClass rdf:resource="#EventSpec"/>
</owl:Class>

<!-- note that queries, tests, and assertions are
  not required to be disjoint -->

<owl:Class rdf:about="#Query">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <owl:disjointWith rdf:resource="#Rule"/>
 <owl:disjointWith rdf:resource="#Event"/>
</owl:Class>

<owl:Class rdf:about="#Condition">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <owl:disjointWith rdf:resource="#Rule"/>
 <owl:disjointWith rdf:resource="#Event"/>
</owl:Class>
```

```
<owl:Class rdf:about="#Test">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <owl:disjointWith rdf:resource="#Rule"/>
 <owl:disjointWith rdf:resource="#Event"/>
</owl:Class>

<owl:Class rdf:about="#Assertion">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <owl:disjointWith rdf:resource="#Rule"/>
 <owl:disjointWith rdf:resource="#Event"/>
</owl:Class>

<owl:Class rdf:about="#Action">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <owl:disjointWith rdf:resource="#Rule"/>
 <owl:disjointWith rdf:resource="#Event"/>
 <owl:disjointWith rdf:resource="#Condition"/>
 <owl:disjointWith rdf:resource="#Query"/>
 <owl:disjointWith rdf:resource="#Test"/>
 <owl:disjointWith rdf:resource="#Assertion"/>
 <owl:equivalentClass rdf:resource="#ActionSpec"/>
</owl:Class>

<rdf:Property rdf:about="#has-event">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:domain rdf:resource="#Rule"/>
 <rdfs:range rdf:resource="#Event"/>
</rdf:Property>

<rdf:Property rdf:about="#has-condition">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:domain rdf:resource="#Rule"/>
 <rdfs:range rdf:resource="#Condition"/>
</rdf:Property>

<rdf:Property rdf:about="#has-query">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:domain rdf:resource="#Rule"/>
 <rdfs:range rdf:resource="#Query"/>
</rdf:Property>

<rdf:Property rdf:about="#has-test">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:domain rdf:resource="#Rule"/>
 <rdfs:range rdf:resource="#Test"/>
</rdf:Property>

<rdf:Property rdf:about="#has-assertion">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:domain rdf:resource="#Rule"/>
 <rdfs:range rdf:resource="#Assertion"/>
</rdf:Property>
```

```
<rdf:Property rdf:about="#has-action">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:domain rdf:resource="#Rule"/>
 <rdfs:range rdf:resource="#Action"/>
</rdf:Property>

<!-- Composite Conditions -->

<owl:Class rdf:about="#SimpleCondition">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#Condition"/>
</owl:Class>

<owl:Class rdf:about="#ConjunctiveCondition">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#Condition"/>
</owl:Class>

<owl:Class rdf:about="#ListCondition">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#Condition"/>
 <rdfs:subClassOf rdf:resource="&rdf;List"/>
</owl:Class>

<rdf:Property rdf:about="#has-atomicCondition">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:domain rdf:resource="#Condition"/>
 <rdfs:range rdf:resource="#AtomicCondition"/>
</rdf:Property>

<rdf:Property rdf:about="#firstInConditionList">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subPropertyOf rdf:resource="&rdf;first"/>
 <rdfs:domain rdf:resource="#ConditionList"/>
 <rdfs:range rdf:resource="#AtomicCondition"/>
</rdf:Property>

<rdf:Property rdf:about="#restConditionList">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subPropertyOf rdf:resource="&rdf;rest"/>
 <rdfs:domain rdf:resource="#ConditionList"/>
 <rdfs:range rdf:resource="#ConditionList"/>
</rdf:Property>

<!-- Opaque stuff -->

<owl:Class rdf:about="#Opaque">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#opaqueSpec"/>
   <owl:cardinality>1</owl:cardinality>
  </owl:Restriction>
 </rdfs:subClassOf>
```

```
</owl:Class>

<owl:Class rdf:about="#OpaqueRule">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#Opaque"/>
 <rdfs:subClassOf rdf:resource="#Rule"/>
</owl:Class>

<owl:Class rdf:about="#OpaqueEvent">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#Opaque"/>
 <rdfs:subClassOf rdf:resource="#Event"/>
 <owl:equivalentClass rdf:resource="#OpaqueEventSpec"/>
</owl:Class>

<owl:Class rdf:about="#OpaqueQuery">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#Opaque"/>
 <rdfs:subClassOf rdf:resource="#Query"/>
</owl:Class>

<owl:Class rdf:about="#OpaqueTest">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#Opaque"/>
 <rdfs:subClassOf rdf:resource="#Test"/>
</owl:Class>

<owl:Class rdf:about="#OpaqueAction">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#Opaque"/>
 <rdfs:subClassOf rdf:resource="#Action"/>
 <owl:equivalentClass rdf:resource="#OpaqueActionSpec"/>
</owl:Class>

<owl:DatatypeProperty rdf:about="#opaqueSpec">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:domain rdf:resource="#Opaque"/>
 <!-- usually the value is parseType:XMLLiteral or a string -->
</owl:DatatypeProperty>

<rdf:Property rdf:about="#language">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:domain rdf:resource="#Opaque"/>
 <rdfs:range rdf:resource="&mars;Language"/>
</rdf:Property>
<rdf:Property rdf:about="#uri">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:domain rdf:resource="#Opaque"/>
 <rdfs:range rdf:resource="&mars;Service"/>
</rdf:Property>
<rdf:Property rdf:about="#method">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:domain rdf:resource="#Opaque"/>
 <!-- range: HTTP's GET/POST - use strings -->
```

```
</rdf:Property>

<!-- variable stuff as in vars.rdf (for expressions) -->

<owl:Class rdf:about="#Variable">
 <rdfs:subClassOf rdf:resource="&expr;Variable"/>
</owl:Class>

<rdf:Property rdf:about="#uses-variable">
 <owl:equivalentProperty rdf:resource="&expr;uses-variable"/>
</rdf:Property>
<rdf:Property rdf:about="#bind-to-variable">
 <owl:equivalentProperty rdf:resource="&expr;bind-to-variable"/>
</rdf:Property>
<rdf:Property rdf:about="#has-free-variable">
 <owl:equivalentProperty rdf:resource="&expr;has-free-variable"/>
</rdf:Property>
<rdf:Property rdf:about="#has-bound-variable">
 <owl:equivalentProperty rdf:resource="&expr;has-bound-variable"/>
</rdf:Property>
<rdf:Property rdf:about="#has-positive-variable">
 <owl:equivalentProperty rdf:resource="&expr;has-positive-variable"/>
</rdf:Property>
<rdf:Property rdf:about="#has-negative-variable">
 <owl:equivalentProperty rdf:resource="&expr;has-negative-variable"/>
</rdf:Property>
<rdf:Property rdf:about="#scopes-variable">
 <owl:equivalentProperty rdf:resource="&expr;scopes-variable"/>
</rdf:Property>
<rdf:Property rdf:about="#has-input-variable">
 <owl:equivalentProperty rdf:resource="&expr;has-input-variable"/>
</rdf:Property>
<rdf:Property rdf:about="#has-output-variable">
 <owl:equivalentProperty rdf:resource="&expr;has-output-variable"/>
</rdf:Property>
</rdf:RDF>
```

```
# call
# jena -q -pellet -il RDF/XML -qf ecaml-query.sparql

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?X ?Y
FROM <file:eca-ml.rdf>
WHERE { ?X rdfs:subClassOf ?Y }
```

12.5 DTDs of ECA-ML

The striped and stripped ECA-ML DTDs differ mainly in the upper level representation:

- striped: Rule(has-event(Event), ((has-condition(Condition))* | ((has-condition(Condition))*, (has-test(Test))*)), has-action(Action))
- stripped: Rule(Event, (Condition* | (Query*, Test*)), Action)

The contents of Event, Condition, and Action components is shared by both; the representation of Opaque is slightly different.

Remarks on Stripped/Striped DTD Design. Note the following principles in the design of the Striped DTD:

- Language Change: Event/Condition/Action have an owl:sameAs subelement, which in turn has Opaque, Atomic, or ANY-other-language contents. Here, ANY-other-language indicates that a language change to another language (and DTD) takes place.
- Striped: there are no attributes except rdf:about and rdf:resource.
- Stripped: the Opaque, Atomic, and ANY-other-language are immediate children of Event/Condition/Action (the owl:sameAs disappears), the subelements of Opaque become attributes.

```
<!ENTITY % initvar "initialize-variable*">
<!ENTITY % usvar "uses-variable*">
<!ENTITY % var-decl "uses-variable | bind-to-variable |
    has-input-variable | has-output-variable |
    has-free-variable | has-bound-variable | has-positive-variable |
    has-negative-variable | scopes-variable | initialize-variable">
<!ENTITY % var-declS "bind-to-variable*, uses-variable*,
    has-input-variable*, has-output-variable*,
    has-free-variable*, has-bound-variable*, has-positive-variable*,
    has-negative-variable*">

<!ELEMENT Rule (%initvar;, %usvar;, has-event,
          has-query*, has-condition?, has-test?, has-action+) >
<!ELEMENT has-event (Event)>
<!ELEMENT has-condition (Condition)>
<!ELEMENT has-query (Query)>
<!ELEMENT has-test (Test)>
<!ELEMENT has-action (Action)>
<!ELEMENT Event (%var-declS;, owl:sameAs)>
<!ELEMENT Condition (%var-declS;,
     (ListCondition | owl:sameAs | ANY-other-language?))>
 <!--  Note: export only as ListCondition, not as ConjunctiveCondition! -->
<!ELEMENT Query (%var-declS;, owl:sameAs)>
<!ELEMENT Test  (%var-declS;, owl:sameAs)>
<!ELEMENT Action (%var-declS;, owl:sameAs)>

<!ATTLIST Rule   rdf:about CDATA #IMPLIED>
<!ATTLIST Event  bind-to-variable CDATA #IMPLIED
                 rdf:about CDATA #IMPLIED>
<!ATTLIST Query  bind-to-variable CDATA #IMPLIED
                 rdf:about CDATA #IMPLIED>
<!ATTLIST Test   bind-to-variable CDATA #IMPLIED
                 rdf:about CDATA #IMPLIED>
<!ATTLIST Action rdf:about CDATA #IMPLIED>
```

```
<!ELEMENT ListCondition (firstInConditionList, restConditionList?)>
<!ELEMENT firstInConditionList (Query | Test)>
<!ELEMENT restConditionList (ListCondition | Query | Test)>

<!-- variable usage -->
<!ELEMENT bind-to-variable EMPTY>
   <!ATTLIST bind-to-variable rdf:resource CDATA #REQUIRED >
<!ELEMENT uses-variable EMPTY>
   <!ATTLIST uses-variable rdf:resource CDATA #REQUIRED >
<!ELEMENT has-input-variable EMPTY>
   <!ATTLIST has-input-variable rdf:resource CDATA #REQUIRED >
<!-- less important (implicit) variable usage -->
<!ELEMENT has-output-variable EMPTY>
   <!ATTLIST has-output-variable rdf:resource CDATA #REQUIRED >
<!ELEMENT has-free-variable EMPTY>
   <!ATTLIST has-free-variable rdf:resource CDATA #REQUIRED >
<!ELEMENT has-bound-variable EMPTY>
   <!ATTLIST has-bound-variable rdf:resource CDATA #REQUIRED >
<!ELEMENT has-positive-variable EMPTY>
   <!ATTLIST has-positive-variable rdf:resource CDATA #REQUIRED >
<!ELEMENT has-negative-variable EMPTY>
   <!ATTLIST has-negative-variable rdf:resource CDATA #REQUIRED >
<!ELEMENT scopes-variable EMPTY>
   <!ATTLIST scopes-variable rdf:resource CDATA #REQUIRED >
<!-- initialize-var: contents may contain a string (prg code) -->
<!ELEMENT initialize-variable (#PCDATA)>
   <!ATTLIST initalize-variable name CDATA #REQUIRED
                                language CDATA #IMPLIED
                                select CDATA #IMPLIED >
<!-- for embedded, opaque and atomic stuff -->
<!ELEMENT owl:sameAs ( Opaque | Atomic | ANY-other-language?)>
   <!ATTLIST owl:sameAs rdf:resource CDATA #IMPLIED >
<!ELEMENT ANY-other-language ANY>
<!ELEMENT Opaque (%var-declS;, language*, uri*, method*,
                  content-type*, opaqueSpec*)>
<!ATTLIST Opaque rdf:about CDATA #IMPLIED>
<!-- Atomic: contents of some other language -->
<!ELEMENT Atomic (%var-declS;, language*, ANY-other-language?) >
<!ATTLIST Atomic rdf:about CDATA #IMPLIED>
<!ELEMENT language (#PCDATA)>
<!ELEMENT uri EMPTY>
   <!ATTLIST uri rdf:resource CDATA #REQUIRED >
<!ELEMENT method (#PCDATA)>
<!ELEMENT opaqueSpec (#PCDATA)>
<!ELEMENT content-type (#PCDATA)>
```

```
<!ENTITY % initvar "initialize-variable*">
<!ENTITY % usvar "uses-variable*">
<!ENTITY % var-declP "(uses-variable | bind-to-variable |
    has-input-variable | has-output-variable |
    has-free-variable | has-bound-variable | has-positive-variable |
    has-negative-variable | scopes-variable | initialize-variable)*">
<!ENTITY % var-decl "uses-variable | bind-to-variable |
```

```
    has-input-variable | has-output-variable |
    has-free-variable | has-bound-variable | has-positive-variable |
    has-negative-variable | scopes-variable | initialize-variable">

<!ELEMENT Rule (%initvar;, %usvar;, Event, Query*, Test?, Action+) >
<!ELEMENT Event (%var-declP;, (Opaque | Atomic | ANY-other-language?))>
<!ELEMENT Query (%var-declP;, (Opaque | Atomic | ANY-other-language?))>
<!ELEMENT Test  (%var-declP;,  (Opaque | Atomic | ANY-other-language?))>
<!ELEMENT Action (%var-declP;, (Opaque | Atomic | ANY-other-language?))>
<!ELEMENT ANY-other-language ANY>

<!ATTLIST Event  bind-to-variable CDATA #IMPLIED>
<!ATTLIST Query  bind-to-variable CDATA #IMPLIED>

<!-- variable usage -->
<!ELEMENT bind-to-variable EMPTY>
   <!ATTLIST bind-to-variable name CDATA #REQUIRED >
<!ELEMENT uses-variable EMPTY>
   <!ATTLIST uses-variable name CDATA #REQUIRED >
<!ELEMENT has-input-variable EMPTY>
   <!ATTLIST has-input-variable name CDATA #REQUIRED
                                use CDATA #IMPLIED>
<!-- less important (implicit) variable usage -->
<!ELEMENT has-output-variable EMPTY>
   <!ATTLIST has-output-variable name CDATA #REQUIRED >
<!ELEMENT has-free-variable EMPTY>
   <!ATTLIST has-free-variable name CDATA #REQUIRED >
<!ELEMENT has-bound-variable EMPTY>
   <!ATTLIST has-bound-variable name CDATA #REQUIRED >
<!ELEMENT has-positive-variable EMPTY>
   <!ATTLIST has-positive-variable name CDATA #REQUIRED >
<!ELEMENT has-negative-variable EMPTY>
   <!ATTLIST has-negative-variable name CDATA #REQUIRED >
<!ELEMENT scopes-variable EMPTY>
   <!ATTLIST scopes-variable name CDATA #REQUIRED >
<!-- initialize-var: contents may contain a string (prg code) -->
<!ELEMENT initialize-variable (#PCDATA)>
   <!ATTLIST initalize-variable name CDATA #REQUIRED
                                language CDATA #IMPLIED
                                select CDATA #IMPLIED >
<!-- for opaque stuff -->
<!ELEMENT Opaque (#PCDATA | %var-decl; | ANY-other-language)* >
<!ATTLIST Opaque language CDATA #IMPLIED
                 uri CDATA #IMPLIED
                 method CDATA #IMPLIED
                 content-type CDATA #IMPLIED>
<!-- Atomic: contents of some other language -->
<!ELEMENT Atomic (%var-declP;, ANY-other-language?) >
<!ATTLIST Atomic language CDATA #IMPLIED
                 rdf:about CDATA #IMPLIED >
    <!-- if not obvious from namespace (e.g. xmlqlmatch) -->
```

Note how the projection of conditions to the ECA-ML XML markup takes place: conditions can be Atomic or Opaque (which can be elements of other namespaces), or composite. For

composite conditions, only ListCondition is actually exported. ListCondition in turn exports the firstInConditionList and restConditionList list structure whose leaves are AtomicConditions.

When projecting the tree according to its stripped DTD, the has-condition and ListCondition elements and the list structure is omitted and only the Query and Test elements remain.

Example 51 (ECA-ML Rule in Striped and Stripped XML) *Consider again the ECA OWL Rule Graph given in Example 50. It is exported according to the Striped ECA-ML DTD as shown below. The striped markup includes the list structure:*

```
<eca:Rule>
 <eca:has-event>
  <eca:Event> ... </eca:Event>
 </eca:has-event>
 <eca:has-condition>
  <eca:ListCondition>
   <eca:firstInConditionList>
    <eca:Query>
     <owl:sameAs>
      <owlq:... xmlns:owlq="..."> ... </owlq:...>
     </owl:sameAs>
    </eca:Query>
   </eca:firstInConditionList>
   <eca:restConditionList>
    <eca:ListCondition>
     <eca:firstInConditionList>
      <eca:Query>
       <owl:sameAs>
        <eca:Opaque language="..." or uri="..." and method="...">
         <opaqueSpec>
          <!-- opaque specification literal -->
         </opaqueSpec>
        </eca:Opaque>
       </owl:sameAs>
      </eca:Query>
     </eca:firstInConditionList>
     <eca:restConditionList>
      <eca:ListCondition>
       <eca:firstInConditionList>
        <eca:Query> ... </eca:Query>
       </eca:firstInConditionList>
       <eca:restConditionList>
        <eca:ListCondition>
         <eca:firstInConditionList>
          <eca:Test> ... </eca:Test>
         </eca:firstInConditionList>
         <!-- no rest here -->
        </eca:ListCondition>
       </eca:restConditionList>
      </eca:ListCondition>
     </eca:restConditionList>
    </eca:ListCondition>
   </eca:restConditionList>
  </eca:ListCondition>
 </eca:has-condition>
 <eca:has-action>
```

```
    <eca:Action> ... </eca:Action>
   </eca:has-action>
  </eca:Rule>
```

The stripped markup strips the list structure and removes the intermediate owl:sameAs*:*

```
<eca:Rule>
  <eca:Event> ... </eca:Event>
  <eca:Query>
    <owlq:... xmlns:owlq="..."> ... </owlq:...>
  </eca:Query>
  <eca:Query>
    <eca:Opaque language="..." or uri="..." and method="...">
      <!-- opaque specification literal -->
    </eca:Opaque>
  </eca:Query>
  <eca:Query> ... </eca:Query>
  <eca:Test> ... </eca:Test>
  <eca:Action> ... </eca:Action>
</eca:Rule>
```

The embedded rule components are then given by the respective ontologies. In the following sections, we give a set of sample component ontologies:

- specification of atomic events,
- queries and tests (the latter are queries with true/false as result),
- specification of atomic actions,
- composite events,
- composite actions.

In Chapter 13 we will first describe the language OWLQ that is used for conjunctive queries, tests, and atomic event specifications (which are actually queries against an event as an RDF graph). Then, in Chapter 14 we describe sample ontologies for composite events and actions which are instances of ontologies of algebras (cf. Section 11.2.2).

Chapter 13

OWLQ: Queries, Tests and Atomic Event Specifications

Queries have yet mainly been discussed in terms of opaque queries (XML – XPath/XQuery, Xcerpt). In Section 5.6 we discussed that querying can be divided into to parts: *obtaining* answers from data, and *combining* answers, usually in an algebraic way. On the other hand, the combination of atomic concepts is often done in a logical way, e.g. in First-Order Logic, with logical connectives. As shown for the relational algebra it is equivalent to the (safe) relational calculus.

Queries, tests and atomic event specifications have been dealt with on an opaque or merely syntactical level until now:

- *Atomic Event Specifications* have been discussed in Section 5.3.2; actually, they are kinds of queries against incoming events (which are small RDF graphs).
- Queries have shortly been discussed in Section 5.6 – as opaque queries on the datamodel level (e.g., using SPARQL, XPath or XQuery).
- Atomic actions have been given as XML terms with variable references as e.g. used in XQuery return statements.

The above formalisms do not allow to reason about a rule, e.g., "which rules are affected from cancelling flight LH123"? – probably e.g., LH123, from Frankfurt to JFK, will trigger a rule whose firing event is "on any event that concerns a flight to New York". For that, domain-level information together with insight into the atomic event specifications of the rules are necessary.

We first discuss in Section 13.1.1 an approach that uses RDF graph *patterns*, reminiscent of the XML-QL style matching formalism from Section 5.3.2.1 that turned out to be quite "declarative" – but it is not semantic at all. Although this looks nice, it must be discarded since it is not compatible with the inherent semantics of RDF/XML information. This is then revised into a cleaner approach which separates the patterns from *real* instance data (Section 13.1.2). Nevertheless, with this, the patterns are not accessible for reasoning about rules. Starting with Section 13.2, we present OWLQ, a a formalism for formulas that allows the use as queries, atomic event specifications, tests, and assertions that is based on *class specifications* and *constraints*, which then allows for DL-based reasoning about rules.

The following sample data contains

- two sample flights, LH123 from Frankfurt to JFK, and LH456 from MUC to LIS,
- a cancellation event for the first of them.

```
<!DOCTYPE rdf:RDF [
  <!ENTITY travel "http://www.semwebtech.org/domains/2006/travel#"> ]>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
   xml:base="sample:traveldata#">
<travel:Flight rdf:about="&travel;/airlines/lufthansa/flights/lh123">
 <travel:from rdf:resource="&travel;/airports/fra"/>
 <travel:to rdf:resource="&travel;/airports/jfk"/>
</travel:Flight>
<travel:Flight rdf:about="&travel;/airlines/lufthansa/flights/lh456">
 <travel:from rdf:resource="&travel;/airports/muc"/>
 <travel:to rdf:resource="&travel;/airports/lis"/>
</travel:Flight>
<travel:CanceledFlight rdf:ID="occur-cf-event">
 <travel:flight rdf:resource="&travel;/airlines/lufthansa/flights/lh123"/>
</travel:CanceledFlight>
</rdf:RDF>
```

13.1 Discarded Pattern Approaches

13.1.1 The Embedded Pattern Approach

For some time, we favored an approach where atomic events and queries are represented by RDF graph patterns like in SPARQL, but explicit in an RDF/XML markup. This would make up for a nice rule representation: Some nodes belong to the ECA-ML URI domain/namespace, the next level belongs to the URI domain/namespace of the algebraic languages like Snoopy or CCS, and the "atomic" leaves of the algebra trees belong to the domain namespaces. Even shared logical variables easily fit into that graph approach.

But, if these rules get "freely accessible" as RDF/XML data in the Web, it is *not* known that the application domain elements stand for patterns, and not for instances. The patterns mix up with the range and domain axioms of the domain ontology would result in extremely unintended, if not even inconsistent "information".

Example 52 (Discarded: Rules with Atomics as Graph Patterns) *We show in two examples how variables are modeled by resources: Consider the ECA rule "if a flight is cancelled that is relevant to one of our customers, then send that customer an SMS" which uses an atomic event, a simple conjunctive query/condition and an atomic action component.*

The most declarative convenient way is to "declare" variables as resources on the "global" level of a rule and to reference them in the statements; nevertheless, the declaration can also happen at arbitrary object positions:

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:eca="http://www.semwebtech.org/languages/2006/eca-ml#"
   xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
   xmlns:comm="http://www.semwebtech.org/domains/2006/communication#"
   xmlns="foo:local#">

<Variable rdf:ID="Flight"/>
<Variable rdf:ID="Date"/>
<Variable rdf:ID="Mobile"/>
<eca:Rule>
```

```
 <eca:has-event>
  <travel:CanceledFlight rdf:ID="cf">
   <travel:flight rdf:resource="#Flight"/>
  </travel:CanceledFlight>
 </eca:has-event>
 <eca:has-query>
   <travel:Customer>
    <travel:booked-for rdf:resource="#Flight"/>
    <travel:mobile-no rdf:resource="#Mobile"/>
   </travel:Customer>
 </eca:has-query>
 <eca:has-action>
  <comm:send-sms>
   <comm:to rdf:resource="#Mobile"/>
   <text>Your flight {#Flight} has been cancelled</text>
  </comm:send-sms>
 </eca:has-action>
</eca:Rule>
</rdf:RDF>
```

In the above rule, the node

```
  <travel:CanceledFlight rdf:ID="cf">
   <travel:flight rdf:resource="#Flight">
  </travel:CanceledFlight>
```

is not *intended as an instance of a* travel:CanceledFlight *event, but it inevitable would be parsed as such. The same holds for the RDF pattern query. Both would even result in the fact that the resource identified by* local#Flight *is not only a variable, as intended, but also a* travel:flight *instance (as implied by the range definitions of the* travel *namespace).*

Below, a SPARQL query is given that illustrates the semantics and the problem of the above definitions. It uses an excerpt of the travel domain definitions:

```
<!DOCTYPE rdf:RDF [
  <!ENTITY mars "http://www.semwebtech.org/mars/2006/mars#"> ]>
<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:travel="http://www.semwebtech.org/domains/2006/travel#"
   xml:base="http://www.semwebtech.org/domains/2006/travel">

<owl:Class rdf:about="#Event">
 <rdfs:subClassOf rdf:resource="&mars;Event"/>
</owl:Class>
<owl:Class rdf:about="#FlightEvent">
 <rdfs:subClassOf rdf:resource="#Event"/>
</owl:Class>
<owl:Class rdf:about="#CanceledFlight">
 <rdfs:subClassOf rdf:resource="#FlightEvent"/>
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#flight"/>
   <owl:allValuesFrom rdf:resource="#Flight"/>
```

```
  </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>
</rdf:RDF>
```

```
# call
# jena -q -pellet -il RDF/XML -qf rule-query-1.sparql

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mars: <http://www.semwebtech.org/mars/2006/mars#>
PREFIX travel: <http://www.semwebtech.org/domains/2006/travel#>

SELECT ?X ?Z ?T
FROM <file:eca-ml.rdf>
FROM <file:rule-example-1.rdf>
FROM <file:travel-miniexcerpt.rdf>
WHERE {{?X rdf:type mars:Event .
        ?X rdf:type ?T } UNION
       {?X travel:flight ?Z .
        ?Z rdf:type ?T }}
```

The result of the query shows that the rule pattern contributes to instances: `<file:rule-example-1.rdf#cf>` *is an instance of* `<http://www.semwebtech.org/mars/2006/mars#Event>`. *The same happens for patterns in the query part.*

13.1.2 Embedded Event and Query Patterns as parseType=XML

To avoid that AES patterns are "found" as RDF instance data in the Web, they can be handled as XML literals. When rule evaluation comes to the atomic level, they are "unpacked", parsed as RDF/XML, and can then be used. The disadvantage is that they are then completely opaque, and cannot be found when e.g. searching for "all rules that react on a travel:CanceledFlight event.

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:eca="http://www.semwebtech.org/languages/2006/eca-ml#"
   xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
   xmlns:comm="http://www.semwebtech.org/domains/2006/communication#"
   xmlns="foo:local#">

<Variable rdf:ID="Flight"/>
<Variable rdf:ID="Date"/>
<Variable rdf:ID="Mobile"/>
<eca:Rule>
 <eca:has-event rdf:parseType="Literal">
  <travel:CanceledFlight rdf:ID="cf">
   <travel:flight rdf:resource="#Flight"/>
  </travel:CanceledFlight>
 </eca:has-event>
 <eca:has-query rdf:parseType="Literal">
   <travel:Customer>
    <travel:booked-for rdf:resource="#Flight"/>
```

```
    <travel:mobile-no rdf:resource="#Mobile"/>
   </travel:Customer>
 </eca:has-query>
 <eca:has-action rdf:parseType="Literal">
  <comm:send-sms>
   <comm:to rdf:resource="#Mobile"/>
   <text>Your flight {#Flight} has been cancelled</text>
  </comm:send-sms>
 </eca:has-action>
</eca:Rule>
</rdf:RDF>
```

```
# call
# jena -q -pellet -il RDF/XML -qf rule-query-2.sparql

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mars: <http://www.semwebtech.org/mars/2006/mars#>
PREFIX eca: <http://www.semwebtech.org/languages/2006/eca-ml#>
PREFIX travel: <http://www.semwebtech.org/domains/2006/travel#>

SELECT ?X ?E ?ME ?TE ?EE
FROM <file:eca-ml.rdf>
FROM <file:rule-example-2.rdf>
FROM <file:travel-miniexcerpt.rdf>
WHERE {{?X eca:has-event ?E} UNION
       {?ME rdf:type mars:Event} UNION
       {?TE rdf:type travel:Event} UNION
       {?EE rdf:type eca:EventSpec}}
```

Thus, languages that use a pattern syntax for queries, and also for updates or rule reads, as known from Prolog, Datalog, F-Logic [42, 43], XPathLog [48, 51], or Xcerpt/XChange [24, 22] are not best-suited for being used in the Semantic Web.

13.2 Design of an Ontology for expressing various parts

The previous section showed that describing anything like queries, event specifications, or actions by "giving" the patterns explicitly is problematic wrt. the basic idea of the Semantic Web as an "open" data source since these patterns are –without understanding their context– understood as actual data. Thus, formulas must be given by using an ontology that *describes* the structure (i.e., the patterns) of the query or formula and its components.

Formulas can serve for all tasks that are still open: The actual use of the formula is then given by the context, i.e., by the property how it is referenced to from its environment:

- Queries are formulas (often constrained to conjunctive ones) with free variables (to be bound to answers during the evaluation). They are referenced by eca:has-query, or as subexpressions of a composite query according to some algebra (boolean or relational Algebra). From the definitions of the algebra ontology, it can be derived that the query is of type eca:Query). All nodes in the query are of types that are defined as static notions (classes or properties) by some domain ontology.
- Atomic Event Specifications are (small) queries against an RDF graph given by the event. They are referenced either as atomic events from an ECA rules (eca:has-event) or as subevents

from composite event languages like SNOOP (where the property also induces that they are of type eca:EventSpec), *and* – which is defined below in detail – they refer to a class (of events) that is defined as an event class (subclass of mars:Event) by some domain ontology.

- Tests and Assertions are (sometimes closed) formulas; free variables are required to be bound before evaluation. Tests can sometimes be included within conjunctive queries.
- Atomic Action specifications are quantifier-free formulas; free variables are required to be bound before evaluation. They describe the RDF graph that represents the action to be executed. The "handle" node of the graph is an instance of a class that is defined as an action (subclass of mars:Action) by some domain ontology.

Note that with the above definition, the same formula (over static concepts of application domains) can act as a query (when some variables are not yet bound), as a test (all variables bound), and as an assertion (all variables bound).

13.3 OWLQ

13.3.1 The OWLQ Conjunctive Query Language

The OWLQ language is designed as an RDF ontology of a logic-based query language that allows to *describe* the queries including the use of join conditions by variables. The language is not only an "ontologization" of SPARQL by providing constructs to describe triples/statements, using resources and variables (i.e., URIs that are of rdf:type Variable), but extends also the SPARQL *semantics* by allowing to define local classes by OWL means that are then used in the conjunctions. Having a logical semantics, OWLQ fits directly with the semantics of ECA rules based on logical variables.
OWLQ queries are based on the following concepts:

- definition of "relevant" classes (by common OWL means) to find out which instances "qualify" for a complex concept description (e.g., for an Atomic Event Specification). The definition can either define a new class as e.g. an intersection of an existing class and an owl:Restriction, or use owl:equivalentClass to refer to an existing one directly.
- declaration of variables ranging over instances of a class (SPARQL equivalent: {?CVar rdf:type :theClass}).
- declaration or constraining of variables ranging over the values (URIs or literals) of a property wrt. a value, a class membership, or another variable. (SPARQL equivalent e.g. {?Var$_1$:property ?Var$_2$});
  for constraints associated to a class, they refer to the instances of that class (or the variable that ranges over them): {?Var$_1$:property ?Var$_2$}).

The core of the OWL ontology is as follows:

- an owlq:Query has properties owlq:definesClass, owlq:hasConstraint, owlq:useVariable, and owlq:resultVariable (the distinguished ones);
- owlq:Class serves for defining relevant classes. The definition can either refer to an existing owl:Class or define a new class by OWL means, as e.g. an intersection of an existing class and an owl:Restriction;
- owlq:scopesVariable declares variables ranging over instances of such a class (SPARQL equivalent: {?CVar rdf:type *class*});
- an owlq:has{Mandatory|Optional}VariableDefinition is associated to an owlq:Variable v and defines another variable ranging over the value(s) of an rdf:Property wrt. v (SPARQL: {?Var$_1$ *dataproperty* ?Var$_2$}).

- an owlq:Constraint specifies additional conditions that relate variables, e.g. {?Var$_1$ *property* ?Var$_2$} or comparisons {?Var$_1$ *comparator* ?Var$_2$}. An owlq:Constraint can also be an owlq:NegatedConstraint.

- owlq:ClosedClasses and owlq:ClosedPredicates are closed wrt. a special form of CWA.

Subclasses of OWLQ:Query. For the embedding in MARS, owlq:EventSpec and owlq:Test are specialized subclasses of owlq:Query; see Section 13.4.

Variables Usage. OWLQ "code" fragments use variables in the following (not necessary disjoint) ways:

- owlq:inputVariable: bound from outside (if the query is e.g. embedded in a rule).

- owlq:resultVariable: variables that occur in the result projection. These can be mandatory or optional:

  - owlq:mustBindVariable,
  - owlq:optionalVariable.

  Note that enbedding a query in a rule usually requires to have the key input variables also as output variables.

- Variables that are neither used as owlq:inputVariable nor as owlq:resultVariables are owlq:DontCareVariables.

- owlq:positiveVariable: All non-Don't-Care variables that are not input must have a positive occurrence in the query to be safe. Positive occurrences are those where the variable occurs in the object position of a owlq:scopesVariable or owlq:VariableDefinition statement. Every positive variable has *one* such defining statement. This defining statement allows to infer the range of a variable when reasoning about queries and rules; the knowledge about further constraints on it can provide further metadata information.

Note: it wold not be appropriate to define classes PositiveVariable, NegativeVariable, etc. since a variable can be used as positiveVariable wrt. one fragment, and as a NegativeVariable wrt. another. Thus, the qualification is via the actual usage in a fragment.

Names of Variables. Variables that are local to OWLQ queries are identified by their URIs. Blank nodes (identified by rdf:nodeID) are allowed. Note that when using OWLQ in MARS rules, further conventions apply concerning the *local names*; see Section 12.3.

Joins: owlq:VariableDefinition vs. owlq:Constraint. Usually, variables ranging over objects are defined via their scope (an owlq:Class) by owlq:scopesVariable. Relationships *between* such variables are expressed by owlq:Constraint. Alternatively, the dependency of an object-valued variable from another variable can be expressed by a owlq:hasVariableDefinition. The former has the advantage that it explicitly mentions the scope (and by that can restrict it to a relevant subclass), while the second makes the dependency explicit as a potential hint to the evaluation. The user is free to use the more intuitive one (similar to SQL where equivalent queries can be stated in multiple ways, using joins or subqueries).

For literal-valued variables, these cannot given by their scope (which is a datatype, which usually do not support statements of the form of e.g. (?X a xsd:integer). They are always defined via owlq:hasVariableDefinition based on a variable ranging over objects.

Types of Constraints. Constraints relate two variables, or a variable and a constant; either via a property, or via a comparison. (owlq:PropertyConstraint, owlq:ComparisonConstraint).

- An owlq:PropertyConstraint $(v_1\ p\ v_2)$ or $(v_1\ p\ v_2)$ refers to a variable v_1, a property p (of the object bound to v_1), and another variable v_2 or a constant (object or literal).

  Examples:

```
<owlq:PropertyConstraint>
  <owlq:onVariable ref="..."/>
  <owlq:onProperty ref="..."/>
  <owlq:equalsVariable ref="..."/>
<owlq:PropertyConstraint>
<owlq:PropertyConstraint>
  <owlq:onVariable ref="..."/>
  <owlq:onProperty ref="..."/>
  <owlq:hasValue ref="..."/>
<owlq:PropertyConstraint>
<owlq:PropertyConstraint>
  <owlq:onVariable ref="..."/>
  <owlq:onProperty ref="..."/>
  <owlq:hasValue> literal </owlq:hasValue>
<owlq:PropertyConstraint>
```

  Note that a property constraint $(v_1\ p\ c)$ wrt. a constant can be expressed equivalently by a restriction hasValue$p.c$ to the class definition that scopes v_1.

- An owlq:ComparisonConstraint $(v_1\ op\ v_2)$ or $(v_1\ op\ v_2)$ refers to a variable v_1, a comparison operator p (of the object bound to v_1), and another variable v_2 or a constant (object or literal).

  Examples:

```
<owlq:ComparisonConstraint>
  <owlq:onVariable ref="..."/>
  <owlq:hasComparator ref="..."/>
  <owlq:withVariable ref="..."/>
<owlq:ComparisonConstraint>
<owlq:ComparisonConstraint>
  <owlq:onVariable ref="..."/>
  <owlq:hasComparator ref="..."/>
  <owlq:withValue> Literal </owlq:withValue>
<owlq:ComparisonConstraint>
```

  Note that comparison constraints $(v_1\ op\ c)$ wrt. a constant can be expressed equivalently by a restriction some/allValuesFrom $p.D$ to a locally defined owl:DataRange D.

  ComparisonConstraints between variables ranging over objects are only seldomly used: the ordering operators $<, \leq, >, \geq$ are not applicable. Equality can be replaced by using replacing one variable by the other and replacing its definition by a PropertyConstraint. Non-Equality can be expressed in the same way by a negated PropertyConstraint.

- both kinds of constraints can also be owlq:NegatedConstraints; the negation semantics here is that of OWL, i.e., Open World.

Accessing Local and Remote Ontologies OWLQ is intended either as query language for a local knowledge base or as standalone query language/tool that accesses a remote knowledge base. Remote ontologies can be incorporated by owlq:useOntology, referencing an HTTP resource.

The OWLQ OWL ontology.

```
# call jena -t -if owlq.n3 -ol RDF/XML -of owlq.rdf
@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 owl11:   <http://www.w3.org/2006/12/owl11#> .
@prefix mars:    <http://www.semwebtech.org/mars/2006/mars#>.
@prefix :        <http://www.semwebtech.org/languages/2006/owlq#> .

# OWLQ classes

:Expression owl:unionOf (:Query :Test :EventSpec).

:Query a owl:Class;
       rdfs:subClassOf [ a owl:Restriction;
                         owl:onProperty :resultVariable;
                         owl:minCardinality 1 ].

:Test a owl:Class;
       rdfs:subClassOf [ a owl:Restriction;
                         owl:onProperty :resultVariable;
                         owl:minCardinality 0 ].

:EventSpec a owl:Class;
       rdfs:subClassOf [ a owl:Restriction;
                         owl:onProperty :baseEvent;
                         owl:minCardinality 1 ].

:Scope a owl:Class;
  owl:equivalentClass
  [ owl11:disjointUnionOf (:Class :DataRange) ].

:Class     a owl:Class;
           rdfs:subClassOf owl:Class.
:DefinedClass  a owl:Class;
           rdfs:subClassOf :Class.

:DataRange a owl:Class;
           rdfs:subClassOf owl:DataRange.

:Variable  a owl:Class.

:VariableDefinition a owl:Class;
      rdfs:subClassOf [ a owl:Restriction; owl:onProperty :onProperty;
                        owl:allValuesFrom owl:Property ],
                      [ a owl:Restriction; owl:onProperty :onProperty;
                        owl:cardinality 1 ],
                      [ a owl:Restriction; owl:onProperty :toVariable;
                        owl:cardinality 1 ].

:Constraint owl:equivalentClass
  [ owl:unionOf ( :PropertyConstraint :ComparisonConstraint ) ].
:ComparisonConstraint owl:equivalentClass
```

```
  [ owl:unionOf
    ( :VariableComparisonConstraint :ValueComparisonConstraint)].
:PropertyConstraint a owl:Class;
      rdfs:subClassOf [ a owl:Restriction; owl:onProperty :onVariable;
                        owl:cardinality 1 ],
                      [ a owl:Restriction; owl:onProperty :onProperty;
                        owl:cardinality 1 ],
                      [ a owl:Restriction; owl:onProperty :equalsVariable;
                        owl:cardinality 1 ],
                      [ a owl:Restriction; owl:onProperty :withVariable;
                        owl:cardinality 0 ],
                      [ a owl:Restriction; owl:onProperty :withValue;
                        owl:cardinality 0 ].
:VariableComparisonConstraint a owl:Class;
      owl:intersectionOf ( :ComparisonConstraint
                      [ a owl:Restriction; owl:onProperty :onVariable;
                        owl:cardinality 1 ]
                      [ a owl:Restriction; owl:onProperty :hasComparator;
                        owl:cardinality 1 ]
                      [ a owl:Restriction; owl:onProperty :withValue;
                        owl:cardinality 0 ]
                      [ a owl:Restriction; owl:onProperty :withVariable;
                        owl:cardinality 1 ] ).
:ValueComparisonConstraint a owl:Class;
      owl:intersectionOf ( :ComparisonConstraint
                      [ a owl:Restriction; owl:onProperty :onVariable;
                        owl:cardinality 1 ]
                      [ a owl:Restriction; owl:onProperty :hasComparator;
                        owl:cardinality 1 ]
                      [ a owl:Restriction; owl:onProperty :withValue;
                        owl:cardinality 1 ]
                      [ a owl:Restriction; owl:onProperty :withVariable;
                        owl:cardinality 0 ]  ).

:NegatedConstraint  a owl:Class;
                    rdfs:subClassOf :Constraint.

:ClosedPredicate    a owl:Class;
                    rdfs:subClassOf rdf:Property.

# OWLQ predicates

:useOntology        a rdf:Property;
                    rdfs:domain :Expression; rdfs:range owl:Ontology.

:usesClass          a rdf:Property;
                    rdfs:domain :Expression; rdfs:range :Class.
:definesClass       a rdf:Property; rdfs:subPropertyOf :usesClass;
                    rdfs:domain :Expression; rdfs:range :DefinedClass.

:baseEvent          a rdf:Property; rdfs:subPropertyOf :usesClass;
                    rdfs:domain :EventSpec; rdfs:range mars:Event.

:hasConstraint      a rdf:Property;
```

```
                    rdfs:domain :Expression; rdfs:range :Constraint.

:useVariable        a rdf:Property;
                    rdfs:domain :Expression; rdfs:range :Variable.

:inputVariable      a rdf:Property; rdfs:subPropertyOf :useVariable.

:resultVariable     a rdf:Property; rdfs:subPropertyOf :useVariable.

:positiveVariable   a rdf:Property; rdfs:subPropertyOf :useVariable.

:negativeVariable   a rdf:Property; rdfs:subPropertyOf :useVariable.

:varDefinedBy       a owl:FunctionalProperty;
                    rdfs:domain :Variable.

:rangesOver         rdfs:subPropertyOf :varDefinedBy;
                    rdfs:range :Scope.

:varDefinedByVarDef rdfs:subPropertyOf :varDefinedBy;
                    rdfs:range :VariableDefinition.

:scopesVariable     a owl:InverseFunctionalProperty;
                    owl:inverseOf :rangesOver.

:hasVariableDefinition a rdf:Property;
  rdfs:domain :Variable; rdfs:range :VariableDefinition.

:hasMandatoryVariableDefinition a rdf:Property;
  rdfs:subPropertyOf :hasVariableDefinition.

:hasOptionalVariableDefinition a rdf:Property;
  rdfs:subPropertyOf :hasVariableDefinition.

:toVariable a owl:FunctionalProperty;
            owl:inverseOf :varDefinedByVarDef.
            rdfs:domain   :VariableDefinition; rdfs:range :Variable.

:onProperty a owl:FunctionalProperty;
            rdfs:domain [ owl:unionOf (:PropertyConstraint
                                       :VariableDefinition) ] ;
            rdfs:range  rdf:Property.

:hasComparator a owl:FunctionalProperty;
            rdfs:domain :ComparisonConstraint;
            rdfs:range  :Comparator.

:onVariable a owl:FunctionalProperty;
            rdfs:domain :Constraint; rdfs:range :Variable.

:equalsVariable a owl:FunctionalProperty;
                rdfs:domain :PropertyConstraint; rdfs:range :Variable.

:withVariable a owl:FunctionalProperty;
```

```
            rdfs:domain :VariableComparisonConstraint; rdfs:range :Variable.
:withValue a owl:FunctionalProperty;
           rdfs:domain :ValueComparisonConstraint; rdfs:range :Literal.

:equals a :Comparator.
:notEquals a :Comparator.
:lessThan a :Comparator.
:greaterThan a :Comparator.
:lessThanOrEqual a :Comparator.
:greaterThanOrEqual a :Comparator.
:numericLessThan a :Comparator.
:numericGreaterThan a :Comparator.
:numericLessThanOrEqual a :Comparator.
:numericGreaterThanOrEqual a :Comparator.
```

13.3.2 Evaluation of OWLQ Conjunctive Query Specifications

OWLQ queries can be translated to OWL and SPARQL. The OWL portion is added to the knowledge base and evaluated by the DL reasoner, and the SPARQL portion forms the actual query:

- OWLQ class definitions become OWL class definitions and are added to the knowledge base;
- for each variable *classvar* ranging over the instances of a class *class*, given by *class* owlq:scopesVariable *classvar*, have a conjunct {*classvar* rdf:type *class*};
- for each definition of a dependent variable var_2 associated with a variable var_1, i.e.,

  {var_1 a owlq:Variable;
  owlq:hasVariableDefinition [owlq:onProperty *property*; owlq:toVariable var_2]}

  add a conjunct (optionally optional) {var_1 *property* var_2};
- for each constraint C such that

  [a owlq:Constraint; owlq:onVariable var_1;
  owlq:onProperty *property*; owlq:equalsVariable var_2]

  add a conjunct {var_1 *property* var_2}.

The following SPARQL query extracts all information from the OWLQ query that is relevant for its translation into SPARQL (independent from the actual knowledge base, only using the OWLQ ontology and the actual query):

```
prefix owlq: <http://www.semwebtech.org/languages/2006/owlq#>
select ?CL ?CV ?P ?V1 ?V2
from <file:owlq.n3>
from <file:join-query.owlq>
where {{{?CL a owlq:Class; owlq:scopesVariable ?CV} OPTIONAL
     {?CL a owlq:Class; owlq:scopesVariable ?CV;
      owlq:hasVariableDefinition
               [ owlq:onProperty ?P; owlq:toVariable ?V1 ]}}
   UNION {[ a owlq:Constraint; owlq:onVariable ?V1;
            owlq:onProperty ?P; owlq:equalsVariable ?V2]}}
```

13.4 OWLQ in MARS

This section discusses the embedding of OWLQ into ECA rules in MARS:

- OWLQ as an Atomic Event Specification Language,
- OWLQ as Query Language, and
- OWLQ as a language for expressing tests.

For the query and test usages, two variants are investigated:

- one tuple of input variables: for each tuple, the query is evaluated;
- several tuples of input variables: the tuples are added to the ontology and the query is then evaluated.

Further examples can be found in the online demonstrator.

13.4.1 Variable Names in OWLQ Components

Since the internal mapping from OWLQ to SPARQL uses string identifiers, OWLQ requires the *local part* of the variable URI to be unique (also called *variable short name*) wrt. the rule. Thus, the usage of variable in OWLQ has the following forms:

```
<owlq:inputVariable rdf:resource="foo:bla#N" />
<owlq:inputVariable rdf:ID="N" />
<owlq:toVariable rdf:resource="#N" />
```

Note that blank nodes are not allowed as variables:

NOT ALLOWED: <owlq:scopesVariable rdf:nodeID="N" />

13.4.2 Specification of Atomic Events by OWLQ Queries

OWLQ queries can be used as a formalism for Atomic Event Specifications in ECA rules (cf. Section 5.3.2). Atomic Event Specifications in OWLQ (i) check if the event matches the specification, and if yes, (ii) generate the variable bindings:

- an AES uses one owlq:baseEvent class of events. The OWLQ AEM registers at the domain broker for events of that class. The remaining AES is then an OWLQ query against instances of this class (defining further classes, variable definitions, and constraints as described above).
- If the OWLQ query against the incoming RDF event yields at least one result, the variables are bound according to the specified owlq:resultVariable specifications.
- The event (as an RDF graph) is considered to be the functional result of the AES. If the AES occurs on the top-level of the rule, it is also bound to the implicit Event variable on the rule level (cf. Section 12.3.3).
- Note: if an AES in a rule consists only of an owlq:baseEvent declaration, any event of this class is considered to be relevant (i.e., it is delivered to the registrant in the logvars:answer portion of the detection message).

Example 53 *Atomic Event Specification in OWLQ*
The following rule reacts on cancellations of flights that start from Frankfurt:

```
<!DOCTYPE rdf:RDF [
  <!ENTITY owlq "http://www.semwebtech.org/languages/2006/owlq#">
  <!ENTITY travel "http://www.semwebtech.org/domains/2006/travel#"> ]>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:owl="http://www.w3.org/2002/07/owl#"
   xmlns:eca="http://www.semwebtech.org/languages/2006/eca-ml#"
   xmlns:owlq="http://www.semwebtech.org/languages/2006/owlq#"
   xmlns:travel="http://www.semwebtech.org/domains/2006/travel#"
   xml:base="the:rule#">

<eca:Variable rdf:ID="Flight"/>

<eca:Rule rdf:ID="the-rule">
 <eca:has-event>
  <eca:Event>
   <rdf:type rdf:resource="&owlq;EventSpec"/>
   <owlq:resultVariable rdf:resource="#Flight"/>
   <owlq:baseEvent rdf:resource="&travel;CanceledFlight"/>
   <owlq:definesClass>  <!-- auxiliary class -->
    <owlq:Class rdf:ID="FromFrankfurt">
     <owl:equivalentClass>
      <owl:Restriction>
       <owl:onProperty rdf:resource="&travel;from"/>
       <owl:hasValue rdf:resource="&travel;/airports/fra"/>
      </owl:Restriction>
     </owl:equivalentClass>
    </owlq:Class>
   </owlq:definesClass>
   <owlq:definesClass>  <!-- the class of actually relevant events -->
    <owlq:Class rdf:ID="CanceledFlightFromFrankfurt">
     <owl:equivalentClass>
      <owl:Class>
       <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="&travel;CanceledFlight"/>
        <owl:Restriction>
         <owl:onProperty rdf:resource="&travel;flight"/>
         <owl:someValuesFrom rdf:resource="#FromFrankfurt"/>
        </owl:Restriction>
       </owl:intersectionOf>
      </owl:Class>
     </owl:equivalentClass>
     <owlq:scopesVariable>
      <owlq:Variable rdf:ID="_X">
       <owlq:hasVariableDefinition>
        <owlq:VariableDefinition>
         <owlq:onProperty rdf:resource="&travel;flight"/>
         <owlq:toVariable rdf:resource="#Flight"/>
        </owlq:VariableDefinition>
       </owlq:hasVariableDefinition>
      </owlq:Variable>
     </owlq:scopesVariable>
    </owlq:Class>
   </owlq:definesClass>
```

```
  </eca:Event>
 </eca:has-event>
 <eca:has-query rdf:resource="#Q"/>
 <eca:has-action rdf:resource="#A"/>
</eca:Rule>
</rdf:RDF>
```

```
# call
# jena -q -pellet -il RDF/XML -qf rule-query-3.sparql

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX mars: <http://www.semwebtech.org/mars/2006/mars#>
PREFIX eca: <http://www.semwebtech.org/languages/2006/eca-ml#>
PREFIX owlq: <http://www.semwebtech.org/languages/2006/owlq#>
PREFIX travel: <http://www.semwebtech.org/domains/2006/travel#>

SELECT ?R ?E ?Q ?CF ?FF ?RE ?V1 ?V2 ?P
FROM <file:eca-ml.rdf>
FROM <file:owlq.rdf>
FROM <file:rule-example-3.rdf>
FROM <file:travel-miniexcerpt.rdf>
FROM <file:sample-event-and-data.rdf>
WHERE {{{?R rdf:type eca:Rule} .
        {?R eca:has-event ?E} .
        {?E rdf:type eca:EventSpec} .
        {?E rdf:type ?Q }} UNION
       {{?CF rdf:type travel:CanceledFlight} .
        {?CF travel:flight ?FF}} UNION
       {?E rdf:type mars:Event} UNION
       {?FF rdf:type <the:rule#FromFrankfurt>} UNION
       {?RE rdf:type <the:rule#CanceledFlightFromFrankfurt>} UNION
       {{?Q rdf:type owlq:EventSpec} .
        {?Q owlq:baseEvent ?C} .
        {?Q owlq:resultVariable ?V2}} UNION
       {{?C owlq:scopesVariable ?V1} .
        {?V1 owlq:hasVariableDefinition ?D} .
        {?D owlq:onProperty ?P} .
        {?D owlq:toVariable ?V2} .
        {?E rdf:type ?C} .
        {?E ?P ?FF}}}
```

The occurred "CanceledFlight" event is correctly identified as an instance of the class <*the:rule#RelevantEvent*>*. Additionally, the rule is identified to have an event specification whose base event class is* *travel:CanceledFlight*.

The last part of the query illustrates how the query is connected with the actual variables: the query specifies *the:rule#Flight* *as result variable to be bound by the event detection. The* *sample:traveldata#occur-cf-event* *is an instance of the class specified as* *eca:EventSpec*. *This class is the scope of the variable* *the:rule#Event* *that has a variable definition to define the variable* *the:rule#Flight* *as the value of its property* *travel:flight*. *The value for the given event is the URL of* *lh123*.

13.4.3 OWLQ Queries in MARS ECA Rules

The query components of an ECA rule can be stated in OWLQ, using the above language. In the setting of a rule, the input variables that obtain their values play a special rule: a query can be evaluated for a set of bindings.

- Each owlq:InputVariable must be declared in the query (by (*q* owlq:inputVariable *var-id*)).
- the input variables *must not* be scoped or defined by a class or a variable definition (the input provides their positive binding). They may occur only as subjects of variable definitions, and as objects or subjects of constraints.
- the result variables are declared in the same way as above for AESs.

Implementation. The core OWLQ functionality has been implemented in [41]; this has then been integrated within MARS. There are two variants of the query service available (cf. LSR entries):

- one accepts no variable bindings (language ID owlq-novars). The bindings must thus be replaced syntactically by the GRH; it returns each answer asynchronously (cf. LSR entry).

  In case that a rule instance with multiple variable bindings is processed, the GRH splits the bindings into each tuple, replaces the variables in the OWLQ fragment by the values and submits a separate query for each tuple to the OWLQ service. As every answer is sent back directly to the ECA engine, the answers arrive there one-by-one and asynchronously.

  As necessary for such answers, every answer tuple contains completely the input bindings to be joined with the bindings of the rule instance.

- the second one (language ID owlq) accepts a set of variable bindings in the standard MARS format. The answers are submitted as a whole asynchronously. Each tuple contains also the binding of he input variables to be joined with the bindings of the rule engine.

  This variant works as follows:

  - define a tuple variable TupVar that ranges over the tuples (as blank nodes). For every input variable v_i, TupVar has

    ```
    :Tuple owlq:scopesVariable :TupVar  .
    :TupVar a owlq:Variable;
            owlq:hasVariableDefinition
           [a owlq:VariableDefinition ;
            owlq:onProperty :var-i ;
            owlq:toVariable v_i ] .
    ```

  - For every input tuple t, a blank node b_t is created which has for every input variable v_i a property var-i whose value is the binding of v_i in the tuple:

    [a :Tuple; :var-1 v_1; ...; var-n v_n].

  - the tuples are added to the ontology facts, and the extended query is evaluated.

13.4.4 ECA-ML Conjunctive Tests in OWLQ

In contrast to queries, the idea of tests in MARS is not to use any ontology, but just to evaluate constraints between variables (probably contributed by different subqueries against different sources/ontologies) locally. For that, OWLQ provides some comparison operators (closely related to those from XPath/XQuery; see also the file `owlq.rdf` above):

```
:equals a :Comparator.
:notEquals a :Comparator.
:lessThan a :Comparator.
:greaterThan a :Comparator.
:lessThanOrEqual a :Comparator.
:greaterThanOrEqual a :Comparator.
:numericLessThan a :Comparator.
:numericGreaterThan a :Comparator.
:numericLessThanOrEqual a :Comparator.
:numericGreaterThanOrEqual a :Comparator.
```

The test part of an ECA rule is actually also an owlq:Query that in this case consists only of owlq:ComparisonConstraints using the above Comparators.

As the test component is actually a query, the syntax for the input variables is done in the same way as above for queries. For each tuple, true/false is returned which is then fed into the evaluation of the rule instance.

Implementation Comment. The evaluation of Tests in OWLQ does currently not use the actual OWLQ engine: OWL reasoners do not support datatypes. Instead, the comparison operators are translated into XPath/XQuery operators, and the evaluation of the actual test is done by the XPath engine built-in by the ECA engine.

The current implementation does only process one tuple at a time and returns the answer asynchronously (cf. LSR entry). The rule processing is then the same as described above for queries in owlq-novars.

13.5 From OWLQ Conjunctive Queries To General Queries and Formulas

The OWLQ queries and tests are conjunctive queries over class expressions and properties (comparable to F-Logic Molecules [42, 43], XPathLog terms [48, 51], or Xcerpt terms [24]. These can be true/false under given variable assignments and thus can be embedded into any language for expressing composite formulas (e.g., defining disjunction, classical negation, implication, and quantification).

More general and expressive queries can be expressed using concepts like relational algebra and relational calculus. Thus, languages for expressing composite queries over OWLQ is actually not part of OWLQ, but there can be again heterogeneous lanmguages that build upon OWLQ as atomic concepts:

- The operators of the Relational Algebra are applicable to sets of tuples of variable bindings. Straightforward extensions include grouping, aggregation, and quantification (where $\exists$ is actually a projection). Query algebras have been developed over different underlying data models, i.e., the relational one, the object-oriented one (which showed that this is not much different from the relational algebra), and F-Logic (cf. [46, Section 2.2.1]); cf. Section 5.6.

- The first-order first-order logic notions, i.e., logical or, several kinds of negation, and quantifiers can be used to express composite queries. Here, the upcoming markup&ontology proposal of the FOL-RuleML initiative [17] can be considered.

The development of an ontology for arbitrary composite queries and formulas is further work.

13.6 RDF-CL: Generating and Updating RDF Data

13.6.1 RDF-CL Core

In the same way as OWLQ, RDF-CL provides an ontology to *describe* what is to be generated. For the same reason as above, RDF/XML *patterns* in the rule are not suitable, as they are interpreted as *instances*, not as *specifications of instances*. Like OWLQ, a suitable ontology for generating OWL, or sufficiently, RDF graphs is required. This language is called *RDF-CL (RDF Creation Language)*.

- An instance of *rdf-cl:Create* is a specification of an RDF fragment to be created. It creates classes (to create instances of them) nodes (and properties of them) as specified by *rdf-cl:NodeSpec*s, and properties (between existing nodes) as specified by *rdf-cl:PropertySpec*s,
- A *rdf-cl:NodeSpec* contains the membership in one or more classes, optionally a URI and also optionally some *rdf-cl:PropertySpec*s,
- A *rdf-cl:PropertySpec* specifies a triple, given by *rdf-cl:subject*, *rdf-cl:predicate*, and *rdf-cl:object*. The subject and predicate are URIs or variables (to be bound to URIs), the object is a URI, a literal, or a variable. If a *rdf-cl:PropertySpec* is associated with a *rdf-cl:NodeSpec*, the node generated by this NodeSpec acts as subject.

```
<!DOCTYPE rdf:RDF [
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <!ENTITY owl "http://www.w3.org/2002/07/owl#"> ]>
<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#"
   xml:base="http://www.semwebtech.org/languages/2006/rdf-cl">

<owl:Class rdf:about="#Variable"/>
<owl:Class rdf:about="#Create"/>
<rdf:Property rdf:about="#defineClass">
 <rdfs:domain rdf:resource="#Create"/>
 <rdfs:range rdf:resource="#Class"/>
</rdf:Property>
<owl:Class rdf:about="#Class">
 <rdfs:subClassOf rdf:resource="&owl;Class"/>
</owl:Class>
<rdf:Property rdf:about="#createNode">
 <rdfs:domain rdf:resource="#Create"/>
 <rdfs:range rdf:resource="#NodeSpec"/>
</rdf:Property>
<rdf:Property rdf:about="#createProperty">
 <rdfs:domain rdf:resource="#Create"/>
 <rdfs:range rdf:resource="#PropertySpec"/>
</rdf:Property>

<owl:Class rdf:about="#NodeSpec"/>
<rdf:Property rdf:about="#ofClass">
 <rdfs:domain rdf:resource="#NodeSpec"/>
 <rdfs:range rdf:resource="&owl;Class"/>
</rdf:Property>
<rdf:Property rdf:about="#withURI">
```

```
 <rdfs:domain rdf:resource="#NodeSpec"/>
 <rdfs:range rdf:resource="&xsd;URI"/>
</rdf:Property>
<rdf:Property rdf:about="#withProperty">
 <rdfs:domain rdf:resource="#NodeSpec"/>
 <rdfs:range rdf:resource="#PropertySpec"/>
</rdf:Property>

<owl:Class rdf:about="#PropertySpec"/>
 <rdf:FunctionalProperty rdf:about="subject">
  <rdfs:domain rdf:resource="#PropertySpec"/>
  <rdfs:range rdf:resource="&owl;Thing"/>
 </rdf:Property>
 <rdf:FunctionalProperty rdf:about="predicate">
  <rdfs:domain rdf:resource="#PropertySpec"/>
  <rdfs:range rdf:resource="&rdf;Property"/>
  <owl:cardinality>1</owl:cardinality>
 </rdf:Property>
 <rdf:FunctionalProperty rdf:about="#object">
  <rdfs:domain rdf:resource="#PropertySpec"/>
 </rdf:Property>
</rdf:RDF>
```

---

rdf-cl:NodeSpec is a generic constructor. Its rdf-cl:ofClass property usually refers to a domain class. Each action generates *one* "handle" node that is a action of some domain ontology. The other created nodes may also be class instances of the domain that are to be generated by the action. The type of the node to be created can also be given by a variable.
rdf-cl:PropertySpec is similar to rdf:Statement, with the following deviations:

- the subject is implicit. The rdf-cl:PropertySpec has always to be linked to a rdf-cl:NodeSpec in the same action. The action must be a connected graph.
- object: variables (i.e., resources X such that X rdf:type rdf-cl:Variable) can be referenced.

13.6.2 RDF-CL as RDF Update Language

When used as a language for updating RDF documents (currently only incremental addition of facts is possible), the defined classes are added to the ontology. Created nodes can be instances of them.

13.6.3 Specification of Atomic Actions by RDF-CL

Atomic actions are RDF (often seen as RDF/XML) instances that are specified by RDF-CL. An atomic action is represented as an RDF graph in which *exactly one* node is an instance of a class (wrt. the target domain ontology) that is a subclass of mars:Action (cf. Section 6.1).

Example 54 (Action in RDF-CL) *The following RDF-CL fragments creates an instance of the action type travelns:DoBooking:*

```
<rdf-cl:Create>
 <rdf-cl:generateNode>
  <rdf-cl:NodSpec>
   <rdf-cl:ofClass ref="&travel;DoBooking"/>
   <rdf-cl:withProperty>
```

```
    <rdf-cl:PropertySpec>
     <rdf-cl:predicate ref="&travel;flight"/>
     <rdf-cl:object ref="&travel/flights/lh/lh123"/>
    <rdf-cl:PropertySpec>
   <rdf-cl:withProperty>
   <rdf-cl:withProperty>
    <rdf-cl:PropertySpec>
     <rdf-cl:predicate ref="&travel;date"/>
     <rdf-cl:object rdf:datatype="&xsd;date">01-01-2008</rdf-cl:object>
    <rdf-cl:PropertySpec>
   <rdf-cl:withProperty>
  </rdf-cl:NodSpec>
 </rdf-cl:generateNode>
</rdf:Create>
```

The generated action RDF fragment is

```
  <travel:DoBooking>
    <travel:flight ref="&travel/flights/lh/lh123"/>
    <travel:date rdf:datatype="&xsd;date">01-01-2008</travel:date>
  </travel:DoBooking>
```

Note that defined classes are rarely used in this setting. They are incorporated in the RDF graph that represents the action. The receiver can then use them for reasoning about the objects that are handled by the action. But they do not become persistent in any ontology.

13.7 The OWLQ DTD

OWLQ is an OWL language; thus its XML markup is Striped RDF. When OWLQ language fragments are serialized, or when an OWLQ engine works on an XML representation, this markup is recommended to be used. OWLQ RDF fragments can be serialized by using the following DTD as Striped DTD and as Stripped DTD (cf. Section 11.4.1).

```
<!ELEMENT Query (useOntology*,
     inputVariable*, resultVariable*, useVariable*,
     usesClass*, definesClass*,
     hasVariableDefinition*, hasConstraint*)>
<!ELEMENT EventSpec (useOntology*,
     inputVariable*, resultVariable*, useVariable*,
     baseEvent, usesClass*, definesClass*,
     hasVariableDefinition*, hasConstraint*)>
<!ELEMENT Test (useOntology*, inputVariable*, hasConstraint*)>

<!ATTLIST Test rdf:about CDATA #IMPLIED>
<!ATTLIST Query rdf:about CDATA #IMPLIED>
<!ATTLIST EventSpec rdf:about CDATA #IMPLIED>

<!ELEMENT useOntology EMPTY>
   <!ATTLIST useOntology rdf:resource CDATA #REQUIRED >

<!-- variable usage -->
<!ELEMENT useVariable (Variable?)>
   <!ATTLIST useVariable rdf:resource CDATA #IMPLIED >
<!ELEMENT inputVariable EMPTY>
```

```
   <!ATTLIST inputVariable rdf:resource CDATA #REQUIRED >
<!ELEMENT resultVariable EMPTY>
   <!ATTLIST resultVariable rdf:resource CDATA #REQUIRED >

<!ELEMENT usesClass (Class?)>
   <!ATTLIST definesClass rdf:resource CDATA #IMPLIED >
<!ELEMENT definesClass EMPTY>
   <!ATTLIST definesClass rdf:resource CDATA #REQUIRED >
<!ELEMENT baseEvent EMPTY>
   <!ATTLIST baseEvent rdf:resource CDATA #REQUIRED >
<!ELEMENT Class (owl:sameAs?, scopesVariable*)>
   <!ATTLIST Class rdf:about CDATA #IMPLIED >
<!ELEMENT scopesVariable (Variable?)>
   <!ATTLIST scopesVariable rdf:resource CDATA #REQUIRED >

<!ELEMENT Variable (hasVariableDefinition*)>
   <!ATTLIST Variable rdf:about CDATA #REQUIRED >
<!ELEMENT hasVariableDefinition (VariableDefinition)>
<!ELEMENT hasConstraint (NegatedConstraint | Constraint)>

<!ELEMENT VariableDefinition (onProperty, toVariable)>
   <!ATTLIST VariableDefinition rdf:about CDATA #IMPLIED >
<!ELEMENT Constraint (onVariable,
     onProperty?, hasComparator?,
     equalsVariable?, toVariable?, withVariable?, withValue?)>
   <!ATTLIST Constraint rdf:about CDATA #IMPLIED >
<!ELEMENT NegatedConstraint (onVariable,
     onProperty?, hasComparator?,
     equalsVariable?, toVariable?, withVariable?, withValue?)>
   <!ATTLIST NegatedConstraint rdf:about CDATA #IMPLIED >

<!ELEMENT onVariable EMPTY>
   <!ATTLIST onVariable rdf:resource CDATA #REQUIRED >
<!ELEMENT onProperty EMPTY>
   <!ATTLIST onProperty rdf:resource CDATA #REQUIRED >
<!ELEMENT hasComparator EMPTY>
   <!ATTLIST hasComparator rdf:resource CDATA #REQUIRED >
<!ELEMENT toVariable EMPTY>
   <!ATTLIST toVariable rdf:resource CDATA #REQUIRED >
<!ELEMENT equalsVariable EMPTY>
   <!ATTLIST equalsVariable rdf:resource CDATA #REQUIRED >
<!ELEMENT withVariable EMPTY>
   <!ATTLIST withVariable rdf:resource CDATA #REQUIRED >
<!ELEMENT withValue (#PCDATA)*>
   <!ATTLIST withValue rdf:resource CDATA #IMPLIED >
<!ELEMENT owl:sameAs (DefinedClass)>
   <!ATTLIST owl:sameAs rdf:resource CDATA #IMPLIED >
<!-- to be extended -->
<!ELEMENT DefinedClass
 ( owl:unionOf | owl:disjointUnionOf | owl:intersectionOf |
   owl:complementOf | owl:equivalentClass )>
<!ELEMENT owl:unionOf (rdf:List)>
<!ELEMENT owl:disjointUnionOf (rdf:List)>
<!ELEMENT owl:intersectionOf (rdf:List)>
```

```
<!ELEMENT owl:complementOf (rdf:List)>
<!ELEMENT rdf:List (rdf:first, rdf:rest?)>
<!ELEMENT rdf:first (owl:Restriction?)>
   <!ATTLIST rdf:first rdf:resource CDATA #IMPLIED >
<!ELEMENT rdf:rest (rdf:List)>
<!ELEMENT owl:equivalentClass (owl:Restriction)>
<!ELEMENT owl:Restriction (owl:onProperty, owl11:onClass?,
   owl:someValuesFrom?, owl:allValuesFrom?, owl:hasValue?,
   owl:cardinality?, owl:minCardinality?, owl:maxCardinality?)>
<!ELEMENT owl:Class EMPTY>
   <!ATTLIST owl:Class rdf:about CDATA #IMPLIED >
<!ELEMENT owl:onProperty EMPTY>
   <!ATTLIST owl:onProperty rdf:resource CDATA #IMPLIED >
<!ELEMENT owl11:onClass EMPTY>
   <!ATTLIST owl11:onClass rdf:resource CDATA #IMPLIED >
<!ELEMENT owl:someValuesFrom EMPTY>
   <!ATTLIST owl:someValuesFrom rdf:resource CDATA #IMPLIED >
<!ELEMENT owl:allValuesFrom EMPTY>
   <!ATTLIST owl:allValuesFrom rdf:resource CDATA #IMPLIED >
<!ELEMENT owl:hasValue EMPTY>
   <!ATTLIST owl:hasValue rdf:resource CDATA #IMPLIED >
<!ELEMENT owl:cardinality (#PCDATA)>
<!ELEMENT owl:minCardinality (#PCDATA)>
<!ELEMENT owl:maxCardinality (#PCDATA)>
```

Chapter 14

Composite Events and Composite Actions

This section presents two reference language ontologies: SNOOP for composite events and CCS for composite actions. Any other languages can be designed in the same way:

- define the language ontology in RDF/OWL,
- register an appropriate service, providing the required tasks, with the LSR,
- if communication is XML-based, then a striped and if necessary, a stripped DTD must be provided.

14.1 The Snoop Ontology

The Snoop classes are the operators. Note that the Snoopy ontology does not refer at all to the ECA-ML ontology since it is an independent language. Note also that what SNOOP calls "Events" are actually not events, but event specifications.
Snoop defines the following classes:

- Bag-Events: And, Or, Any, MultiOccurrence. Property: has-subevent,
- Any-or-Multi: Composers with counting (Any and MultiOccurrence), property: number-of-occurrences.
- Bounded: event definitions that are expressed by an interval with something to be checked in the interval (Not, Cumulative/Non-Cumulative (A)Periodic). Subsets: Cumulative, Non-Cumulative, Not.

  The interval is given by has-start, has-end. The relevant events in the interval are given by the properties cumulate (Cumulative), fired-by (Non-Cumulative) and exit-not (Not).
- Sequence: with its unbounded cardinality and ordered semantics, this operator is a problem in RDF (whereas in XML it is trivial since the contents is ordered).

  The current proposal is that Sequence can be modeled in two ways:

  – with two subevents first and second (that can be sequences again for encoding longer sequences).

  – long sequences are cumbersome in that modeling. For that, subevents (of Sequences) can be serialized by a ccs:followedBy property that refers to the next subevent.

Classes of Expressions vs. Operators As an algebraic language, SNOOP defines operators with arities. By that, "Sequence" is also an operator, and as above a class of terms induced by that operator.

14.1.1 The Snoopy OWL File

```
<!DOCTYPE rdf:RDF [
  <!ENTITY mars "http://www.semwebtech.org/mars/2006/mars#">
  <!ENTITY expr "http://www.semwebtech.org/mars/2006/expressions#"> ]>

<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:expr="http://www.semwebtech.org/mars/2006/expressions#"
   xml:base="http://www.semwebtech.org/languages/2006/snoopy">

<owl:Ontology rdf:about="#"/>

<owl:Class rdf:about="#Event">
 <rdfs:isDefinedBy rdf:resource="#"/>
</owl:Class>

<owl:Class rdf:about="#AtomicEvent">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#Event"/>
 <owl:disjointWith rdf:resource="#CompositeEvent"/>
</owl:Class>

<owl:Class rdf:about="#CompositeEvent">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#Event"/>
 <rdfs:subClassOf rdf:resource="&expr;AlgebraicCompositeExpression"/>
 <owl:disjointWith rdf:resource="#AtomicEvent"/>
 <owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#Event"/>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#has-operator"/>
   <owl:cardinality>1</owl:cardinality>
  </owl:Restriction>
 </owl:intersectionOf>
</owl:Class>

<owl:Class rdf:about="#Opaque">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <owl:equivalentClass rdf:resource="#OpaqueEvent"/>
 <rdfs:subClassOf rdf:resource="#Event"/>
 <owl:disjointWith rdf:resource="#AtomicEvent"/>
 <owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#Event"/>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#opaqueSpec"/>
   <owl:cardinality>1</owl:cardinality>
  </owl:Restriction>
```

```
 </owl:intersectionOf>
</owl:Class>

<!-- events expressed in other languages -->
<!-- not that these can be equivalent to a snoop one
  of the operator has a snoop equivalent -->
<owl:Class rdf:about="#ExternalEvent"/>
<!-- detection if something is an external event requires
  nonmonotonic or closed world reasoning; it is only
  required when translating to the XML markup -->

<owl:FunctionalProperty rdf:about="#has-operator">
 <rdfs:domain rdf:resource="#CompositeEvent"/>
 <rdfs:range rdf:resource="#Operator"/>
 <rdfs:subPropertyOf rdf:resource="&expr;has-operator"/>
</owl:FunctionalProperty>

<owl:DatatypeProperty rdf:about="#opaqueSpec">
 <rdfs:domain rdf:resource="#Opaque"/>
</owl:DatatypeProperty>

<rdf:Property rdf:about="#language">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:domain rdf:resource="#Opaque"/>
 <rdfs:range rdf:resource="&mars;Language"/>
</rdf:Property>

<expr:Operator rdf:about="#AndOp">
 <expr:has-arity>2</expr:has-arity>
</expr:Operator>
<expr:Operator rdf:about="#OrOp">
 <expr:has-arity>2</expr:has-arity>
</expr:Operator>
<expr:Operator rdf:about="#AnyOp">
 <expr:has-arity>*</expr:has-arity>
</expr:Operator>
<expr:Operator rdf:about="#MultiOccurrencesOp">
 <expr:has-arity>1</expr:has-arity>
</expr:Operator>
<expr:Operator rdf:about="#SequenceOp">
 <expr:has-arity>2</expr:has-arity>
</expr:Operator>
<expr:Operator rdf:about="#NotOp">
 <expr:has-arity>3</expr:has-arity>
</expr:Operator>
<expr:Operator rdf:about="#PeriodicOp">
 <expr:has-arity>3</expr:has-arity>
</expr:Operator>
<expr:Operator rdf:about="#CumulativePeriodicOp">
 <expr:has-arity>3</expr:has-arity>
</expr:Operator>
<expr:Operator rdf:about="#AperiodicOp">
 <expr:has-arity>3</expr:has-arity>
</expr:Operator>
```

```
<expr:Operator rdf:about="#CumulativeAperiodicOp">
 <expr:has-arity>3</expr:has-arity>
</expr:Operator>

<owl:Class rdf:about="#Sequence">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#CompositeEvent"/>
 <owl:disjointWith rdf:resource="#BagEvent"/>
 <owl:disjointWith rdf:resource="#Bounded"/>
 <owl:equivalentClass>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#has-operator"/>
   <owl:hasValue rdf:resource="#SequenceOp"/>
  </owl:Restriction>
 </owl:equivalentClass>
</owl:Class>

<rdf:Property rdf:about="#has-subevent">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:domain rdf:resource="#CompositeEvent"/>
 <rdfs:range rdf:resource="#Event"/>
 <owl:inverseOf rdf:resource="#subEventOf"/>
</rdf:Property>

<owl:FunctionalProperty rdf:about="#first">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subPropertyOf rdf:resource="#has-subevent"/>
 <rdfs:domain rdf:resource="#Sequence"/>
 <rdfs:range rdf:resource="#Event"/>
 <owl:inverseOf rdf:resource="#firstOf"/>
</owl:FunctionalProperty>

<owl:FunctionalProperty rdf:about="#second">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subPropertyOf rdf:resource="#has-subevent"/>
 <rdfs:domain rdf:resource="#Sequence"/>
 <rdfs:range rdf:resource="#Event"/>
 <owl:inverseOf rdf:resource="#secondOf"/>
</owl:FunctionalProperty>

<owl:FunctionalProperty rdf:about="#followed-by">
 <rdfs:domain rdf:resource="#Event"/>
 <rdfs:range rdf:resource="#Event"/>
 <owl:inverseOf rdf:resource="#follows"/>
</owl:FunctionalProperty>

<owl:Class rdf:about="#BagEvent">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#CompositeEvent"/>
 <owl:unionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#And">
   <rdfs:isDefinedBy rdf:resource="#"/>
   <owl:disjointWith rdf:resource="#Or"/>
   <owl:disjointWith rdf:resource="#AnyOrMulti"/>
```

```
   <owl:equivalentClass>
    <owl:Restriction>
     <owl:onProperty rdf:resource="#has-operator"/>
     <owl:hasValue rdf:resource="#AndOp"/>
    </owl:Restriction>
   </owl:equivalentClass>
  </owl:Class>
  <owl:Class rdf:about="#Or">
   <rdfs:isDefinedBy rdf:resource="#"/>
   <owl:disjointWith rdf:resource="#And"/>
   <owl:disjointWith rdf:resource="#AnyOrMulti"/>
   <owl:equivalentClass>
    <owl:Restriction>
     <owl:onProperty rdf:resource="#has-operator"/>
     <owl:hasValue rdf:resource="#OrOp"/>
    </owl:Restriction>
   </owl:equivalentClass>
  </owl:Class>
  <owl:Class rdf:about="#AnyOrMulti">
   <rdfs:isDefinedBy rdf:resource="#"/>
   <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Any">
     <rdfs:isDefinedBy rdf:resource="#"/>
     <owl:disjointWith rdf:resource="#MultiOccurrences" />
     <owl:equivalentClass>
      <owl:Restriction>
       <owl:onProperty rdf:resource="#has-operator"/>
       <owl:hasValue rdf:resource="#AnyOp"/>
      </owl:Restriction>
     </owl:equivalentClass>
    </owl:Class>
    <owl:Class rdf:about="#MultiOccurrences">
     <rdfs:isDefinedBy rdf:resource="#"/>
     <owl:disjointWith rdf:resource="#Any" />
     <owl:equivalentClass>
      <owl:Restriction>
       <owl:onProperty rdf:resource="#has-operator"/>
       <owl:hasValue rdf:resource="#MultiOccurrencesOp"/>
      </owl:Restriction>
     </owl:equivalentClass>
    </owl:Class>
   </owl:unionOf>
   <owl:disjointWith rdf:resource="#And"/>
   <owl:disjointWith rdf:resource="#Or"/>
  </owl:Class>
 </owl:unionOf>
 <owl:disjointWith rdf:resource="#Sequence"/>
 <owl:disjointWith rdf:resource="#Bounded"/>
</owl:Class>

<rdf:Property rdf:about="#number-of-occurrences">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:range rdf:resource="#AnyOrMulti"/>
 <rdfs:domain rdf:resource="http://www.w3.org/2001/XMLSchema#integer"/>
```

```
</rdf:Property>

<owl:Class rdf:about="#Bounded">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#CompositeEvent"/>
 <owl:unionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#Not">
   <rdfs:isDefinedBy rdf:resource="#"/>
   <owl:disjointWith rdf:resource="#Cumulative"/>
   <owl:disjointWith rdf:resource="#NonCumulative"/>
   <owl:equivalentClass>
    <owl:Restriction>
     <owl:onProperty rdf:resource="#has-operator"/>
     <owl:hasValue rdf:resource="#NotOp"/>
    </owl:Restriction>
   </owl:equivalentClass>
  </owl:Class>
  <owl:Class rdf:about="#Cumulative">
   <rdfs:isDefinedBy rdf:resource="#"/>
   <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#CumulativeAperiodic">
     <rdfs:isDefinedBy rdf:resource="#"/>
     <owl:disjointWith rdf:resource="#CumulativePeriodic"/>
     <owl:equivalentClass>
      <owl:Restriction>
       <owl:onProperty rdf:resource="#has-operator"/>
       <owl:hasValue rdf:resource="#CumulativePeriodicOp"/>
      </owl:Restriction>
     </owl:equivalentClass>
    </owl:Class>
    <owl:Class rdf:about="#CumulativePeriodic">
     <rdfs:isDefinedBy rdf:resource="#"/>
     <owl:disjointWith rdf:resource="#CumulativeAperiodic"/>
     <owl:equivalentClass>
      <owl:Restriction>
       <owl:onProperty rdf:resource="#has-operator"/>
       <owl:hasValue rdf:resource="#CumulativeAperiodicOp"/>
      </owl:Restriction>
     </owl:equivalentClass>
    </owl:Class>
   </owl:unionOf>
   <owl:disjointWith rdf:resource="#Not"/>
   <owl:disjointWith rdf:resource="#NonCumulative"/>
  </owl:Class>
  <owl:Class rdf:about="#NonCumulative">
   <rdfs:isDefinedBy rdf:resource="#"/>
   <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Aperiodic">
     <rdfs:isDefinedBy rdf:resource="#"/>
     <owl:disjointWith rdf:resource="#Periodic"/>
     <owl:equivalentClass>
      <owl:Restriction>
       <owl:onProperty rdf:resource="#has-operator"/>
       <owl:hasValue rdf:resource="#PeriodicOp"/>
```

```
      </owl:Restriction>
     </owl:equivalentClass>
    </owl:Class>
    <owl:Class rdf:about="#Periodic">
     <rdfs:isDefinedBy rdf:resource="#"/>
     <owl:disjointWith rdf:resource="#Aperiodic"/>
     <owl:equivalentClass>
      <owl:Restriction>
       <owl:onProperty rdf:resource="#has-operator"/>
       <owl:hasValue rdf:resource="#AperiodicOp"/>
      </owl:Restriction>
     </owl:equivalentClass>
    </owl:Class>
   </owl:unionOf>
   <owl:disjointWith rdf:resource="#Not"/>
   <owl:disjointWith rdf:resource="#Cumulative"/>
  </owl:Class>
 </owl:unionOf>
 <owl:disjointWith rdf:resource="#BagEvent"/>
 <owl:disjointWith rdf:resource="#Sequence"/>
</owl:Class>

<rdf:Property rdf:about="#has-start">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:domain rdf:resource="#Bounded"/>
 <rdfs:range rdf:resource="#Event"/>
</rdf:Property>

<rdf:Property rdf:about="#has-end">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:domain rdf:resource="#Bounded"/>
 <rdfs:range rdf:resource="#Event"/>
</rdf:Property>

<rdf:Property rdf:about="#cumulate">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:domain rdf:resource="#Cumulative"/>
 <rdfs:range rdf:resource="#Event"/>
</rdf:Property>

<rdf:Property rdf:about="#fired-by">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:domain rdf:resource="#NonCumulative"/>
 <rdfs:range rdf:resource="#Event"/>
</rdf:Property>

<rdf:Property rdf:about="#exit-not">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:domain rdf:resource="#Not"/>
 <rdfs:range rdf:resource="#Event"/>
</rdf:Property>
</rdf:RDF>
```

```
# call
# jena -q -pellet -il RDF/XML -qf snoopy-query.sparql

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 snoop: <http://www.semwebtech.org/languages/2006/snoopy#>

SELECT ?X ?Y
FROM <file:snoopy.rdf>
WHERE {
 ?X rdfs:subClassOf ?Y . ?Y rdfs:subClassOf snoop:Event
 }
```

14.1.2 The XML Representation: Snoopy-ML

The XML representation follows the term structure and omits the term construction properties. The order of arguments (when relevant, such as for cumulative and not events) corresponds to that of the original SNOOP event algebra. As motivated above, the mapping is given implicitly via the target DTDs for Striped RDF/XML SNOOP and SNOOP-ML (see Appendix C.2).

14.2 The CCS Process Ontology

The CCS ontology is analogous to the SNOOP ontology, using the CCS operators as described in the DTD given in Section 5.8.

```
<!DOCTYPE rdf:RDF [
  <!ENTITY expr "http://www.semwebtech.org/2006/expressions#"> ]>
<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:expr="http://www.semwebtech.org/mars/2006/expressions#"
   xml:base="http://www.semwebtech.org/languages/2006/ccs">

<owl:Ontology rdf:about="#"/>

<owl:Class rdf:about="#CCSExpr">
 <owl:unionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#Process">
   <rdfs:isDefinedBy rdf:resource="#"/>
  </owl:Class>
  <owl:Class rdf:about="#Event"> <!-- see also glue ontology below -->
   <owl:equivalentClass rdf:resource="#EventSpec"/>
   <owl:disjointWith rdf:resource="#Process"/>
  </owl:Class>
  <owl:Class rdf:about="#Query">
   <owl:disjointWith rdf:resource="#Process"/>
   <owl:disjointWith rdf:resource="#Event"/>
  </owl:Class>
  <owl:Class rdf:about="#Test">
   <owl:disjointWith rdf:resource="#Process"/>
```

```
   <owl:disjointWith rdf:resource="#Event"/>
   <owl:disjointWith rdf:resource="#Query"/>
  </owl:Class>
 </owl:unionOf>
</owl:Class>

<owl:Class rdf:about="#AtomicAction">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#Process"/>
 <owl:disjointWith rdf:resource="#CompositeProcess"/>
</owl:Class>

<owl:Class rdf:about="#CompositeProcess">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#Process"/>
 <rdfs:subClassOf rdf:resource="&expr;AlgebraicCompositeExpression"/>
</owl:Class>

<owl:FunctionalProperty rdf:about="#has-operator">
 <rdfs:domain rdf:resource="#CompositeProcess"/>
 <rdfs:range rdf:resource="#Operator"/>
 <rdfs:subPropertyOf rdf:resource="&expr;has-operator"/>
</owl:FunctionalProperty>

<expr:Operator rdf:about="#SequenceOp">
 <expr:has-arity>*</expr:has-arity>
</expr:Operator>
<expr:Operator rdf:about="#AlternativeOp">
 <expr:has-arity>*</expr:has-arity>
</expr:Operator>
<expr:Operator rdf:about="#ConcurrentOp">
 <expr:has-arity>*</expr:has-arity>
</expr:Operator>
<expr:Operator rdf:about="#FixpointOp">
 <expr:has-arity>1</expr:has-arity>
</expr:Operator>

<owl:Class rdf:about="#Delay">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#AtomicAction"/>
</owl:Class>
<owl:Class rdf:about="#Sequence">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#CompositeProcess"/>
 <owl:equivalentClass>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#has-operator"/>
   <owl:hasValue rdf:resource="#SequenceOp"/>
  </owl:Restriction>
 </owl:equivalentClass>
</owl:Class>
<owl:Class rdf:about="#Alternative">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#CompositeProcess"/>
```

```
 <owl:disjointWith rdf:resource="#Sequence"/>
 <owl:equivalentClass>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#has-operator"/>
   <owl:hasValue rdf:resource="#AlternativeOp"/>
  </owl:Restriction>
 </owl:equivalentClass>
</owl:Class>
<owl:Class rdf:about="#Concurrent">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#CompositeProcess"/>
 <owl:disjointWith rdf:resource="#Sequence"/>
 <owl:disjointWith rdf:resource="#Alternative"/>
 <owl:equivalentClass>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#has-operator"/>
   <owl:hasValue rdf:resource="#ConcurrentOp"/>
  </owl:Restriction>
 </owl:equivalentClass>
</owl:Class>
<owl:Class rdf:about="#Fixpoint">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subClassOf rdf:resource="#CompositeProcess"/>
 <owl:disjointWith rdf:resource="#Sequence"/>
 <owl:disjointWith rdf:resource="#Alternative"/>
 <owl:disjointWith rdf:resource="#Concurrent"/>
 <owl:equivalentClass>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#has-operator"/>
   <owl:hasValue rdf:resource="#FixpointOp"/>
  </owl:Restriction>
 </owl:equivalentClass>
 <owl:equivalentClass>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#fix-variable"/>
   <owl:minCardinality>1</owl:minCardinality>
  </owl:Restriction>
 </owl:equivalentClass>
</owl:Class>

<rdf:Property rdf:about="#fix-variable">
 <rdfs:domain rdf:resource="#Fixpoint"/>
 <rdfs:range rdf:resource="#ProcessVariable"/>
</rdf:Property>
<rdf:Property rdf:about="#local-variable">
 <rdfs:domain rdf:resource="#Fixpoint"/>
 <rdfs:range rdf:resource="#Variable"/>
</rdf:Property>
<owl:DatatypeProperty rdf:about="#has-index">
 <rdfs:domain rdf:resource="#Fixpoint"/>
</owl:DatatypeProperty>

<owl:Class rdf:about="#Variable"/>
<owl:Class rdf:about="#ProcessVariable">
```

```
 <owl:disjointWith rdf:resource="#Variable"/>
</owl:Class>

<rdf:Property rdf:about="#has-subprocess">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:domain rdf:resource="#CompositeProcess"/>
 <rdfs:range rdf:resource="#CCSExpr"/>
 <owl:inverseOf rdf:resource="#subProcessOf"/>
</rdf:Property>

<owl:FunctionalProperty rdf:about="#first">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subPropertyOf rdf:resource="#has-subprocess"/>
 <rdfs:domain rdf:resource="#Sequence"/>
 <rdfs:range rdf:resource="#FirstInSequence"/>
 <owl:inverseOf rdf:resource="#firstOf"/>
</owl:FunctionalProperty>

<owl:FunctionalProperty rdf:about="#second">
 <rdfs:isDefinedBy rdf:resource="#"/>
 <rdfs:subPropertyOf rdf:resource="#has-subprocess"/>
 <rdfs:domain rdf:resource="#Sequence"/>
 <rdfs:range rdf:resource="#SecondInSequence"/>
 <owl:inverseOf rdf:resource="#secondOf"/>
</owl:FunctionalProperty>

<owl:FunctionalProperty rdf:about="#followed-by">
 <rdfs:domain rdf:resource="#CCSExpr"/>
 <rdfs:range rdf:resource="#CCSExpr"/>
 <owl:inverseOf rdf:resource="#follows"/>
</owl:FunctionalProperty>
</rdf:RDF>
```

Chapter 15

Rules and Rule Components as Resources

Using the above ontologies, every rule is interpreted as a network of RDF resources of the contributing ontologies (ECA, event algebras, OWLQ specifications, application domains etc.). RDF/OWL reasoning can e.g. be applied for analysis of stuctural correctness, safety of variables, evaluation order etc. The chapter starts with some examples and then sketches reasoning about rules (which is still future work).

15.1 Putting the Ontologies Together

The composition of rules from elements of different ontologies, such as the ECA-ML ontology, composite event ontologies, process ontologies, query language ontologies, domain ontologies and specification of patterns about domain notions follows the structure given in the previous chapters, e.g. depicted in Figures 3.7 and 5.1. The individual ontologies are independent from each other.

For composing them in a *semantic* way, they should either refer to the notions in the MARS ontology, or sepadate *glue* ontologies are needed to describe the relationships of their notions with the MARS notions. This is mostly done by making rdfs:subClassOf, rdf:subPropertyOf, owl:equivalentClass, or owl:disjointWith statements.

The rule becomes then a semantic structure that allows for evaluating/executing it as already described for the XML level, and for *reasoning* about it. First, some examples of RDF-level rules are shown.

Example 55 *Consider again the university example (Example 33) which combines three application-independent language ontologies. Now, we give the RDF/XML representation:*

- *the ECA ontology,*
- *the SNOOP ontology of the event algebra in the event part,*
- *the CCS ontology of the process algebra in the action part,*
- *the OWQL ontology for queries for representing atomic event specification, queries and tests,*

and an application-dependent ontology:

- *the university application ontology: there, the semantics of the atomic events defined in this ontology must be available (diagonally crosshatched).*

This point of view leads to the above service-oriented distributed architecture by associating the "responsibility" for handling the respective resources by appropriate services. Additionally, e.g. collections of (sub)events as well as complete (application-specific) rule bases can be designed, published by associating them with a URI, and reused.

```
<!DOCTYPE rdf:RDF [
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <!ENTITY owlq "http://www.semwebtech.org/languages/2006/owlq#">
  <!ENTITY fml "http://www.semwebtech.org/languages/2006/fml#">
  <!ENTITY uni "http://www.semwebtech.org/domains/2006/uni#">
  <!ENTITY smtp "http://www.semwebtech.org/domains/2006/smtp#">
  <!ENTITY httpget "http://www.semwebtech.org/languages/2006/http-get#"> ]>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:eca="http://www.semwebtech.org/languages/2006/eca-ml#"
   xmlns:snoop="http://www.semwebtech.org/languages/2006/snoopy#"
   xmlns:ccs="http://www.semwebtech.org/languages/2006/ccs#"
   xmlns:owlq="http://www.semwebtech.org/languages/2006/owlq#"
   xmlns:rdf-cl="http://www.semwebtech.org/languages/2006/rdf-cl#"
   xmlns:fml="http://www.semwebtech.org/languages/2006/fml#"
   xmlns:uni="http://www.semwebtech.org/domains/2006/uni#"
   xmlns:smtp="http://www.semwebtech.org/domains/2006/smtp#"
   xml:base="foo:rule">

<eca:Rule>

 <!-- the following variable declarations are redundant
  <eca:uses-variable rdf:resource="#student"/>
  <eca:uses-variable rdf:resource="#subject"/>
  <eca:uses-variable rdf:resource="#student-email"/>
  <eca:uses-variable rdf:resource="#secondfailure"/>
  <eca:uses-variable rdf:resource="#lecturer"/>
  <eca:uses-variable rdf:resource="#lecturer-email"/>  -->
 <eca:has-event>
  <snoop:Sequence>
   <snoop:bind-to-variable rdf:resource="#event"/>
   <snoop:first>
    <owlq:EventSpec>
     <owl:baseEvent>
      <owlq:Class rdf:about="&uni;Failure">
       <owlq:scopesVariable>
        <owlq:Variable rdf:about="#F">
         <!-- the variables are join variables between the two
              relevant occurrences of the event -->
         <owlq:hasVariableBinding>
          <owlq:onProperty rdf:resource="&uni;student"/>
          <owlq:toVariable rdf:resource="#student"/>
         </owlq:hasVariableBinding>
         <owlq:hasVariableBinding>
          <owlq:onProperty rdf:resource="&uni;subject"/>
          <owlq:toVariable rdf:resource="#subject"/>
         </owlq:hasVariableBinding>
        </owlq:Variable rdf:about="#F">
       </owlq:scopesVariable>
      </owlq:Class>
     </owl:baseEvent>
    </owlq:EventSpec>
   </snoop:first>
   <snoop:second>
```

```
   <owlq:EventSpec>
    <owl:baseEvent>
     <owlq:Class rdf:about="&uni;Failure">
      <owlq:scopesVariable>
       <owlq:Variable rdf:about="#F">
        <owlq:hasVariableBinding>
         <owlq:onProperty rdf:resource="&uni;student"/>
         <owlq:toVariable rdf:resource="#student"/>
        </owlq:hasVariableBinding>
        <owlq:hasVariableBinding>
         <owlq:onProperty rdf:resource="&uni;subject"/>
         <owlq:toVariable rdf:resource="#subject"/>
        </owlq:hasVariableBinding>
       </owlq:Variable rdf:about="#F">
      </owlq:scopesVariable>
     </owlq:Class>
    </owl:baseEvent>
   </owlq:EventSpec>
  </snoop:second>
 </snoop:Sequence>
</eca:has-event>

<!-- get e-mail adresses of student and lecturer -->
<eca:has-query>
 <owlq:Query>
  <owlq:hasVariableBinding>
   <owlq:VariableBinding>
    <owlq:onVariable rdf:resource="#student"/>
    <owlq:onProperty rdf:resource="&uni;has-email"/>
    <owlq:toVariable rdf:resource="#student-email"/>
   </owlq:VariableBinding>
   <!-- get the lecturer of the 2nd exam in the
     multi-occurrences -->
  </owlq:VariableBinding>
  <owlq:hasVariableBinding>
   <owlq:VariableBinding>
    <owlq:onVariable rdf:resource="#event"/>
    <owlq:onProperty rdf:resource="\_2"/>
    <owlq:equalsVariable rdf:resource="#secondfailure"/>
   </owlq:VariableBinding>
  </owlq:hasVariableBinding>
  <owlq:hasVariableBinding>
   <owlq:VariableBinding>
    <owlq:onVariable rdf:resource="#secondfailure"/>
    <owlq:onProperty rdf:resource="&uni;lecturer"/>
    <owlq:equalsVariable rdf:resource="#lecturer"/>
   </owlq:VariableBinding>
  </owlq:VariableBinding>
  <owlq:VariableBinding>
   <owlq:VariableBinding>
    <owlq:onVariable rdf:resource="#lecturer"/>
    <owlq:onProperty rdf:resource="&uni;has-email"/>
    <owlq:equalsVariable rdf:resource="#lecturer-email"/>
   </owlq:VariableBinding>
```

```
  </owlq:VariableBinding>
 </owlq:Query>
</eca:has-query>

<!-- was the second failure a written or oral exam? -->
<!-- this could aternatively be encoded already in the
     second failure event -->
<eca:has-test>
 <owlq:Constraint>
  <owlq:onVariable rdf:resource="#secondfailure"/>
  <owlq:onProperty rdf:resource="&rdf;type"/>
  <owlq:hasValue rdf:resource="&uni;oral-exam"/>
 </owlq:Constraint>
</eca:has-test>

<!-- start process -->
<eca:has-action>
 <ccs:Sequence>
  <ccs:has-subprocess>
   <ccs:Fixpoint>
    <ccs:fix-variable rdf:resource="#X"/>
    <ccs:has-index>1</ccs:has-index>
    <ccs:local-variable rdf:resource="#datetime"/>
    <ccs:local-variable rdf:resource="#time"/>
    <ccs:local-variable rdf:resource="#room"/>
    <ccs:has-body>
     <ccs:Sequence>
      <ccs:has-subprocess>
       <!-- from the smtp domain metadata it is clear that this is a mars:Action -->
       <rdf-cl:NodeSpec rdf:nodeID="seq2.1">
        <rdf-cl:ofClass rdf:resource="&smtp;Send-Mail"/>
        <rdf-cl:with-property>
         <rdf-cl:PropertySpec>
          <rdf-cl:property rdf:resource="&smtp;to"/>
          <rdf-cl:value-of rdf:resource="#lecturer-email"/>
         </rdf-cl:PropertySpec>
        </rdf-cl:with-property>
        <rdf-cl:with-property>
         <rdf-cl:PropertySpec>
          <rdf-cl:property rdf:resource="&smtp;text"/>
          <rdf-cl:value>Please suggest appointment for {#student}, {#subject}</rdf-cl:value>
         </rdf-cl:PropertySpec>
        </rdf-cl:with-property>
        <ccs:followedBy rdf:nodeID="seq2.2"/>
       </rdf-cl:NodeSpec>
      </ccs:has-subprocess>
      <ccs:has-subprocess>
       <owlq:Query rdf:nodeID="seq2.2">
        <!-- evaluation binds $datetime -->
        <owlq:hasResultClass>
         <owlq:Class xmlns:owlq="http://www.semwebtech.org/languages/2006/owlq#"
           rdf:about="&uni;AppointmentProposal">
          <owlq:hasConstraint>
           <owlq:Constraint>
```

```
      <owlq:onProperty rdf:resource="&uni;at-time"/>
      <owlq:equalsVariable rdf:resource="#datetime"/>
     </owlq:Constraint>
    </owlq:hasConstraint>
   </owlq:Class>
  </owlq:hasResultClass>
  <ccs:followedBy rdf:nodeID="seq2.3"/>
 </owlq:Query>
</ccs:has-subprocess>
<ccs:has-subprocess>
 <ccs:Query rdf:nodeID="seq2.3">
  <ccs:language rdf:resource="&httpget;"/>
  <ccs:has-input-variable rdf:resource="#datetime"/>
  <ccs:bind-to-variable rdf:resource="#room"/>
  <ccs:has-specification rdf:parseType="Literal">
   http://localhost/free-rooms?date=$date&amp;time=$time
  </ccs:has-specification>
  <ccs:followedBy rdf:nodeID="seq2.4"/>
 </ccs:Query>
</ccs:has-subprocess>
<ccs:has-subprocess>
 <ccs:Assertion  rdf:nodeID="seq2.4"> <!-- assertion expressed by an owlq constraint -->
  <rdf:type rdf:resource="&owlq;Constraint"/>
  <owlq:onVariable rdf:resource="#room"/>
  <owlq:onProperty rdf:resource="&uni;available-at"/>
  <owlq:hasValue rdf:resource="#datetime"/>
  <ccs:followedBy rdf:nodeID="seq2.5"/>
 </ccs:Assertion>
</ccs:has-subprocess>
<ccs:has-subprocess>
 <ccs:Alternative rdf:nodeID="seq2.5">
  <ccs:has-subprocess>
   <ccs:Test> <!-- test expressed by an owlq constraint -->
    <rdf:type rdf:resource="&owlq;Constraint"/>
    <!-- not satisfied when #room is unbound -->
    <owlq:onVariable rdf:resource="#room"/>
    <owlq:onProperty rdf:resource="&rdf;type"/>
    <owlq:hasValue rdf:resource="&uni;Room"/>
   </ccs:Test>
   <!-- then go on-->
  </ccs:has-subprocess>
  <ccs:has-subprocess>
   <ccs:Sequence>
    <ccs:has-subprocess>
     <ccs:Test rdf:nodeID="seq3.1"> <!-- test expressed by a fml Negation -->
      <rdf:type rdf:resource="&fml;Negation"/>
      <fml:negates>
       <owlq:Constraint>
        <!-- not satisfied when #room is unbound -->
        <owlq:onVariable rdf:resource="#room"/>
        <owlq:onProperty rdf:resource="&rdf;type"/>
        <owlq:hasValue rdf:resource="&uni;Room"/>
       </owlq:Constraint>
      </fml:negates>
```

```
            <ccs:followedBy rdf:nodeID="seq3.2"/>
           </ccs:Test>
          </ccs:has-subprocess>
          <ccs:has-subprocess>
           <ccs:ContinueFixpoint rdf:nodeID="seq3.2">
            <ccs:withVariable rdf:resource="#X"/> <!-- the CCS fix variable -->
           </ccs:ContinueFixpoint>
          </ccs:has-subprocess>
         </ccs:Sequence>
        </ccs:has-subprocess>
       </ccs:Alternative>
      </ccs:has-subprocess>
     </ccs:Sequence>
    </ccs:has-body>
    <ccs:followedBy rdf:nodeID="seq1.2"/>
   </ccs:Fixpoint>
  </ccs:has-subprocess>
  <ccs:has-subprocess>
   <!-- from the smtp domain metadata it is clear that this is a mars:Action -->
   <rdf-cl:NodeSpec rdf:nodeID="seq1.2">
    <rdf-cl:ofClass rdf:resource="&smtp;Send-Mail"/>
    <rdf-cl:with-property>
     <rdf-cl:PropertySpec>
      <rdf-cl:property rdf:resource="&smtp;to"/>
      <rdf-cl:value-of rdf:resource="#lecturer-email"/>
     </rdf-cl:PropertySpec>
    </rdf-cl:with-property>
    <rdf-cl:with-property>
     <rdf-cl:PropertySpec>
      <rdf-cl:property rdf:resource="&smtp;to"/>
      <rdf-cl:value-of rdf:resource="#student-email"/>
     </rdf-cl:PropertySpec>
    </rdf-cl:with-property>
    <rdf-cl:with-property>
     <rdf-cl:PropertySpec>
      <rdf-cl:property rdf:resource="&smtp;text"/>
      <rdf-cl:value>exam with {#student}, {#subject} at
        {#datetime} in room {#room}</rdf-cl:value>
     </rdf-cl:PropertySpec>
    </rdf-cl:with-property>
   </rdf-cl:NodeSpec>
  </ccs:has-subprocess>
 </ccs:Sequence>
</eca:has-action>
</eca:Rule>
</rdf:RDF>
```

The following small files illustrates how to glue notions of different languages together:

```
<!DOCTYPE rdf:RDF [
  <!ENTITY eca "http://www.semwebtech.org/languages/2006/eca-ml#">
  <!ENTITY snoop "http://www.semwebtech.org/languages/2006/snoopy#">
  <!ENTITY ccs "http://www.semwebtech.org/languages/2006/ccs#">
```

```
  <!ENTITY owlq "http://www.semwebtech.org/languages/2006/owlq#"> ]>
<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#">
 <rdf:Description rdf:about="&snoop;Variable">
  <owl:equivalentClass rdf:resource="&eca;Variable"/>
 </rdf:Description>
 <rdf:Description rdf:about="&ccs;Variable">
  <owl:equivalentClass rdf:resource="&eca;Variable"/>
 </rdf:Description>
 <rdf:Description rdf:about="&ccs;EventSpec">
  <owl:equivalentClass rdf:resource="&eca;EventSpec"/>
 </rdf:Description>
 <rdf:Description rdf:about="&ccs;Test">
  <owl:equivalentClass rdf:resource="&eca;Test"/>
 </rdf:Description>
 <rdf:Description rdf:about="&ccs;Query">
  <owl:equivalentClass rdf:resource="&eca;Query"/>
 </rdf:Description>
 <rdf:Description rdf:about="&owlq;Variable">
  <owl:equivalentClass rdf:resource="&eca;Variable"/>
 </rdf:Description>
</rdf:RDF>
```

The following query shows that all rule-wide variables are identified as eca:Variables from the rdfs:range predicates. For the analysis of positive/negative variables, see Section 15.2.2.

```
# call
# jena -q -pellet -il RDF/XML -qf rule-query-4.sparql

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 mars: <http://www.semwebtech.org/mars/2006/mars#>
PREFIX eca: <http://www.semwebtech.org/languages/2006/eca-ml#>
PREFIX snoop: <http://www.semwebtech.org/languages/2006/snoopy#>
PREFIX ccs: <http://www.semwebtech.org/languages/2006/ccs#>
PREFIX owlq: <http://www.semwebtech.org/languages/2006/owlq#>
PREFIX uni: <http://www.semwebtech.org/domains/2006/uni#>

SELECT ?X ?Y ?Z
FROM <file:eca-ml.rdf>
FROM <file:snoopy.rdf>
FROM <file:ccs.rdf>
FROM <file:owlq.rdf>
FROM <file:rdf-cl.rdf>
FROM <file:example-rule-exam.rdf>
FROM <file:example-glue.rdf>
WHERE {{?X a owlq:Variable} UNION
       {?Y ccs:followedBy ?Z}}
```

URIs of ECA Rule Instances and Components. With the above point of view, it is obvious that each rule, and also each of its components and subexpressions is an RDF object that has a corresponding RDF URI.

Rule components, e.g., process specifications or composite event specifications can also be given in standalone RDF sources, and can then be used by other specifications or rules.

These URIs are e.g. used as identifiers when registering rules or event components, and also when communicating answers to such requests.

A note on Variable Bindings. Note that there is no RDF/OWL ontology for variable bindings. The variable bindings, answers etc stuff is purely for data exchange, which is not appropriate to be modeled and transferred in RDF format. Here, we stay with pure XML.

15.2 Reasoning about Rules

Having rules as RDF graphs allows for several kinds of reasoning about the rules. As the reference languages themselves are not yet stable, the reasoning section is still subject to change according to evaluation of the component languages.

15.2.1 Validity

The glue ontologies define how subontologies may be composed , mainly by associating classes of "higher" and "lower" ontologies (wrt. the tree representation of the rules): If a composition is not allowed, the OWL reasoner complains (e.g., when combining an action as a subexpression of an event expression). On the other hand, especially with this definition, one must be careful in the ontology design: CCS process expressions may e.g. *contain* events and queries.

Additionally, reasoning about the use of variables is a primary goal whose results can immediately be applied for supporting the evaluation of rules.

15.2.2 Safety and Evaluation

Reasoning about the use of variable and the related notions of *safety* uses the ontology for variables that has already been introduced in Section 11.2.3. The usage of variables can be added as (derived) labeled edges in the RDF graph of a rule:

- usage of variables is defined by structural induction over expressions of a language,
- expressions are represented by the RDF representation of the rules and subexpressions,
- every sublanguage must provide the rules for the inductive definition for atomic expressions and for composite expressions:

  $$\textit{uses-relationship}_1\textit{(expression,var)} \text{ :-}$$
  $$\textit{structural-relationship}_2\textit{(expr,subexpression)}, \quad \textit{uses-relationship}_2\textit{(subexpression,var)}.$$

  Note that "atomic" expressions on a higher level may be composite expressions on a lower level. As they are then members of two classes, the variable properties are propagated upwards.
- as the rule structure is a tree (or at least a directed graph, in case that subexpressions occur multiply), stratified negation and closed predicates (wrt. usage of variables) is allowed in these definitions.

For the actual implementation, there are currently several alternatives:

- use Pellet with SWRL (prototype SVN branch),

- encode the expressions and the rules in F-Logic and use Florid,
- combine Jena with a Florid-based Service for F-Logic Rules.

The third alternative is anticipated, based on [38]. The resulting information allows to analyze *variable dependencies* between fragments. Then, queries can be regarded as *sets* of subqueries, and any safe evaluation strategy can be applied. Furthermore, subqueries that only need a subset of the variables can sometimes be evaluated earlier, already during the incremental event detection process.

Chapter 16

Further Work

The MARS framework and the prototype are extended continuously.

Ongoing Work:

- stabilization of the RDF level,
- RDF-graph-based analysis of the usage of variables,
- an ontology for general queries and formulas.

16.1 Cross-Integration of Component Languages

In the above architecture, every component uses a specific kind of languages that come with a certain semantics. The semantics does not only allow the combination in form of ECA rules, but also allows for certain combinations inside components:

- events: input: variable bindings, result: answer and variable bindings if something happened,
- queries: input: variable bindings, result: answer and variable bindings according to the current state of information sources in the Web,
- test: input: variable bindings, result: constrained set of variable bindings,
- action: input: variable bindings, executes something.

The above "integration" of languages shows another nice feature: since events and queries are identified clearly in the ontology, they can also be mixed as long as the combination makes sense:

- add conditions and queries already to the event component (apart from equality via join variables, also conjunction between atomic events and immediate testing of a condition is possible),
- add conditions and even events to the action part to describe complex processes (see Section 5.8).

The respective language extensions (e.g., SNOOP + queries + tests, queries + tests (cf. Section 12.2) and CCS + events + queries + tests) must only support the respective ECA notions (nesting, opaque, variable, uses-variable etc.).

16.2 Reasoning about Rules

16.2.1 Syntactical Validation of Rules and Expressions

Every language should provide functionality to validate an expression of "its" language. Validation of leaf expressions is forwarded to the corresponding service. Note that not each service must provide this functionality, but only one service per language (finding an appropriate service is done by the LSR+GRH). This can e.g. associated with the "owner" of the namespace where also the RDFS metadata is located.

16.2.2 Safety of Rules, Goal Reordering

The RDF graph of a rule can be used for deriving the positive and negative use of variables instead of giving this information explicitly by the rule designer. From this, the subqueries can be reordered and even parallelized depending on the use of variables. The prototype on the RDF level provides already some of this functionality (cf. Section 12.2).

16.2.3 Typing

The handling of variables as resources and with references allows to do type reasoning about the variables (e.g., in OWLQ based on the owlq:scopesVariable statements and the owlq:VariableDefinitions). This can serve for optimization, validation, and also the schema of the generated structures (e.g., wenn creating RDF graphs that are added to a data source) can be derived.

16.2.4 Compositionality

The MARS Framework is actually a comprehensive compositional approach that allows to compose specific languages for events, processes, queries, etc. to rules. Having an ontology-level description of *semantics* would allow for inter-language, probably distributed analysis of the semantics of rules. Such a "semantic semantics" does not yet exist, but considering that "generic" notions like denotational semantics and operational semantics, and translations from many formalisms to model checking exist, show a promising research direction.

16.2.5 Inter-Rule Reasoning

Reasoning about rule interferences is also possible, e.g., checking which rules will/cannot be triggered by the actions of a certain rule. If a rule is triggered by another one, often even the variable bindings of the triggered rule can be expressed in terms of the variable bindings of the triggering rule.

This can be used for more comprehensive correctness/assertion proofs, and for termination analysis.

16.3 Pre-Registration and Evaluation of Components.

The downwards communication by default consists of (i) the component markup fragment and (ii) the actual variable bindings. It is in general not necessary to communicate the fragment each time the rule is evaluated. With the first evaluation, or already with the registration of a rule, the code can be submitted with an ID (e.g., its URI). Later this reference is then used. Upon de-registration of a rule, the registered components are also deleted.

- register a component with a service,
- submit a task by referencing a registered component (events: deactivate/activate, queries: evaluate, actions: execute)

- response: complain if a reference is not available (component must be sent again),
- de-register a registered component.

This is especially relevant on the RDF level when everything is considered as an RDF resource and identified by a URI (then, e.g. it is not necessary to send rules and rule components – they and their components and subexpressions can always be referenced by the URIs).

Pre-registration also allows for optimized evaluation such as inter-rule optimization and view materialization.

16.4 Evaluation and Optimization

In Section 3, we have given a Normal Form for expressing ECA rules in the Semantic Web in the general case. Evaluation, optimiziations, and special cases can deviate from this normal form as long as the original semantics is preserved.

16.4.1 Clustered Evaluation of Rule Instances

As discussed in Section 4.4.1, there may be cases where several instances of the same event are detected simultaneously. in this case, the event detection component is allowed either to return multiple singleton answers, or a set containing multiple answers. In the latter case, the "correct" semantics would be that the ECA engine immediately splits the result set and fires one instance of the rule for each occurrence. Nevertheless, the declarative semantics allows to handle all instances in parallel, only splitting them for transactional reasons when entering the action component.
When considering to handle the query "for a set a time" or "for one instance a time",

- In case that multiple event occurrences can be handled in parallel, it is more efficient to evaluate the query part once, using free variables in all positions (or binding possible values in an XQuery let style for restricting the result set), and afterwards *join* the result with the original tuples and then splitting the execution.
- Sometimes, it is more efficient (or even the only possible way) to evaluate the query component individually for each instance of the rule, e.g. when the "query" is actually a functional call that requires an input parameter (e.g., obtaining the rates for some hotels from a Web Service). Nevertheless, instead of splitting the complete rule, here it would be sufficient to evaluate the query for each tuple, and put the results together.

16.4.2 Iterator-based vs. Materialized Answers

While the event component always returns a single answer that initializes the execution of a rule, queries in general return multiple answers. Depending on the implementation, these are returned all-at-a-time, or as an iterator that enumerates one answer after the other. In the latter case, the first answers can already be returned before all answers are computed, allowing for pipelining with later components. Such properties of languages (or, more exactly, individual implementations) have to be described in the metadata. In case that an answer is given as an iterator, subsequent steps can already be executed.

The current implementation of the ECA engine already supports asynchronous and partial query answers. Whenever an answer is received, the respective tuples are split off and their evaluation continues.

16.4.3 Inter-Rule Optimization

The straightforward strategy evaluates the query part for each firing of a rule. In case events contribute to the firing of several rules, or that queries are used in different rules (recall that

queries and also subqueries are resources that have an identifier and can be reused), synergies between evaluation of queries for different rules and instances can be exploited.

Here, strategies investigated in the areas of *query rewriting* and *query containment*, *answer caching*, *continuous query services* etc. can be applied.

Inter-rule optimization is expected to be especially promising in the service-oriented architecture, where related tasks are concentrated on dedicated services. The rule execution, event detection, and also event and information brokering services can also make use of the above strategies.

16.5 Privacy and Authentification Issues

Throughout the whole report, we do not consider privacy and authentification issues, such as access control for queries and "events visible only to ..." etc. Thus, we assume that everything in the Semantic Web is open to all users. Adding authentification etc. is orthogonal to the language and architecture issues and can be solved in different ways:

- Applications that want to have privacy (e.g., an exam administration system at a university that operates based on ECA rules) duplicate parts of the structure. Inside a "known world", only one language of each type is sufficient, and also the brokering can be replaced by direct communication. If events from the outside and queries to the outside are relevant, this can be done by an appropriate interface and subscriptions to an external event broker.
- Extending the concepts with ownership data. Rules have an owner (that is obviously inherited to their components), and execution of rules and visibility of events can be restricted to certain sets of users. Then, an ontology of user groups is required, and appropriate information about users and grants has to be accessible. Queries and actions are stated under the rights of the owner/user of the rule. Event detection has to match the rights of the owner/user with the allowances for the event. Note that in RDF, each rule can easily be associated with an owner, and the same holds for events.

16.6 Transactions

The issue of transactions does not directly have to do with the semantics of ECA rules. 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.

Chapter 17

Related Work

17.1 Relationship with Existing Languages

There are several proposals for "Reactive or ECA Languages for the Web". Most of these, e.g. see [6] for a discussion, are very restricted trigger-like languages that operate on a local XML or RDF database. They are (i) covered by our approach (at least as opaque rules, but can in general also be marked up explicitly in it), and (ii) used in our approach on the lowest level for catching data-oriented events. The existing proposals can be grouped into three classes, corresponding to the discussion in Sections 2.1.2 and 7.2.1:

- Local triggers where E, C, and A component use only the local database (like for SQL triggers) – these are not languages for the *Web*, but only for XML or RDF *data*,
- "Web-Level triggers" whose E-component is a data-level event in the local database, the condition component uses the local database and possibly also remote ones, and the action part can include arbitrary actions on the Web level (sending messages, SOAP),
- ECA rules where the event component uses any events that are known at the node, and the condition and action component are arbitrary.

17.1.1 Triggers on XML Data

The existing proposals for XML data inherit much from SQL database triggers. Especially, all of them react only on atomic database update events.

"Active Rules for XML", [19].

The proposal for "Active Rules for XML" [19], transfers the "trigger" idea to XML. In contrast to similar proposals that use the common plain ON ... WHEN ...DO ... syntax, this proposal is based on an XML markup. The E, C, and A sublanguages are fixed: The event component allows for atomic events in the local XML database (monitoring nodes specified by XPath expressions) that provide the `$old` and `$new` values of the updated data items. The condition component then works on this information and on data from the local database, using XQuery. In the action component, SOAP methods can be invoked.

It is worth noting that the design of the condition and action component deviates from the SQL style, in the same direction as the query-test-action combination in the MARS Framework: since the action component's SOAP invocation acts remote, the introduction of a separate local query component was necessary that has been done here by extending the condition component: The condition component consists of an XQuery query that binds variables in a FOR clause, tests in the WHERE, and then (for each successful binding) "jumps" to the action component that invokes a SOAP method using the variable bindings of the condition component.

Active X-Query.

The proposal of Active XQuery [18] proposes triggers for XQuery in the style of SQL triggers:

```
CREATE TRIGGER name
[WITH PRIORITY number]
(BEFORE|AFTER) (INSERT|DELETE|REPLACE|RENAME)+ OF XPathExpression+
[FOR EACH (NODE|STATEMENT)]
[XQuery-LET-clause]
[WHEN XQuery-WHERE-clause]
DO (XQuery-UpdateOP|ExternalOp)
```

The trigger is associated with an XML resource and reacts after or before changes in the XML data, with granularities as for SQL. In the same way as in the MARS Framework, a query component that binds variables (LET clause to bind variables) has been introduced before the evaluation of the condition. The actions are XQuery-update operations, or external actions.

E-C-A for XML [8].

The proposal for an "ECA Language for XML" [8] transfers the "trigger" idea to XML, using the common

ON *event* IF *condition* DO *actions*

syntax. The E, C, and A sublanguages are fixed: The event component is restricted to insertions or deletions of nodes nodes in the local XML database (monitoring nodes specified by simple XPath expressions); changes are bound to a variable `$delta`. The condition and action components are evaluated separately for each instantiation of `$delta`. The condition allows *simple* XPath expressions (that return true/false – empty/nonempty) connected by boolean operators and cannot bind extra variables. The action component is a sequence of updates of local XML data (insertion/deletion of XML nodes).

Here, the condition component extends the SQL only slightly: the condition is not restricted to the information obtained in the events, but can use *simple* path expressions to use additional information from the local database in a restricted way. The action component is weaker (but cleaner) than in SQL since it does not allow "program fragments", but only updates.

This proposal is the most restricted one that is discussed in the comparison since it is completely restricted to the local database.

17.1.2 Triggers on RDF Data

The trigger concept from [8] has been extended to RDF data in [60]. The E, C, and A sublanguages are again fixed: events are inserting or deleting an instance of a class or a triple, or updating a triple; syntax; changes are bound to a variable `$delta`. The condition is given by RDFTL's path expressions (path expressions on RDF graphs, with filters) connected by boolean operators and cannot bind extra variables. The action component is a sequence of local updates of RDF data where updates can be inserting or deleting of a resource, or inserting, deleting or updating an arc. This proposal is also completely restricted to the local database.

17.1.3 ECA Rules on XML

In the XChange language [23], the E, C, and A parts are also fixed, using languages that are closely related to Xcerpt [24]. Events are represented as XML instances, that can be communicated and queried (XChange event messages). Composite events are supported, using "event queries" (Xcerpt query language extended by operators similar to an event algebra). The conditions are Xcerpt queries that collect variable bindings. Actions are complex updates of Web data (XML or RDF), using an update extension of Xcerpt. The action component can be executed as transaction. The language uses logical variables to communicate values between the components.

17.1.4 ECA Rules in XML

RuleCore [14] is a modular system for executing active rules. The ECA rules are marked up in XML (rCML – ruleCore Markup Language) and provide a clean dictinstion between event, conditon, and action part. The focus of ruleCore are the ECA engine itself, and its event detection component; the query and action component are relatively simple, but can be replaced by more sophisticated ones. The detection component is also extensible since new operators can be added by via appropriate classes. In this aspect, ruleCore provides already a simple form of the MARS Framework.

17.1.5 Coverage of the MARS Framework

Trigger-Style languages. The trigger-style languages discussed above could find applications as low-level rules located inside the databases

- inside the local database, and
- raising higher-level events based on XML update events.

They can be embedded into the MARS Framework either as opaque rules (which is reasonable for rules that are covered inside the database), or by defining a mapping from ECA-ML to their native languages. Since they all use primitive events, standard query languages and standard update concepts or SOAP, the embedding into the component markup is straightforward.

Xcerpt and XChange. The XChange/Xcerpt ECA language consists of separate parts that provide reasonable own semantics and are based on communicating variable bindings:

- event queries, closely related to event algebras (and with the same functionality; returning a variable bindings and a sequence of events that materialized a given pattern,
- a query and test language (Xcerpt), returning variable bindings,
- combinators for composite actions.

Thus, every XChange rule can be mapped easily to the ECA-ML markup using opaque components. Furthermore, given an XML markup of the above sublanguages, rules can be marked up in XML completely.

RuleCore. As already mentioned above, with its modular design, ruleCore provides already a simple form of the MARS Framework. On the other hand, the event detection component with the current event algebra can also be employed in an event detection node in the MARS Framework.

17.2 REWERSE/I5/r3

From the *General Framework for ECA Rules*, presented in REWERSE Deliverables I5-D4 and D5, [1, 2], two prototypes have been developed: MARS (*Modular Active Rules in the Semantic Web* at Georg-August-Universit"at Göttingen/Germany ; earlier versions have been described in REWERSE Deliverables I5-D6 and D9 [4, 3]), and r^3 (*Resourceful Reactive Rules* at Universidade Nova de Lisboa/Portugal). r^3 is described in more detail in [62].

Comparison of MARS with REWERSE/I5/r3. The ontology of r^3 is on a lower level, namely an ontology of software constructs that form the base for a library to implement actual languages for rules and their components.
The main functional correspondence is as follows:

- The r^3 functors of expressions correspond to the types of service tasks decribed in the MARS LSR: when using the r^3 library for implementing a component language, the respective functors/tasks have to be implemented. In contrast, the MARS infrastructure uses the LSR+GHR to submit the language fragment to a *separate, remote, autonomous* (existing) service that implements the task.
- By that, r^3 provides a generic *library* for *programming* components, while MARS provides a generic, open *framework* for cooperative work between Web Services.

Both prototypes can communicate, and use the languages provided by the each other:

- MARS using r^3: the LSR contains an entry for an r^3 node that serves a *central* entry portal for a network of r^3 nodes for certain language. Any language supported by that network, e.g., XChange for event detection, can be embedded as r^3-XChange fragment as shown below (cf. test rules at [47]). The r^3-based implementation of the functors then provides the appropriate tasks.

  The embedding is possible both on the XML level, and on the RDF level.

```
<eca:Rule xmlns:eca="http://www.semwebtech.org/languages/2006/eca-ml#">
 <eca:has-event> <!-- only on RDF level -->
  <eca:Event>
   <owl:sameAs>  <!-- only on RDF level -->
    <Expression xmlns="http://rewerse.net/I5/NS/2006/r3#"
       xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
       xml:base="http://rewerse.net/I5/NS/r3/2005/eval/xchange">
      <with>
        <Variable><name>To</name></Variable>
      </with>
      <with>
        <Variable><name>Person</name></Variable>
      </with>
      <is rdf:resource="#on" />
      <having>
       <Parameter>
        <is rdf:resource="#query" />
        <literal rdf:parseType="Literal">
           "xchange":event {{
              Booking {{ to {var To}, person {var Person}  }}
            }}
        </literal>
       </Parameter>
      </having>
    </Expression>
   </owl:sameAs>   <!-- only on RDF level -->
  </eca:Event>
 </eca:has-event>  <!-- only on RDF level -->
   :
</eca:Rule>
```

  An advanced MARS-r^3 wrapper (belonging more to the MARS side than to the r^3 side) could be written that does not require to have the r^3 elements explicit in the rule, but adds them on-the-fly.

- r^3 using MARS: a generic r^{3}toMars wrapper node has been implemented (belonging more to the r^3 side than to the MARS side) that uses an instance of the MARS *Generic Request*

Handler (GRH) class (cf. Section 9.5.4) that in turn uses the *Language and Services Registry (LSR)* on the Web to address the *individual* services directly (here, no central MARS portal is needed). Thus, any language that is available via the LSR can be embedded by just using the MARS infrastructure.

```
 <Evaluate rdf:about="dummy:evalquery2"
    xmlns="http://rewerse.net/I5/NS/2006/r3#"
    xml:base="http://rewerse.net/I5/NS/2006/r3/eval/r3tomars"
    xmlns:eca="http://www.semwebtech.org/languages/2006/eca-ml#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <solve>
      <Expression>
        <is rdf:resource="#evaluate-query"/>
        <having>
          <Parameter>
            <is rdf:resource="#language-fragment"/>
            <literal rdf:parseType="Literal">
<eca:Opaque uri="http://www.semwebtech.org/util/local/exist-wrapper/direct">
<eca:has-input-variable name="To"/>
<![CDATA[
    <response>
      <subject>...</subject>
        <log:answers xmlns:log="http://www.semwebtech.org/languages/2006/logic#">
        {
          let $root := doc('/db/travel/car-rental.xml')/car-rental
          for $car in $root/cities/city[@name=$To]/car
          return
          <log:answer>
            <log:variable-bindings>
              <log:tuple>
                <log:variable name="Class"> {$root/cars/car[@name=$car/@name]/string(@class) }
                <log:variable name="AvailableCar"> {$car/string(@name) } </log:variable>
                <log:variable name="Price"> {$car/string(@price) } </log:variable>
                <log:variable name="To"> {$To} </log:variable>
              </log:tuple>
            </log:variable-bindings>
          </log:answer>
        }
        </log:answers>
    </response>
]]>
</eca:Opaque>
              </literal>
            </Parameter>
          </having>
<with><BoundVariable><name>To</name></BoundVariable></with>
<with><Variable><name>Class</name></Variable></with>
<with><Variable><name>AvailableCar</name></Variable></with>
<with><Variable><name>Price</name></Variable></with>
      </Expression>
    </solve>
    <using>
      <Substitution>
        <binding>
```

```
          <Variable>
            <name>To</name>
            <literal rdf:parseType="Literal">Paris</literal>
          </Variable>
        </binding>
      </Substitution>
    </using>
<issuer><Client/></issuer>
   </Evaluate>
```

Appendix A

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 Engine (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

Appendix B

MARS Infrastructure URIs and Filenames

B.1 RDF Files

MARS Ontology Files.

- Ontology of domains: `mars-domain-notions.rdf`
  Actual Location:
  `http://www.semwebtech.org/mars/2006/mars-domain-notions.rdf`
- Ontology of languages and service types: `mars-language-and-service-types.rdf`
  Actual Location:
  `http://www.semwebtech.org/mars/2006/mars-language-and-service-types.rdf`
- Ontology of service characteristics: `services-ontology.rdf`
  Actual Location:
  `http://www.semwebtech.org/mars/2006/services-ontology.rdf`
- Ontology of expression structure: `expr.rdf`
  Actual Location:
  `http://www.semwebtech.org/mars/2006/expr.rdf`

The basic LSR data in RDF.

- The LSR RDF file `lsr.rdf`
  Actual Location:
  `http://www.semwebtech.org/mars/2006/lsr.rdf`

Sample and Reference Languages.

- ECA-ML ontology: `eca-ml.rdf`
  Actual Location:
  `http://www.semwebtech.org/languages/2006/eca-ml.rdf`
- Snoopy Ontology: `snoopy.rdf`
  Actual Location:
  `http://www.semwebtech.org/languages/2006/snoopy.rdf`
- CCS Ontology: `ccs.rdf`
  Actual Location:
  `http://www.semwebtech.org/languages/2006/ccs.rdf`

- OWLQ Ontology: `owlq.rdf`
  Actual Location:
  `http://www.semwebtech.org/languages/2006/owlq.rdf`
- RDF-CL Ontology: `rdf-cl.rdf`
  Actual Location:
  `http://www.semwebtech.org/languages/2006/rdf-cl.rdf`
- Travel Domain: `travel.rdf`
  Actual Location:
  `http://www.semwebtech.org/domains/2006/travel.rdf`

  `orafly.rdf`
  Actual Location: `http://www.semwebtech.org/nodes/2006/orafly/orafly.rdf`

B.2 URI Schemata

Language Types.
Used in Files `files/mars-language-and-service-types.rdf`, and `lsr.rdf`:

```
http://www.semwebtech.org/mars/2006/mars#RuleLanguage
```

Service Types and Tasks.
Used in Files `services-ontology.rdf` and `lsr.rdf`:

```
http://www.semwebtech.org/mars/2006/mars#ECAService
http://www.semwebtech.org/mars/2006/eca-service#register-rule
```

Languages.
In general, languages can have any URI (which the author assigns to them). The MARS reference languages are identified as follows (used in File `lsr.rdf`):

```
http://www.semwebtech.org/languages/2006/eca-ml#
http://www.semwebtech.org/languages/2006/snoopy#
```

Services.
In general, services can have any URI (which the author assigns to them). The services that implement the MARS reference languages are identified as follows (used in File `lsr.rdf`):

```
http://www.semwebtech.org/services/2007/snoopy
```

Prototype/Online Interface.

```
http://www.semwebtech.org/mars/frontend
http://www.semwebtech.org/mars/log
```

Appendix C

Language DTDs

C.1 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-->
```

C.2 Snoop DTDs

As described in Section 11.3, every language has a DTD for the actual Markup, and up to two auxiliary DTDs. One is a DTD for the intermediate striped RDF/XML markup that connects the XML markup with the complete set of notions defined by the ontology (cf. Section 14.1.1), and one serves for obtaining a stripped markup from the striped one; preferably the result from stripping is valid wrt. the normative language DTD.

C.2.1 The SNOOP Striped DTD

```
<!ENTITY % operand "And | Or | Sequence | Not |
   Any | MultiOccurrences |
   Aperiodic | CumulativeAperiodic | Periodic | CumulativePeriodic |
   Atomic | Opaque | ANY-other-language?">
<!ENTITY % var-decl "uses-variable | bind-to-variable |
    has-input-variable | has-output-variable |
    has-free-variable | has-bound-variable | has-positive-variable |
    has-negative-variable | scopes-variable | initialize-variable">
<!ENTITY % var-declS "bind-to-variable*, uses-variable*,
    has-input-variable*, has-output-variable*,
    has-free-variable*, has-bound-variable*, has-positive-variable*,
    has-negative-variable*">

<!ELEMENT And (%var-declS;, has-subevent+)>
<!ELEMENT Or (%var-declS;, has-subevent+)>
<!ELEMENT Sequence (%var-declS;, first, second)>
<!ELEMENT Any (%var-declS;, has-subevent+, number-of-occurrences)>
```

```
<!ELEMENT MultiOccurrences (%var-declS;, has-subevent, number-of-occurrences)>
<!ELEMENT Aperiodic (%var-declS;, has-start, fired-by, has-end)>
<!ELEMENT CumulativeAperiodic (%var-declS;, has-start, cumulate, has-end)>
<!ELEMENT Periodic (%var-declS;, has-start, fired-by, has-end)>
<!ELEMENT CumulativePeriodic (%var-declS;, has-start, cumulate, has-end)>
<!ELEMENT Not (%var-declS;, has-start, exit-not, has-end)>
<!ELEMENT ANY-other-language ANY>

<!ATTLIST And rdf:about CDATA #IMPLIED>
<!ATTLIST Or rdf:about CDATA #IMPLIED>
<!ATTLIST Sequence rdf:about CDATA #IMPLIED>
<!ATTLIST Any rdf:about CDATA #IMPLIED>
<!ATTLIST MultiOccurrences rdf:about CDATA #IMPLIED>
<!ATTLIST Aperiodic rdf:about CDATA #IMPLIED>
<!ATTLIST CumulativeAperiodic rdf:about CDATA #IMPLIED>
<!ATTLIST Periodic rdf:about CDATA #IMPLIED>
<!ATTLIST CumulativePeriodic rdf:about CDATA #IMPLIED>
<!ATTLIST Not rdf:about CDATA #IMPLIED>

<!ELEMENT has-start (%operand;)>
<!ELEMENT has-end (%operand;)>
<!ELEMENT cumulate (%operand;)>
<!ELEMENT fired-by (%operand;)>
<!ELEMENT exit-not (%operand;)>
<!ELEMENT has-subevent (%operand;)>
<!ELEMENT first (%operand;)>
<!ELEMENT second (%operand;)>
<!ELEMENT number-of-occurrences (#PCDATA)>

<!-- variable usage -->
<!ELEMENT bind-to-variable EMPTY>
   <!ATTLIST bind-to-variable rdf:resource CDATA #REQUIRED >
<!ELEMENT uses-variable EMPTY>
   <!ATTLIST uses-variable rdf:resource CDATA #REQUIRED >
<!ELEMENT has-input-variable EMPTY>
   <!ATTLIST has-input-variable rdf:resource CDATA #REQUIRED >
<!-- less important (implicit) variable usage -->
<!ELEMENT has-output-variable EMPTY>
   <!ATTLIST has-output-variable rdf:resource CDATA #REQUIRED >
<!ELEMENT has-free-variable EMPTY>
   <!ATTLIST has-free-variable rdf:resource CDATA #REQUIRED >
<!ELEMENT has-bound-variable EMPTY>
   <!ATTLIST has-bound-variable rdf:resource CDATA #REQUIRED >
<!ELEMENT has-positive-variable EMPTY>
   <!ATTLIST has-positive-variable rdf:resource CDATA #REQUIRED >
<!ELEMENT has-negative-variable EMPTY>
   <!ATTLIST has-negative-variable rdf:resource CDATA #REQUIRED >

<!ELEMENT Opaque (#PCDATA | %var-decl; | ANY-other-language)* >
<!ATTLIST Opaque language CDATA #IMPLIED
                 uri CDATA #IMPLIED
                 method CDATA #IMPLIED
                 domain CDATA #IMPLIED
                 content-type CDATA #IMPLIED >
```

```
<!ELEMENT Atomic (%var-declS;, ANY-other-language)>
<!ATTLIST Atomic xmlns:other-namespace CDATA #IMPLIED
                 language CDATA #IMPLIED
                 domain CDATA #IMPLIED>
```

C.2.2 The SNOOP-ML DTD

```
<!ENTITY % operand "(And | Or | Sequence | Not |
   Any | MultiOccurrences |
   Aperiodic | CumulativeAperiodic | Periodic | CumulativePeriodic |
   Atomic | Opaque )">
<!ENTITY % var-decl "uses-variable | bind-to-variable |
    has-input-variable | has-output-variable |
    has-free-variable | has-bound-variable | has-positive-variable |
    has-negative-variable | scopes-variable | initialize-variable">
<!ENTITY % var-declP "(uses-variable | bind-to-variable |
    has-input-variable | has-output-variable |
    has-free-variable | has-bound-variable | has-positive-variable |
    has-negative-variable)*">

<!ELEMENT And (%var-declP;, %operand;,%operand;+)>
<!ELEMENT Or (%var-declP;, %operand;,%operand;+)>
<!ELEMENT Sequence (%var-declP;, %operand;, %operand;)>
<!ELEMENT Any (%var-declP;, %operand;+)>
<!ATTLIST Any number-of-occurrences CDATA #REQUIRED>
<!ELEMENT MultiOccurrences (%var-declP;, %operand;)>
<!ATTLIST MultiOccurrences number-of-occurrences CDATA #REQUIRED>
<!ELEMENT Aperiodic (%var-declP;, %operand;, %operand;, %operand;)>
<!ELEMENT CumulativeAperiodic (%var-declP;, %operand;, %operand;, %operand;)>
<!ELEMENT Periodic (%var-declP;, %operand;, %operand;, %operand;)>
<!ELEMENT CumulativePeriodic (%var-declP;, %operand;, %operand;, %operand;)>
<!ELEMENT Not (%var-declP;, %operand;, %operand;, %operand;)>
<!ELEMENT Atomic (%var-declP;, ANY-other-language?)>
<!ELEMENT Opaque ANY>

<!-- variable usage -->
<!ELEMENT bind-to-variable EMPTY>
   <!ATTLIST bind-to-variable name CDATA #REQUIRED >
<!ELEMENT uses-variable EMPTY>
   <!ATTLIST uses-variable name CDATA #REQUIRED >
<!ELEMENT has-input-variable EMPTY>
   <!ATTLIST has-input-variable name CDATA #REQUIRED
                                use CDATA #IMPLIED>
<!-- less important (implicit) variable usage -->
<!ELEMENT has-output-variable EMPTY>
   <!ATTLIST has-output-variable name CDATA #REQUIRED >
<!ELEMENT has-free-variable EMPTY>
   <!ATTLIST has-free-variable name CDATA #REQUIRED >
<!ELEMENT has-bound-variable EMPTY>
   <!ATTLIST has-bound-variable name CDATA #REQUIRED >
<!ELEMENT has-positive-variable EMPTY>
   <!ATTLIST has-positive-variable name CDATA #REQUIRED >
```

```
<!ELEMENT has-negative-variable EMPTY>
   <!ATTLIST has-negative-variable name CDATA #REQUIRED >
<!ELEMENT scopes-variable EMPTY>
   <!ATTLIST scopes-variable name CDATA #REQUIRED >

<!ATTLIST Atomic xmlns:other-namespace CDATA #IMPLIED
                 language CDATA #IMPLIED
                 domain CDATA #IMPLIED>

<!ATTLIST Opaque language CDATA #REQUIRED
                 domain CDATA #REQUIRED>

<!-- variable declaration stuff as in ECA-ML -->
```

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, Erik Behrends, Michael Eckert, Oliver Fritzen, Wolfgang May, Paula Lavinia Pătrânjan, and Franz Schenk. A first prototype on evolution and behavior at the XML level. Technical Report I5-D5, REWERSE EU FP6 NoE, 2006. Available at `http://www.rewerse.net`.

[3] José Júlio Alferes, Ricardo Amador, Erik Behrends, Tiago Franco, Oliver Fritzen, Ludwig Krippahl, Wolfgang May, and Franz Schenk. Prototype on the rdf/owl level. Technical Report I5-D9, REWERSE EU FP6 NoE, 2007. Available at `http://www.rewerse.net`.

[4] José Júlio Alferes, Ricardo Amador, Erik Behrends, Oliver Fritzen, Tobias Knabke, Wolfgang May, Franz Schenk, and Daniel Schubert. Reactive rule ontology: Rdf/owl level. Technical Report I5-D6, REWERSE EU FP6 NoE, 2007. Available at `http://www.rewerse.net`.

[5] 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 LNCS, pages 101–115. Springer, 2005.

[6] José Júlio Alferes, James Bailey, Mikael Berndtsson, François Bry, Jens Dietrich, Alexander Kozlenkov, Wolfgang May, Paula-Lavinia Pătrânjan, Alexandre Pinto, Michael Schröder, and Gerd Wagner. State-of-the-art on evolution and reactivity. Technical Report I5-D1, REWERSE EU FP6 NoE, 2004. Available at `http://www.rewerse.net`.

[7] 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`.

[8] James Bailey, Alexandra Poulovassilis, and Peter T. Wood. An Event-Condition-Action Language for XML. In *Int. WWW Conference*, 2002.

[9] Erik Behrends, Oliver Fritzen, Tobias Knabke, Wolfgang May, and Franz Schenk. Rule-Based Active Domain Brokering for the Semantic Web. In *Web Reasoning and Rule Systems (RR)*, number 4524 in LNCS, pages 259–268, 2007.

[10] Erik Behrends, Oliver Fritzen, Wolfgang May, and Franz Schenk. Combining ECA Rules with Process Algebras for the Semantic Web. In *Rule Markup Languages (RuleML)*, pages 29–38. IEEE Press, 2006.

[11] Erik Behrends, Oliver Fritzen, Wolfgang May, and Franz Schenk. Embedding Event Algebras and Process Algebras in a Framework for ECA Rules for the Semantic Web. *Fundamenta Informaticae*, 82:237–263, 2008.

[12] 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 LNCS, pages 887–898. Springer, 2006.

[13] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 1(37):77–121, 1985.

[14] Mikael Berndtsson and Marco Seiriö. Design and Implementation of an ECA Rule Markup Language. In *Rule Markup Languages (RuleML)*, number 3791 in LNCS, pages 98–112. Springer, 2005.

[15] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 2001.

[16] Eike Best, Raymond Devillers, and Maciej Koutny. The Box Algebra = Petri Nets + Process Expressions. *Information and Computation*, 178:44–100, 2002.

[17] 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/`.

[18] 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.

[19] 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.

[20] A. J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133(2):205–265, 1994.

[21] Anthony J. Bonner and Michael Kifer. Transaction logic programming. In David S. Warren, editor, *Intl. Conference on Logic Programming (ICLP)*. MIT Press, 1993.

[22] François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan. Reactivity on the Web: Paradigms and applications of the language XChange. *J. of Web Engineering*, 5(1):3–24, 2006.

[23] 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.

[24] 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.

[25] 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.

[26] 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`.

[27] Description Logic Implementation Group (DIG). http://dl.kr.org/dig/.

[28] Document object model (DOM). `http://www.w3.org/DOM/`, 1998.

[29] 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`.

[30] eXcelon Corp. XML application development using excelon, 2001. http://www.exceloncorp.com/.

[31] eXist: an Open Source Native XML Database. http://exist-db.org/.

[32] Florid homepage. http://www.informatik.uni-freiburg.de/~dbis/florid/, 1998.

[33] Carsten Gottschlich. Evaluation of the Oracle 10g Rules Manager for a Domain Node Architecture in the Semantic Web ECA Framework. Master Thesis, Univ. Göttingen, 2006.

[34] D. Harel. *First-Order Dynamic Logic*. Number 68 in LNCS. Springer, 1979.

[35] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[36] R. Janicki and P. E. Lauer. *Specification and Analysis of Concurrent Systems – the COSY Approach*. EATCS Monographs on Theoretical Computer Science. Springer, 1992.

[37] Jena: A java framework for semantic web applications. http://jena.sourceforge.net.

[38] Heiko Kattenstroth. Combining Description Logic and F-Logic Reasoning. Diploma Thesis, Univ. Göttingen, 2007.

[39] Heiko Kattenstroth, Wolfgang May, and Franz Schenk. Combining OWL with F-Logic Rules and Defaults. In *International Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services (ALPSWS)*, pages 60–75, 2007. http://www.ceur-ws.org/Vol-287.

[40] Michael Kay. SAXON: an XSLT processor. http://saxon.sourceforge.net/.

[41] Jochen Kemnade. OWLQ – an OWL-based Query Language for OWL Data. Master Thesis, Univ. Göttingen, 2007.

[42] 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.

[43] 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.

[44] Tobias Knabke. Development and Implementation of a Domain Broker for the Semantic Web. Master Thesis, Univ. Göttingen, 2006.

[45] Patrick Lehti. Design and Implementation of a Data Manipulation Processor for an XML Query Language (diploma thesis), August 2001. Technische Universität Darmstadt.

[46] 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.

[47] The MARS (Modular Active Rules for the Semantic Web) Framework. http://www.semwebtech.org/mars/.

[48] 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.

[49] Wolfgang May. The LOPix system, 2001. http://dbis.informatik.uni-goettingen.de/lopix/.

[50] Wolfgang May. A rule-based querying and updating language for XML. In *Workshop on Databases and Programming Languages (DBPL 2001)*, number 2397 in LNCS, pages 165–181, 2001.

[51] 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.

[52] 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 LNCS, pages 30–44. Springer, 2005.

[53] 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 LNCS, pages 1553–1570. Springer, 2005.

[54] 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 LNCS, pages 134–148. Springer, 2006.

[55] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, pages 267–310, 1983.

[56] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[57] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 1(100):1–77, 1992.

[58] 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.

[59] 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.

[60] 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.

[61] Pellet: An OWL DL reasoner. Maryland Information and Network Dynamics Lab, `http://www.mindswap.org/2003/pellet`.

[62] REWERSE I5 r3 : Resourceful Reactive Rules. `http://di150.di.fct.unl.pt:15080/r3/`.

[63] Best Practice Recipes for Publishing RDF Vocabularies. `http://www.w3.org/TR/2006/WD-swbp-vocab-pub-20060314/`, 2006.

[64] Rule markup language (ruleml). `http://www.ruleml.org/`.

[65] Daniel Schubert. Development of a Prototypical Event-Condition-Action Engine for the Semantic Web. Bachelor Thesis, Univ. Göttingen, 2005.

[66] Software AG. Tamino – an internet database system, 2001. `http://www.tamino.com/`.

[67] Sebastian Spautz. Automatenbasierte Detektion von Composite Events gemäss SNOOP in XML-Umgebungen. Diplomarbeit, TU Clausthal (in german), 2006.

[68] 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.

[69] Heiko Vollmann. DTD-Driven Export of OWL Data to XML. Bachelor Thesis, Univ. Göttingen, 2008.

[70] Elke von Lienen. Entwicklung eines RDF-Web-Services mit Trigger-Funktionalität. Diplomarbeit, TU Clausthal (in german), 2006.

[71] Thomas Westphal. Translation of Semantic Level Actions in an RDF Web Service. Bachelor Thesis, Univ. Göttingen, 2007.

[72] XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Recommendation, `http://www.w3.org/TR/xquery-operators`, 2007.

[73] XML Syntax for XQuery 1.0 (XQueryX). `http://www.w3.org/TR/xqueryx`, 2001.