I5-D7

Completion of the prototype scenario

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/P (report and prototype)
Dissemination level:	PU (public)
Document number:	IST506779/Lisbon/I5-D7/D/PU/a1
Responsible editors:	José Júlio Alferes
Reviewers:	Wolfgang May
Contributing participants:	Göttingen, Lisbon, Munich
Contributing workpackages:	I5
Contractual date of deliverable:	28 February 2007
Actual submission date:	16 March 2007

Abstract
This report presents the refinements made in the prototypes described in the previous deliverable (deliverable I5-D5 – "A first prototype on evolution and behaviour at the XML level"), and their usage in use-case scenarios. In particular, the use-case scenarios have been chosen according to the sketch of use-cases designed in the deliverable I5-D2&3 – "Use-cases on evolution and reactivity". Besides this report, the deliverable consists also of the prototypes themselves, which are all freely available online from http://rewerse.net/I5.

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

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

© *REWERSE 2007.*

Completion of the prototype scenario

José Júlio Alferes[1], Ricardo Amador[1], Erik Behrends[2], François Bry[3], Michael Eckert[3], Tiago Franco[1], Oliver Fritzen[2], Hendrik Grallert[3], Tobias Knabke[2], Ludwig Krippahl[1], Wolfgang May[2], Paula Lavinia Pătrânjan[3], Franz Schenk[2], Daniel Schubert[2]

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

[2] Institut für Informatik, Universität Göttingen

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

16 March 2007

Abstract

This report presents the refinements made in the prototypes described in the previous deliverable (deliverable I5-D5 – "A first prototype on evolution and behaviour at the XML level"), and their usage in use-case scenarios. In particular, the use-case scenarios have been chosen according to the sketch of use-cases designed in the deliverable I5-D2&3 – "Use-cases on evolution and reactivity". Besides this report, the deliverable consists also of the prototypes themselves, which are all freely available online from http://rewerse.net/I5.

Keyword List

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

Contents

1 Introduction **1**

2 r^3-based Bio Domain Broker **3**
2.1 ECA Prototype - r^3 v0.12 . . . 4
2.1.1 Introduction . . . 4
2.1.1.1 What is r^3? . . . 5
2.1.1.2 Why Resourceful? . . . 5
2.1.2 Resourceful reactive engines . . . 6
2.1.2.1 Resourceful reactive rules . . . 7
2.1.2.2 Messages and loading reactive rules . . . 14
2.1.2.3 Evaluating reactive rules . . . 16
2.1.2.4 Dealing with language heterogeneity . . . 17
2.1.2.5 Additional details . . . 25
2.1.3 On the way to full r^3, growing up... . . . 26
2.1.3.1 From childhood (v0.0x)... . . . 26
2.1.3.2 Through adolescence (v0.1x)... . . . 27
2.1.3.3 Keeping in good company... . . . 33
2.1.3.4 Towards maturity (v0.20)... . . . 37
2.2 Use-Case Scenario . . . 37
2.2.1 Scenario overview . . . 37
2.2.2 Use-Cases overview . . . 39
2.3 Current state . . . 39
2.3.1 Available prototype, quick start . . . 39
2.3.2 Description . . . 40
2.3.3 Coverage of E&R Use-Cases . . . 42
2.3.4 Interoperability with other WGI5 Prototypes . . . 42
2.3.5 Future work . . . 42

3 XChange for the Eighteenth Century Studies Society **49**
3.1 XChange Language and Prototype . . . 49
3.1.1 An overview of XChange . . . 50
3.1.2 Queries as Patterns . . . 51
3.1.3 Event Queries . . . 51
3.1.4 Web Queries . . . 52
3.1.5 Actions: Updates and Raising New Events . . . 52

3.2 Use-Case Scenario 52

A Component Engines. Examples and implementation in r^3 63
A.1 HTTP Engine 63
A.2 Prova Engine 64
A.3 Xcerpt Engine 65
A.4 XChange Engine 66
A.5 XQuery Engine 68

Chapter 1

Introduction

In our deliverable of the first year I5-D2&3 – "Use-cases on evolution and reactivity" we have sketched three use-case scenarios for experimenting and illustrating with evolution and reactivity on the Semantic Web. Namely:

Project Portal. In this scenario, a portal for a project, just like REWERSE was imagined, for: exchanging and collecting information about the project, its participants, and its results; and for presenting and providing information about the project on the (Semantic) Web.

Travels domain. This scenario was concerned with planning travels based on information gathered in the web, and acting on the web for the organization of such travels (e.g. by buying train tickets, booking flights online, etc). The issue of reacting to happenings that influence the initial plan, and re-planning accordingly, was also taken in consideration in this scenario.

Bioinformatics data sources. In the bioinformatics domain there are many publicly accessible data sources, which are often mirrored locally and integrated with other data. The bioinformatics use case discusses four specific data sources, PubMed, a database of 12.000.000 biomedical literature abstracts, GeneOntology, an ontology for molecular biology, with 19.000 concepts, PDB, a database with some 25.000 protein structures, and SCOP, the Structure Classification of Proteins, which groups PDB structures according to their evolutionary relationships. The scenario was concerned with mirroring these data sources locally, keeping them consistent and integrating them, and provide a level of service an infra-structure by which researchers may define their own personalized mirrors of PDB.

In our deliverable of 6 month ago, I5-D5 – "A first prototype on evolution and behaviour at the XML level", we reported on the state of the first prototype implementations of the *General Framework for Evolution and Reactivity in the Semantic Web* and of the language XChange. These two sets of implementations reflect the two approaches that are investigated in this working group, both based on the paradigm of *Event-Condition-Action (ECA) Rules*: one is a *General Framework for Evolution and Reactivity in the Semantic Web* that supports ECA Rules over *heterogeneous* component languages, i.e., integrates arbitrary event formalisms, query languages and action languages. The other is the Xcerpt/XChange approach whose goal

is an *homogeneous* ECA language. Both designs have been described in depth in the previous deliverable [2].

Regarding the implementation of the General Framework, two prototypes were mentioned in the previous deliverable: one being developed in Göttingen, of a Modular Active Rules for the Semantic Web – MARS; the implementation of Resourceful Reactive Rules – $\boldsymbol{r^3}$, being mainly developed in Lisbon. The first of these two prototypes was more advanced at that stage, and thus was presented with many details, whilst the latter was less detailed. For this reason, the $\boldsymbol{r^3}$ prototype, that has made significant advances in this last period, is described with much more detail in this report (in Section 2.1). The prototypical implementation of XChange, developed in Munich, has been described in our previous deliverable in sufficient detail too.

With prototypes in place, and with use-case scenarios designed, the time was up to experiment the prototypes with the scenarios. This is what was done in this last period, and is partly reported in this report.

All the three scenarios were approached, each by each of the three existing prototypes[1]. The scenario of Bioinformatics was experimented in Lisbon and implemented with $\boldsymbol{r^3}$, in collaboration with colleagues from the A2 work-package also of Lisbon, and is detailed in Chapter 2. The scenario of the project portal was experimented in Munich, implemented using XChange, and is detailed in Chapter 3. Regarding the travel domain, in the context of a MSc. Thesis in Göttingen a basic architecture for a domain broker for the MARS prototype was implemented, using fragments of the travel scenario for illustration. More about the details of the MARS prototype can be found on the companion deliverable I5-D6 – "Reactive rule ontology: RDF/OWL level". To avoid repetitions we don't include it in this report.

[1] Notably, each of the main developers group of a prototype implemented a use-case proposed by another group.

Chapter 2

r^3-based Bio Domain Broker

Resourceful Reactive Rules (r^3) is a prototype of a (Semantic) Web Rule Engine for (Semantic) Reactive Rules. r^3 is implemented by the CENTRIA research centre at the New University of Lisbon.

At the heart of the r^3 prototype is the decision to fully base its implementation on an RDF model, embracing Semantic Web technologies and standards by taking an ontology/resource-based perspective on reactive rules (to the extent allowed by the actual availability of "stable" prototypes[1]). Every resource that matters to an r^3 engine (e.g. rules) is to be kept in this RDF model and described in terms of an OWL-DL foundational ontology, the r^3 ontology. The different components of each reactive rule are specified (and composed) using different component languages. Each of those languages is to be implemented by specific expression sub-engines, all of them, languages and engines, also described using the terminology provided by the r^3 ontology. For an r^3 engine, everything is a resource, even rule and rule component evaluation instances are to be represented as resources (for these an adequate extension of the r^3 ontology is being defined). Naturally, openness and flexibility (as opposed to efficiency and optimization) are the main drivers for the technical choices behind r^3.

The Bioinformatics Domain Broker (B-Domain), in its current state, provides/manages personalized mirrors of the protein databank (PDB), i.e. tertiary bioinformatics data sources (cf. the use-case scenario "*Updates and evolution in bioinformatics data sources*" described in [5]). Extension of B-Domain to other bioinformatics specific functionalities is far from excluded (e.g. a Personalization Service for the Personal Reader framework is currently under consideration).

r^3 provides a Java development library intended to help in the development of (generic or domain/application specific) language engines. This library abstracts many of the details needed to realize an engine conforming to the General ECA framework detailed in [2] (e.g. communication/protocols, RDF/XML manipulation, binding variables, joining substitution sets).

B-Domain uses the r^3 library to implement a bioinformatics specific language[2] that currently includes three atomic constructs. An atomic event that signals the addition of *new* structures to PDB (occurs every time a new structure is added and returns the PDBID of the added structure). An atomic condition that checks if a specific PDBID *satisfies* a criterion specified using domain specific concepts. An atomic action that *store*s a PDBID in a personal repository

[1] Integration with Semantic Web Services was attempted but had to be postponed. Nevertheless by using a literal SOAP body conforming to an RDF/XML serialization r^3 tries to keep an open perspective for the future.

[2] `http://di150.di.fct.unl.pt:15080/b-domain/schemas/2007/b-domain.owl`

of structures. Using this language, the maintenance of the personalized PDB mirrors is achieved by a set of reactive rules generated and loaded to an r^3 ECA engine by B-Domain, once again, using the r^3 development library.

2.1 ECA Prototype - r^3 v0.12

Resources provide the foundation for the Semantic Web. Ultimately, in the Semantic Web everything will be described in resource terms (RDF). For rules to be first class citizens in the Semantic Web, they should also be described at the RDF level. This approach allows the manipulation of rules as pure semantic objects, (re-)using other Semantic Web technologies, not only to define rules, but, more importantly, to reason about rules, leading to a truly adaptive Semantic Web.

Based on previous work, where we have proposed an ontology approach for describing reactive (ECA) rules, in this section we provide additional details about the current results of ongoing work on implementing a prototype for an ECA-based reactive rule engine (r^3) using that approach. The current prototype "talks" at the semantic (conceptual) level, relying on an open and collaborative architecture (that provides a framework for extensible reactive behavior), and as such it avoids imposing a particular (concrete) syntax for ECA rules (or, for that matter, on any of its event, query/condition and update/action sub-languages), using instead a foundational ontology which is also detailed here (at least its current state, since it is object of ongoing work).

2.1.1 Introduction

The Semantic Web architecture, since its inception [14] to current proposals [15], has always included rules as one of its upper layers. In this architecture the Rules layer is presented as based upon the Ontology layer. Despite this, existing rule engines (including Semantic Web engines like [59]) rely heavily upon concrete XML markups (e.g. [16]) or proprietary languages for communicating rules. This reality restricts the level of integration with other Semantic Web tools and technologies wrt. rules themselves. Also, this way rule languages do not build upon a truly re-usable and extensible model (as the one offered by the ontology layer).

Semantic Web rules and rule engines should go semantic all the way. Rule engines have to embrace the ontology layer, even when designing their interface, quite similarly to (or even better, applying if possible) the results achieved by the Semantic Web Services community. Furthermore, the ontology layer should also be used to describe the internal state of rule engines; allowing, for instance, to check the consistency of different representations for the same resource (e.g. different instances of the "same" rule, as "held" by different rule engines). The ultimate goal would be to realize the concept of *Reliable (Reactive) Rules*. Defining a foundational ontology and realizing *Resourceful Reactive Rules* is just a first step in a long way yet to be trailed towards that final goal.

Also, rule engines cannot aim at doing everything by themselves, they have to become really Web oriented, taking an open and collaborative perspective when evaluating different rule components (in cases where such different components exist, like the case of the reactive ECA rules considered in this work). The incompleteness concept is core to the Semantic Web and it applies not only to data but also to behavior. Rule engines must focus on providing the inference mechanisms, evaluating (different types of) rules, towards drawing conclusions

(or performing actions) based on given conditions (or event occurrences), regardless of what the specific rule components (conditions, events, conclusions or actions) really are/mean. The semantic interpretation of rule components must be an open matter, for which a main rule engine relies on other sub-engines. In most cases these "sub-engines" will be domain/application specific, but generic inference engines, rule-based or not, are not to be excluded.

2.1.1.1 What is r^3?

r^3 stands for "*Resourceful Reactive Rules*", it is a REWERSE WGI5[3] sub-project [53] being developed by CENTRIA in Lisbon, and aims at building a functional research prototype implementing the concepts originally proposed in [4], thus trying to fulfill (and at the same time evaluate and experiment) the reactive approach to evolution on the Semantic Web as stated in the goals of REWERSE WGI5 (and initially outlined in [31]).

Two major requirements were previously identified in order to realize this open/semantic perspective on reactivity and evolution for the Web: taking an ontology/resource-based perspective (as detailed in [30]), and dealing with language heterogeneity at the rule component level (as explained in [29]). These requirements led to the specification of a model [2] for a general framework that r^3 aims to bring to life.

In more concrete terms, r^3 is a prototype of a (Semantic) Web Rule Engine for (Semantic) Reactive Rules. Reactive rules have the general form "*on Event if Condition do Action*", and are also known as: ECA rules, triggers, or active rules. They are intuitively easy to understand, viz. when an event (atomic or composite) occurs, evaluate a condition, and if the condition is satisfied then execute an (atomic or complex) action. This prototype is capable of dealing with reactive rules that use different languages either at the rule component level (event, condition, action), or within each component (by algebraic composition, based also on different algebraic languages).

Component languages may range from general purpose languages (with high flexibility, but limited semantic transparency/value), to domain or application specific languages (with much richer semantic value). For instance, if a language specific to the currency trading domain exists, an event stating that "`the exchange rate between Euro and US$ has changed`" may be specified using exactly these terms; for sure in any such language, the precise meaning of this statement is well known. Otherwise, if no such domain specific language is available, a general purpose language like XChange [17] (viz. $\boldsymbol{XChange^{EQ}}$ [18]) may be used instead, namely by specifying: an XML pattern to be applied to every incoming event message originated from the addresses of the International Monetary Fund or of the Federal Reserve Bank of New York (assuming these addresses and the actual XML markups used are all known). Of course, in the latter case, one would have to ensure (manually in the rule itself) that such a low level specification is (and will be kept) consistent with the currency market common sense; whereas in the former case, the semantic consistency of the rule is intrinsic to the domain specific language used.

2.1.1.2 Why Resourceful?

r^3 fully embraces the Semantic Web by taking an ontology/resource-based perspective on reactive rules. At the heart of the r^3 prototype is the decision to fully base its implementation on

[3] `http://rewerse.net/I5/`

an RDF [54] model. Every resource that matters to an $\boldsymbol{r^3}$ engine (e.g. rules) is to be described in terms of an OWL-DL foundational ontology: the $\boldsymbol{r^3}$ ontology.

$\boldsymbol{r^3}$ sees Rules as First Class Citizens of the Semantic Web, allowing research results from other areas, related to the Semantic Web, to be applied also to rules. For instance, it becomes possible to reason about rules by defining rule ontologies and by stating rules about rules, eventually leading to rule evaluation policies and adaptive behaviour.

The different components of each reactive rule are specified (and composed) using different component languages. Each of those languages is to be implemented by specific expression sub-engines (all of them, languages and engines, also described using the terminology provided by the $\boldsymbol{r^3}$ ontology). For an $\boldsymbol{r^3}$ engine, everything is a resource, even rule and rule component evaluation instances are to be represented as resources (for these an adequate extension of the $\boldsymbol{r^3}$ ontology is being defined).

Natively $\boldsymbol{r^3}$ "talks" RDF/XML [55] (using HTTP POST for communication, SOAP [58] wrapped or not), but any other XML serialization (concrete markup) of an RDF model is acceptable, provided an appropriate (bi-directional) translator is available. Any request received by an $\boldsymbol{r^3}$ engine is expected to be translated into an RDF model based on the $\boldsymbol{r^3}$ ontology. This model is then added to an internal ontology that includes every resource known to a particular $\boldsymbol{r^3}$ engine. This internal ontology must also be dynamically completed by means of automatic searching and fetching missing pieces of information (resource representations) directly from the (Semantic) Web (already identified, in [15], as an important issue, yet to be supported by $\boldsymbol{r^3}$).

Notice that besides adding resources/models to this internal ontology (e.g. loading rules), which already introduces consistency matters, it is also possible to remove resources/models (e.g. unloading rules), which clearly involves non-monotonic matters. Also when we talk about merging ontologies that contain rules, this merging may lead to updating rules and evolving rule-bases. At the current stage of $\boldsymbol{r^3}$ these issues are (almost) ignored, but they are expected to be considered at a later stage.

If it is true that this fully resource-based (or down-to-earth, *resource-full*) approach used by $\boldsymbol{r^3}$ opens many research issues (some of them yet untouched in what concerns the Semantic Web), it is also true that every research issue opened by $\boldsymbol{r^3}$ (given its modular architecture) constitutes an opportunity for scientific cooperation with other research communities, paving the way to more *Resourceful* solutions.

2.1.2 Resourceful reactive engines

In this section, we present the current state of the foundational ontology used by $\boldsymbol{r^3}$ including the abstract API of $\boldsymbol{r^3}$ engines, trying to clarify what we mean by reactive rules and rule engines going semantic all the way. We start by describing the ontology top level that deals with the operations supported by a reactive rule engine (and consequently with the generic concepts of reactive rules and rule sets, described in section 2.1.2.1); namely operations to load (section 2.1.2.2) and evaluate (section 2.1.2.3) reactive rules. In section 2.1.2.4 we detail the lower levels of the ontology that provide the support for dealing with language heterogeneity at the rule component level. For the sake of completeness of the presented ontology, in section 2.1.2.5, additional details (some more concerned with the actual implementation of the prototype) are briefly introduced.

For the sake of readability, we have chosen here to completely define the current state of the $\boldsymbol{r^3}$ ontology using UML 2.0 [34] diagrams, in figures 2.1 to 2.8, and to illustrate it with

examples written using N3 [13] notation[4]. Nonetheless, the normative definition used by r^3 is an OWL-DL [50] ontology (`http://rewerse.net/I5/NS/r3/2005`), and currently r^3 only supports XML-based serializations (e.g. RDF/XML [55]).

The r^3 prototype is actually formed by an abstract network (physically distributed or not) of r^3 *Engine*s that cooperate towards evaluating ECA rules. The entry point into this network is an r^3 main *Engine* (or *ECAEngine*, as depicted in figure 2.1) providing a *Load* operation that allows an external *Client* to *activate* a *RulePackage*. A *RulePackage* is either a single *Rule* or a *RuleSet* (as shown in figure 2.2), where the latter recursively *contains* other *RulePackage*s.

The *ECAEngine* interfaces with r^3 *Language* specific sub-engines (called *LanguageEngine*s), e.g. sub-engines for detecting events, for querying Web data, for testing conditions, etc. One such r^3 *LanguageEngine implements* one or more *Language*s, where each *Language defines* a set of parametric *LanguageConstruct*s[5]. Each *LanguageConstruct* actually provides the semantics of any *CodingElement Construction* that *is* defined by it. *BrokerEngine*s may also be used in cases where an appropriate *LanguageEngine* is not known.

The API of r^3 *Engine*s is defined at a conceptual level based on an ontology of concepts (instantiated into RDF resources, thus being resource-full), fostering the use of other Semantic Web technologies (thus becoming resourceful). This resource-based approach is present in every operation supported by an r^3 *Engine* and consistently carried across the concepts of *RuleSet*s and *Rule*s, introducing rule *component*s that are heterogeneous *Language Expression*s. These *Expression*s are *Evaluate*d by *ExpressionEngine*s (which are *Language* specific) providing the *implementation* of the semantics of all the *ExpressionConstruct*s that a *Language defines*.

*ExpressionConstruct*s allow both the specification of atomic *Expression*s (e.g. events or actions) using either generic or domain/application specific *Language*s (that some, e.g., *EventDetector* or *ActionProcessor implements*), and also the specification of composite *Expression*s using algebraic *Operator*s, of some (e.g. event or process) algebra provided by *AlgebraEngine*s that recursively *Evaluate* the *OperatorArgument*s using other r^3 *ExpressionEngine*s.

2.1.2.1 Resourceful reactive rules

As proposed by previous work [29], *ECARule*s are formed by several distinct *component*s (further partitioned here between *antecedent*s and *consequent*s), namely: an *event component* (specifying the event occurrences that trigger the activation of the *ReactiveRule*), an optional condition part formed by several *component*s (that, based on an actual event occurrence, gather/*query* additional information and define/*test* the applicability of the rule under consideration), and an *action component* (stating the *consequent* actions to execute whenever the *ActiveRule* is applicable, given its *antecedent*s). It is worth noting that this abstract component structure could also be used to accommodate other kinds of rules (as depicted in figure 2.3).

The rule *component*s that form an *AbstractRule* are *Expression*s (as depicted in figure 2.2) defined using some *Language* (as detailed later in section 2.1.2.4), thus allowing the use of different *Language*s for different *component*s. The communication between these heterogeneous *component*s is achieved through the use of logical variables, bound to XML literals[6].

[4]For the sake of simplicity, the included N3 examples omit most of the `@prefix` declarations, and they all assume at least the declaration of `@prefix : <http://rewerse.net/I5/NS/r3/2005#>.`

[5]Notice that the UML diagram, in figure 2.1, does not prevent an *ECAEngine* from being at the same time a *LanguageEngine*.

[6]Currently these literals are opaque to an r^3 main engine, but nevertheless RDF references, serializations, or typed data values are not excluded.

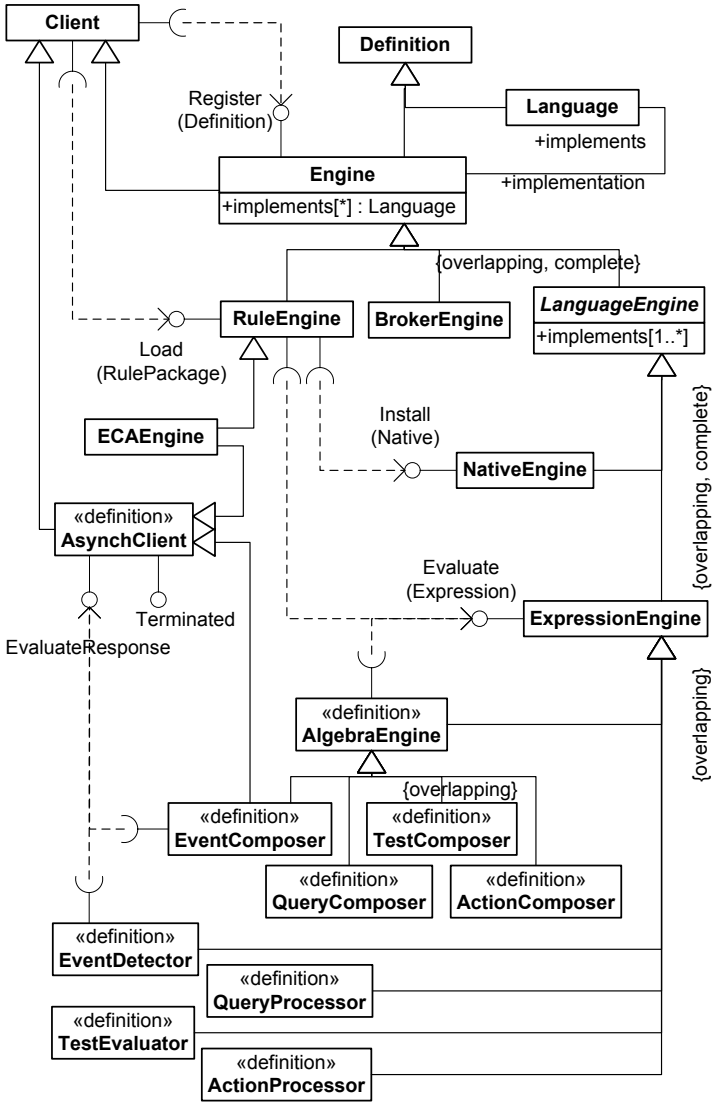

Figure 2.1: Engines (v0.10)

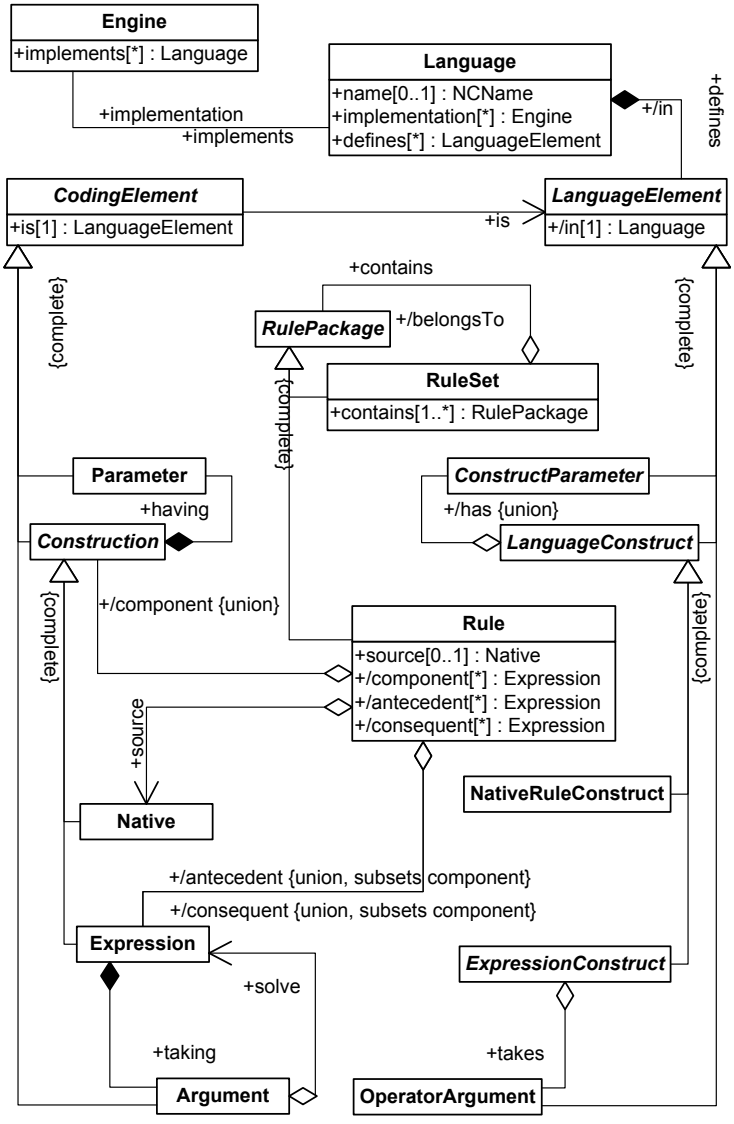

Figure 2.2: Rule Sets (v0.12)

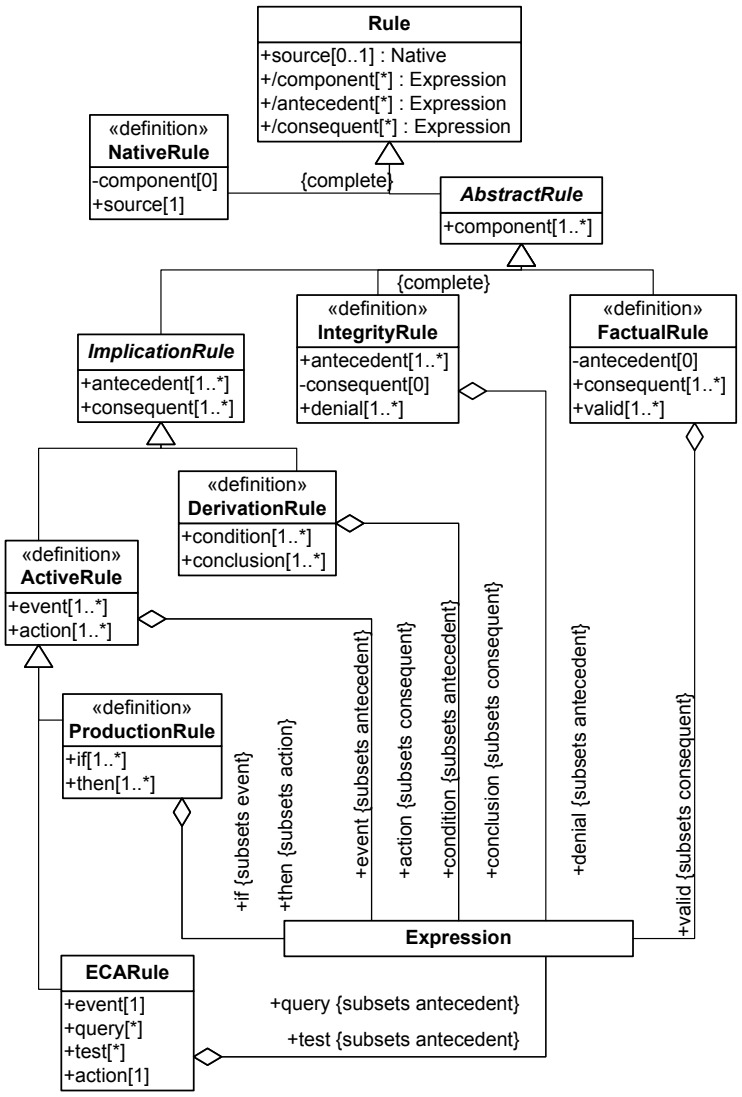

Figure 2.3: Rules (v0.10)

Example 2.1 *To illustrate, consider an* ECARule *notifying a teacher in case some administrative office processes a registration cancellation (*`ex:ev1`*), from a particular student to one of the teacher's lectures, provided that the student is considered a good student (according to some departmental criteria). In this example several languages are in fact used, namely: a language about people and notifying people (*`people:`*), an application/domain specific event language particular to some administrative office (*`office:`*), another domain specific query and test language (*`dept:`*), and a generic language for text manipulation (*`text:`*).*

```
ex:ev1 a :Expression;
  :is office:registration_cancelled;
  :having [ a :Parameter;
    :is office:student; :boundTo "St" ];
  :having [ a :Parameter;
    :is office:lecture; :boundTo "Lect" ].
ex:qry1 a :Expression;
  :is dept:lecture_teacher;
  :having [ a :Parameter;
    :is dept:lecture; :boundTo "Lect" ];
  :having [ a :Parameter;
    :is dept:teacher; :boundTo "Prof" ].
ex:tst1 a :Expression;
  :is dept:good_student;
  :having [ a :Parameter;
    :is dept:student; :boundTo "St" ].
ex:r1 a :ECARule;
  :event ex:ev1;
  :query ex:qry1;
  :test ex:tst1;
  :action [ a :Expression;
    :is people:notify;
    :having [ a :Parameter;
      :is people:person; :boundTo "Prof" ];
    :having [ a :Parameter;
      :is people:message; :boundTo "Msg" ];
    :with [ a :Variable;
      :name "Msg"; :mode :local;
      :equals [ a :Expression;
        :is text:merge;
        :with
          [ a :Variable; :mode :use;
            :name "Prof" :rename "1" ],
          [ a :Variable; :mode :use;
            :name "Lect" :rename "3" ],
          [ a :Variable; :mode :use;
            :name "St" :rename "2" ];
        :having [ a :Parameter;
          :is text:pattern;
          :literal """
            Dear %1,
            We are sorry to inform you that
            student %2 has cancelled his
            registration to your lecture %3."""
        ] ] ] ].
```

As an example of using logical variables for communicating among these different languages, consider example 2.1: a specific event occurrence provides a particular binding for variables `St` and `Lect` (identifying, respectively, a student and a lecture), given this binding, `Lect` is used to issue a query retrieving one or more bindings for variable `Prof` (the lecture professors, provided the query succeeds), whereas `St` is tested in order to filter out all non-good students, and finally the action is executed for all of the resulting substitutions (each containing all the previously bound variables, viz. `St`, `Lect` and `Prof`, and also an auxiliary local variable `Msg`, which is in fact a derived functional variable for each substitution).

*Language*s may be generic or application/domain specific. For instance, in example 2.1 instead of using `ex:qry1` based on the domain specific query element `dept:lecture_teacher` we could have used instead an XQuery [63] expression, this way re-using a generic language with a generic (but opaque) semantics (resulting in the domain specific semantics being hidden in `:data` and `:literal` opaque properties and so no longer available):

```
ex:r1 a :ECARule;
  ...
  :query [ a :Expression;
    :boundTo "Prof";
    :is xquery:opaque;
    :having [ a :Parameter;
      :is xquery:base-uri;
      :data "DEPT_LECTURES" ];
    :having [ a :Parameter;
      :is xquery:literal;
      :literal """
        for $t in doc($Lect)//teacher
        return $t/@id;""" ];
    :with [ a :Variable;
      :name "Lect"; :mode :use ] ]
  ...
```

On the other hand, also in example 2.1, instead of using a generic element like `text:merge` we could have used a domain specific language to build the `Msg` text, e.g.:

```
ex:r1 a :ECARule;
  ...
    :with [ a :Variable;
      :name "Msg"; :mode :local;
      :equals [ a :Expression;
        :is office:cancel_registration_warning;
        :having [ a :Parameter;
          :is office:teacher; :boundTo "Prof" ];
        :having [ a :Parameter;
          :is office:student; :boundTo "St" ];
        :having [ a :Parameter;
          :is office:lecture; :boundTo "Lect" ]
        ] ]
      ...
```

The recursive nature of *Expression*s, depicted in figure 2.2, also allows the use of different *Language*s in a single *component* through the composition of several *Argument Expression*s. This composition is achieved using generic algebraic *Language*s (e.g. an event algebra like Snoop [22, 6] or a process algebra like CCS [33, 11]) that may compose different application/domain specific (or even generic) *Language*s, as long as they share a common algebraic domain. For instance, by using an event algebra to define composite events, the event used in example 2.1 could be refined to trigger the rule only if if the teacher has previously confirmed the student merit:

Example 2.2 *The* RuleSet *below* contains *the* AbstractRule *already defined in example 2.1 and also a* NativeRule *for a DB2 trigger that shows how the firing event of the* AbstractRule *(*`ex:ev1`*) could be generated from an existing DB2 database.*

```
ex:rs1 a :RuleSet;
  :includes ex:r1;
  :includes [ a :NativeRule; :source ex:ns1 ].
ex:ns1 a :Native;
  :is officedb2:trigger;
  :having [ a :Parameter;
      :is officedb2:schema;
      :literal "office" ];
  :having [ a :Parameter;
      :is officedb2:user;
      :literal "trigger_adm" ];
  :having [ a :Parameter;
      :is officedb2:passwd;
      :literal "some_pwd" ];
  :having [ a :Parameter;
      :is officedb2:opaque;
      :literal """
        after delete on office.registration
            referencing old as reg
            for each row mode db2sql
            begin atomic
              values(raise_event(
                "registration_cancelled",
                reg.student, reg.lecture));
            end""" ].



ex:r1 a :ECARule;
  :event [ a :Expression;
    :is snoop:sequence;
    :taking [ a :Argument;
      :is snoop:first;
      :solve [ a Expression;
        :is teacher:confirmed_student_merit
        :having [ a :Parameter;
          :is teacher:id; :boundTo "Prof" ]
        :having [ a :Parameter;
          :is teacher:student; :boundTo "St" ] ];
    :taking [ a :Argument;
      :is snoop:next;
      :solve ex:ev1 ];
  ...
```

According to figure 2.3, a *Rule* may be either an *AbstractRule* (if it contains at least one *antecedent*/*consequent*) or a *NativeRule* (if it has no known *component*s, only a *source*). A *NativeRule* (as illustrated in example 2.2) is thus formed by a *Native source* which (as detailed later in figure 2.5) *is* a specific *NativeRuleConstruct* (of some *Language*), that *digs* an *Opaque-Parameter* containing its literal specification (e.g. a DB2 [42] database trigger or an Outlook mail routing rule).

*NativeRule*s provide a minimal level of integration for legacy (or any other non-$\boldsymbol{r^3}$) sub-systems, allowing a *RuleSet* to contain all the relevant rules. Ignoring the existence of *NativeRule*s, and simply considering them as outcasts, can lead to unexpected inferences/actions. Considering them, not only integrates these other sub-systems, but it also makes it explicit (in

the *RuleSet* description) that those *NativeRule*s and their associated *NativeEngine*s do have an impact on the behavior of the global system.

Nonetheless, *AbstractRules* are the preferred form of a *Rule* and it should be noticed that the *NativeRule* included in example 2.2 has an (intuitive) translation into an *AbstractRule*. Extending the interface of *NativeEngine*s to include also a *Compile* operation, although not considered currently, may provide an higher level of integration whenever such a translation is possible.

2.1.2.2 Messages and loading reactive rules

r^3 abstract *Message*s provide the foundation for the r^3 API, as defined in figure 2.4. Each actual message is to be seen as a serialization of a Resource Description (as in RDF [54]) of an r^3 abstract *Messsage*, defining a (possibly incomplete) RDF graph (optionally including references to external resources).

It is of particular relevance to clearly distinguish between the abstract concepts presented in figure 2.4 and any actual messaging framework (abstract or not) that is used to support them. The concepts presented here provide a conceptual view of *Message*s as required by r^3 *Engine*s. This conceptual view is to be supported by full-blown messaging frameworks and (concrete) protocols, preferably standardized ones. Such supporting frameworks/protocols/standards must account for asynchronous communication; examples may include (and are not restricted to): pure HTTP using a REST architectural style [24], FIPA [25], (Semantic) Web Services, JMS [49].

*Load*ing a *RuleSet* (like `ex:rs1` in example 2.2) into an *ECAEngine* (as introduced in figure 2.1) causes this *Engine* not only to activate the included *AbstractRule*s (like `ex:r1` in example 2.1) by resorting to the adequate *ExpressionEngine*s to *Evaluate* their *RuleComponent*s (as further detailed in section 2.1.2.3), but also to deliver any *NativeRule*s to the appropriate *NativeEngine*s where they must be natively embedded. For this purpose, r^3 *NativeEngine*s must provide an *Install* operation for embedding a *Native source* (like `ex:ns1` in example 2.2).

A *Load* (like any other) *Request* is sent by its *issuer Client* to the *target* of the appropriate *Interface* that some *Engine provides*, as illustrated in example 2.3. The actual *markup* to be used must be compatible with the one required by this *Interface*. This *Request* is then answered by the *Engine* (using the same *markup*) providing at least a *Response* (possibly asynchronous if dealing with an *AsynchClient* that *provides* a *notifyTo Interface* for this specific purpose).

The *ECAEngine* will return a *Loaded Response* as soon as the given *RulePackage* is *activate*d, otherwise, if it is not able to *activate* it, a *Terminated Response* will be returned. In the former case, the *RulePackage* will remain active until the *Client* sends a *Terminate* that will result in a *Terminated Response* acknowledging that the *RulePackage* deactivation is under way[7].

A *Load Request* causes the *ECAEngine* to issue additional subordinated *Request*s *target*ed to specific *LanguageEngine*s. For instance, as a result of the *Request* included in example 2.3,

[7] An *Engine* may also send a *TerminatedResponse* at any point (provided the *Client* is an *AsynchClient*), and not only after getting a *Terminate* from the *Client*, thus signalling that, from that moment on, there will be no more *Response*s to the initial *Request*. An *Engine* may choose to do so either because all *Response*s were already generated, or because it can no longer honor the initial *Request* due to some (probably unexpected) circumstance. In any case, a *TerminatedResponse* is always the last *Message* that *relatesTo* a particular *Request*, signalling that all the *Engine*'s resources associated with that *Request* are already released (or at least they must be assumed as such). A *Terminate* of a *Request* trivially succeeds, if it is sent after a *TerminatedResponse* for that same *Request* has been issued.

Example 2.3 *The following illustrates the loading of the* RuleSet `ex:rs1` *from example 2.2.*

```
@prefix ecaeng: <http://engine.nop#>.
@prefix ecarls: <http://engine.nop/rules/>.
@prefix clireq: <http://client.nop/requests#>.

ecaeng: a :ECAEngine;
  :provides ecaeng:interface;
  :notifyTo ecaeng:interface.
ecaeng:interface a :Interface;
  :target "http://engine.nop/service";
  :markup "http://rewerse.net/I5/NS/r3/2005".

 # The Client wants to Load a RuleSet
 clireq:rs1 a :Load;
   :issuer [ a :Client;
     :notifyTo [ a :Interface;
       :target "http://client.nop/notify";
       :markup "http://rewerse.net/I5/NS/r3/2005"
     ] ];
   :activate ex:rs1.

 # Rules have been activated, in this case the
 # Engine chooses to expose the Loaded rules
 # through the resource ecarls:rs1
 [] a :Loaded; :relatesTo clireq:rs1;
   :available ecarls:rs1.

 # Rules are active in the Engine and
 # the Client may retrieve the current
 # state of resource ecarls:rs1

 # The Client wants to deactivate
 # the previously Loaded rules
 [] a :Terminate; :relatesTo clireq:rs1.

 # All rules relating to clireq:rs1 are
 # assumed as deactivated/unloaded and
 # any associated resources as released
 [] a :Terminated; :relatesTo clireq:rs1.
```

Example 2.4 *Installing the* NativeRule `ex:ns1` *contained in* `ex:rs1` *from example 2.2. These* Message*s should be interleaved with the ones already included in example 2.3.*

```
@prefix engdb2:
   <http://office.nop/db2service#>.
@prefix ecareq:
   <http://engine.nop/requests#>.

 engdb2: a NativeEngine;
   :provides [ a :Interface;
     :target "http://office.nop/db2service";
     :markup "http://rewerse.net/I5/NS/r3/2005"
   ];
   :implements officedb2:.

 # Requested Load clireq:rs1 and before
 # responding Loaded must Install ex:ns1
 ecareq:ns1 a :Install;
   :issuer ecaeng:;
   :source ex:ns1.

 [] a :Installed; :relatesTo ecareq:ns1.
 # The Loaded Response can now be sent
 # after knowing that ex:ns1 is Installed

 # Got Terminate for clireq:rs1, so
 # must also Terminate ecareq:ns1
 [] a :Terminate; :relatesTo ecareq:ns1.

 [] a :Terminated; :relatesTo ecareq:ns1.
 # Now clireq:rs1 can also be Terminated
```

the *ECAEngine* would submit an *Install Request* (as shown in example 2.4) to the *target* of a *NativeEngine* that *implements* the `officedb2:` *Language* used in `ex:ns1`.

2.1.2.3 Evaluating reactive rules

Upon a *Load Request*, an *ECAEngine*, besides *Install*ing *Native* rules, as explained in section 2.1.2.2, also has to activate (viz. *Evaluate*) any *AbstractRule*s (viz. *ECARule component*s) contained in the given *RulePackage*. To achieve this, it must interface with several *ExpressionEngine*s (that actually provide the specific *Evaluate* implementation) according to the specific *LanguageConstruct*s involved in the different *RuleComponent*s, namely: an *event*, an optional *query* (or more than one), an optional *test* (or more), and an *action*. Since rule *component*s are *Expression*s and communicate with each other using logical variables, they are *Evaluate*d (by *ExpressionEngines*) *using* a context provided by several alternative *Substitution*s. Each *Substitution* must be unique and must enumerate all (and only those) variables involved in the *Expression* (possibly restricting some of them, by *binding* them to specific values or to other variables, as further detailed in section 2.1.2.4). While *Evaluating* an expression *using* a given context the *ExpressionEngine returns* all possible *Result*s, or none if the expression fails (meaning the *Expression* was *Evaluated* and *returns* no *Result*s). Besides (optionally) *using* a *Substitution*, any *Result* may also return a *literal* value that may be used as an *Argument* for another *Expression* or *boundTo* a variable. The actual expression to *Evaluate* must be a *NormalizedExpression* which is a restricted form of an *Expression with*out (and also not *boundTo*)

any declared *Variable* (cf. figure 2.6).

As an illustrative example consider the activation of the *ECARule* `ex:r1` of example 2.1. When an *ECAEngine* receives a *Load Request* for a *RulePackage* containing this rule (as `clireq:rs1` in example 2.3), it must activate the rule by issuing a subordinated *Evaluate Request* for the rule *event* (`ex:ev1`) to the *EventDetector Engine* that *implements* the (`office:`) *Language* (as shown in example 2.5).

This *EventDetector* will then signal back (asynchronously) each occurrence (of `ex:ev1`) it detects, by sending an *Evaluating Response* that *returns* the *Result* associated with the detected event. Note that if, at any point, the *EventDetector* concludes that the event can no longer occur (e.g. because deadline for registration cancellations has expired), it may send an *Evaluated Response* (depicted in figure 2.4) meaning that the event expression has no additional solutions and so the *Evaluate Request* is *Terminated*.

For each *Result* of *Evaluating* the *event* component, the *ECAEngine* will create a rule instance *using* the *Substitution* it *returns*[8]. For this rule instance, *query* and *test* components are then taken into consideration by issuing *Evaluate Request*s to the appropriate *QueryProcessor*s and *TestEvaluator*s, as shown in example 2.6.

Assuming no optimizations, first the *query* is *Evaluated*, extending the original *Substitution* with bindings for additional variables (possibly multiple ones, generating alternative *Substitution*s as in example 2.6), and if it succeeds, then the *test* is *Evaluated*. Both *test* and *query* may possibly filter out some of the *Substitution*s.[9].

Given the set of *Substitution*s for which all *query* and *test* components succeed, the *action* component is finally executed by an *ActionProcessor*, as partially shown in example 2.7.

For the *action* component, it is of particular relevance to ensure that all (distinct) *Substitution*s (for the variables involved in the *Expression*) are included in a single *Evaluate Request*, in order to account for any possible transactional behavior (associated with the *LanguageConstruct* defining the actual operational semantics of the *action*).

2.1.2.4 Dealing with language heterogeneity

A *Language defines* a set of *LanguageElement*s. For such a *Language*, at least one *Engine implements* the operational semantics specific to each *LanguageConstruct*, as previously depicted in figure 2.1. Figure 2.5 provides additional detail about the different *LanguageElement*s (viz. *LanguageConstruct*s, *ConstructParameter*s and *OperatorArgument*s), and defines how they are used to build the different kinds of *Language* specific *Construction*s (viz. *Native* rules and *Expression*s).

The semantics of a *Construction having* a set of *Parameter*s *is* defined by a *LanguageConstruct* that *has* a set of *ConstructParameter*s, and so the semantics of a *Parameter is* defined by the associated *ConstructParameter*. A *ConstructParameter* may be either a *FunctionalParameter*, that *use*s a value, or a *LogicalParameter* that *bind*s a variable. A functional *Parameter* must have a *literal* value (at least a *default* one, if a *literal* value is not present), unless it is *boundTo* a variable (in which case the *LanguageConstruct* semantics remains undefined until the variable is bound to a *literal* value). Logical *Parameter*s may be (explicitly) present or

[8]The *literal* value is to be discarded unless the *event Expression* is *boundTo* a variable, as detailed later in section 2.1.2.4, in which case this variable binding must be added to the *Substitution*.

[9]Notice that an *Expression* succeeds if it *returns* at least one *Result*, even an empty one like the *test Evaluated* in example 2.6.

Example 2.5 *Activating the* ECARule `ex:r1` *contained in* `ex:rs1` *from example 2.2. These* Message*s should be interleaved with the ones already included in examples 2.3 and 2.4.*

```
# Requested Load clireq:rs1 and before
# responding Loaded must launch Evaluate
# for ex:ev1 used in ex:r1
ecareq:ev1 a :Evaluate;
  :issuer ecaeng:;
  :using [ a Substitution;
    :binding
      [ a :Variable; :name "St" ],
      [ a :Variable; :name "Lect" ] ];
  :solve ex:ev1.

[] a :Evaluating; :relatesTo ecareq:ev1.
# Evaluate for ex:ev1 accepted, actual
# Results will be communicated later
# The Loaded Response can now be sent

# Later whenever an event occurrence is
# detected the literal event is returned
# together with the substitution
[] a :Evaluating; :relatesTo ecareq:ev1;
  :returns [ a Result;
    :literal """
      <offc:registration_cancelled
        xmlns:offc="http://office.nop/events#"
        offc:docid=
          "http://office.nop?did=DRC20062395"
      />""";
    :using [ a Substitution;
      :binding
        [ a :Variable; :name "St"
          :literal "S4521" ],
        [ a :Variable; :name "Lect"
          :literal "L144" ] ] ].

# Got Terminate for clireq:rs1, so
# must also Terminate ecareq:ev1
[] a :Terminate; :relatesTo ecareq:ev1.

[] a :Terminated; :relatesTo ecareq:ev1.
# Now clireq:rs1 can also be Terminated
```

Example 2.6 *Considering an ECARule instance, originated by the event occurrence* return*ed in example 2.5.*

```
ecareq:qry1 a :Evaluate;
  :issuer ecaeng:;
  :using [ a Substitution;
    :binding
      [ a :Variable; :name "Prof" ];
      [ a :Variable; :name "Lect"
        :literal "L144" ] ];
  :solve [ a :Expression;
    :is dept:lecture_teacher;
    :having [ a :Parameter;
      :is dept:lecture; :boundTo "Lect" ];
    :having [ a :Parameter;
      :is dept:teacher; :boundTo "Prof" ] ].
[] a :Evaluated; :relatesTo ecareq:qry1;
  :returns
    [ a Result; :using [ a Substitution;
      :binding
        [ a :Variable; :name "Prof"
          :literal "T57" ],
        [ a :Variable; :name "Lect"
          :literal "L144" ] ] ],
    [ a Result; :using [ a Substitution;
      :binding
        [ a :Variable; :name "Prof"
          :literal "T01" ],
        [ a :Variable; :name "Lect"
          :literal "L144" ] ] ].

ecareq:tst1 a :Evaluate;
  :issuer ecaeng:;
  :using [ a Substitution;
    :binding
      [ a :Variable; :name "St"
        :literal "S4521" ] ];
  :solve [ a :Expression;
    :is dept:good_student;
    :having [ a :Parameter;
      :is dept:student; :boundTo "St" ] ];
[] a :Evaluated; :relatesTo ecareq:tst1;
  :returns [ a Result ]
```

Example 2.7 *Executing an ECARule instance, with the extended (or filtered)* Substitutions *resulting after consideration of the rule conditions in example 2.6.*

```
# {Prof=T57, Lect=L144, St=S4521}
ecareq:ex1 a :Evaluate;
  :issuer ecaeng:;
  :using [ a Substitution;
    :binding
      [ a :Variable; :name "Prof";
        :rename "1"; :literal "T57" ],
      [ a :Variable; :name "Lect";
        :rename "3"; :literal "L144" ],
      [ a :Variable; :name "St";
        :rename "2"; :literal "S4521" ] ];
  :solve [ a :Expression;
      :is text:merge;
      :literal # simplified
        "%1, %2 not attending %3."
      ] ].
[] a :Evaluated; :relatesTo ecareq:ex1;
  :returns [ a Result;
    :literal "T57, S4521 not attending L144." ].
# {Prof=T01, Lect=L144, St=S4521}
# ... omitted ...

ecareq:act1 a :Evaluate;
  :issuer ecaeng:;
  :using [ a Substitution;
    :binding
      [ a :Variable; :name "Prof"
        :literal "T57" ],
      [ a :Variable; :name "Msg"
        :literal
          "T57, S4521 not attending L144." ] ];
  :using [ a Substitution;
    :binding
      [ a :Variable; :name "Prof"
        :literal "T01" ],
      [ a :Variable; :name "Msg"
        :literal
          "T01, S4521 not attending L144." ] ];
  :solve [ a :Expression;
    :is people:notify;
    :having [ a :Parameter;
      :is people:person; :boundTo "Prof" ];
    :having [ a :Parameter;
      :is people:message; :boundTo "Msg" ] ].
[] a :Evaluated; :relatesTo ecareq:act1;
  :returns [ a Result ]
```

omitted in a *Construction*, and if present they may be *boundTo* a variable or restricted to have a particular *literal* value.

A *LanguageConstruct uses* several *FunctionalParameter*s, *binds* several *LogicalParameter*s, and *digs* at most one *OpaqueParameter*. An *OpaqueParameter* is a *FunctionalParameter* but it cannot have a *default* value. As such an opaque *Parameter* must always be included in a *Construction* if it *is* a *LanguageConstruct* that *digs* an *OpaqueParameter*. The *literal* value of an opaque *Parameter* (or of the variable *boundTo* it) is to be parsed and interpreted by the *implementation* of the *LanguageConstruct* that actually *digs* it. Without this value, the semantics of the *LanguageConstruct* is incomplete. Furthermore, even when this value is known, the semantics of the *Construction*, although completely defined, remains opaque and unknown until actually submitted to an appropriate *LanguageEngine*.

As said before, there are two different kinds of *Construction*s, viz. *Native*s and *Expression*s. At rule level, the semantics of a *Native source* (of a *NativeRule*, as previously described in section 2.1.2.1) *is* defined by a *NativeRuleConstruct* that *digs* an opaque *Parameter* and optionally *uses* additional functional *Parameter*s. At rule component level, the semantics of an *Expression is* defined by an *ExpressionConstruct*. An *Expression* (as shown in figure 2.6) is either an *AbstractExpression* (viz. a *Term*, an *Aggregation*), or a *Formula*), or an *Opaque Expression*. Contrary to an *AbstractExpression*, the actual semantics of an *Opaque Expression is* only partially defined by its parametric *OpaqueConstruct*. In order to fully express the actual semantics of an *Opaque Expression* a non-transparent *literal Parameter* must also be provided, since an *OpaqueConstruct* always *digs* an *OpaqueParameter*.

A *Native Construction* is *Install*ed in a *NativeEngine* (as explained in section 2.1.2.2), whereas an *Expression* is *Evaluate*d by an *ExpressionEngine*. As described in section 2.1.2.3, the *Evaluate* of an *Expression* (or, more precisely, the *NormalizedExpression* to *solve using* several distinct input *Substitution*s) *returns* several *Result*s. Each *Result* may provide a *literal* value *using* a *Substitution*. Usually the returned *Substitution* is subsumed by one of the input *Substitution*s, but it may instead actually subsume several input *Substitution*s (by omitting some of the involved *Variable*s, to be considered as unbound). An empty *Result Substitution* (subsuming all input *Substitution*s) may be omitted altogether. The *literal* value of a *Result* is required to be present if the *Expression* is *boundTo* a *Variable* or used as an *Argument* of an algebraic *Operator*. If the *Expression* is *boundTo* a *Variable* a *binding* for this *Variable* is to be created (using the returned *literal* value) and joined with the returned *Substitution*. The *Substitution* resulting from this join is additionally joined with all the input *Substitution*s, yielding the actual set of output *Substitution*s associated with that *Result* (this set must not be empty). Given this, a *Result* may actually stand for several *Result*s, all sharing the same *literal* value (if there is one). The *Evaluate* of the *Expression* is said to succeed if it outputs at least a *Result*.

The *Evaluate* of an *Aggregation* is expected to *aggregate* the values of a set of *Variable*s (contained in the input *Substitution*s) optionally *groupedBy* a set of other *Variable*s (also contained in the input *Substitution*s). As such, the *Result*s of an *Aggregation* are further restricted: their *Substitution*s cannot contain any *binding* for the *aggregate*d *Variable*s (unless explicitly included as *groupedBy* at the same time[10]), and they must always include the *aggregate*d value as the *literal Result* value. An *Aggregation returns* at most a single *Result* for each set of distinct values of the *groupedBy Variable*s. The actual *Aggregation* to be performed *is* defined by parametric *Aggregator*s like `util:count` or `xcerpt:eval` in example 2.9.

[10]If an *Aggregation* is *groupedBy* on the same set of variables it *aggregate*s, it becomes a simple function that returns a single composed value for each set of distinct values of the given variables.

Example 2.8 *The following further illustrates the concept of* Language*s showing a possible definition for the* Language*s used in examples 2.1, 2.2 and 2.9.*

```
office:language a :Language;
  :defines office:registration_cancelled,
           office:student, office:lecture,
           office:registration, office:state.
office:registration_cancelled a :Functor;
  :binds office:student, office:lecture.
office:registration a :Functor;
  :binds office:student, office:lecture,
         office:state.

dept:language a :Language;
  :defines
    dept:term_completed,
    dept:course_lecture, dept:lecture_teacher,
    dept:course, dept:lecture, dept:teacher,
    dept:good_student, dept:student.
dept:term_completed a :Functor;
  :binds dept:course.
dept:course_manager a :Functor;
  :binds dept:course, dept:teacher.
dept:course_lecture a :Functor;
  :binds dept:course, dept:lecture.
dept:lecture_teacher a :Functor;
  :binds dept:lecture, dept:teacher.
dept:good_student a :Functor;
  :uses dept:student.

people:language a :Language;
  :defines people:notify,
           people:person, people:format,
           people:message.
people:notify a :Functor;
  :uses people:person, people:format,
        people:message.
people:format a :FunctionalParameter;
  :default "plain".

text:language a :Language;
  :defines text:merge, text:pattern.
text:merge a :OpaqueConstruct;
  :digs text:pattern.

officedb2:language a :Language;
  :defines officedb2:trigger,
           officedb2:schema,
           officedb2:user, officedb2:passwd.
officedb2:trigger a :NativeRuleConstruct;
  :uses officedb2:schema,
        officedb2:user, officedb2:passwd;
  :digs officedb2:trigger.

util:language a :Language;
  :defines util:count.
util:count a :Aggregator;
  :binds office:student, office:lecture.

xcerpt:language a :Language;
  :defines xcerpt:eval, xcerpt:construct.
util:count a :Aggregator;
  :uses xcerpt:construct.
```

Example 2.9 *To illustrate the use of* Aggregations *the following specifies an* ECARule *(*`ex:r2`*) that sends a map of all registration cancellations for a given course at the end of every term (re-using* `ex:qry1` *from example 2.1). Notice that the action uses only two variables (*`Mngr` *and* `CourseMsg`*) and so it sends a message for each distinct pair of values of these variables.*

```
ex:r2 a :ECARule;
  :event [ a :Expression;
    :is dept:term_completed;
    :having [ a :Parameter;
      :is dept:course; :boundTo "Course" ] ];
  :query [ a :Expression;
    :is dept:course_manager;
    :having [ a :Parameter;
      :is dept:course; :boundTo "Course" ];
    :having [ a :Parameter;
      :is dept:teacher; :boundTo "Mngr" ] ];
  :query [ a :Expression;
    :is dept:course_lecture;
    :having [ a :Parameter;
      :is dept:course; :boundTo "Course" ];
    :having [ a :Parameter;
      :is office:lecture; :boundTo "Lect" ] ];
  :query ex:qry1;
  :query [ a :Expression;
    :is office:registration;
    :having [ a :Parameter;
      :is office:state; :literal "cancelled" ];
    :having [ a :Parameter;
      :is office:student; :boundTo "St" ];
    :having [ a :Parameter;
      :is office:lecture; :boundTo "Lect" ] ];
  :query [ a :Expression; :boundTo "Num";
    :is util:count; :aggregate "St";
    :groupedBy "Course", "Lect" ];
  :action [ a :Expression;
    :is people:notify;
    :having [ a :Parameter;
      :is people:person; :boundTo "Mngr" ];
    :having [ a :Parameter;
      :is people:format; :literal "html" ];
    :having [ a :Parameter;
      :is people:message; :boundTo "CourseMsg" ];
    :with [ a :Variable;
      :name "CourseMsg"; :mode :local;
      :equals [ a :Expression;
        :is xcerpt:eval;
        :aggregate "Lect", "Num", "Prof";
        :groupedBy "Course";
        :having [ a :Parameter;
          :is xcerpt:construct; :literal """
          html { body {
            h1 { var Course },
            table {
              tr {
                th { "Lecture" },
                th { "Cancellations" },
                th { "Teachers" } },
              all tr {
                td { var Lect },
                td { var Num },
                td { ul {
                  all li { var Prof } } }
              } } } }""" ] ] ] ].
```

Every input *Substitution* must explicitly include all (and only those) *Variable*s involved in the *Expression* to *Evaluate*. If there is a single empty input *Substitution* (no variables involved) it may be omitted. The set of *Variable*s involved in an *Expression* includes the ones *boundTo* its *Parameter*s or *Argument Expression*s. If there is a *Variable boundTo* the whole *Expression*, it is also included in this set. Additionally, for an *Aggregation*, *groupedBy* and *aggregate Variable*s are also included. Besides these implicitly involved *Variable*s, an *Expression* may also be specified *with* several explicit *Variable* declarations[11] (explicitly declaring also any *Variable*s *boundTo* those). The set of all involved *Variable*s must be fully included in every input *Substitution* (regardless if they are bound or unbound, thus allowing all the *with* declarations to be omitted in a *NormalizedExpression*).

An *Opaque Expression* must be specified together *with* all the *Variable*s non-transparently referenced by its *literal* part. An *Opaque* may also *rename* its *Variable*s according to the naming conventions used by its *literal* part[12] (as shown in example 2.1).

For each *Expression*, either a single *Evaluate Request* is issued *using* all the distinct input *Substitution*s (this is mandatory for *action component*s, as said before, and also for *Aggregation*s); or several individual *Request*s may be issued (e.g. one for each *Substitution*, or one for each set of distinct values for *FunctionalParameter*s).

As usual, the scope of a *Variable* is the *Rule* that contains the *Expression*, unless its *mode* is declared as *local* in a particular *Expression*. The scope of such *local Variable*s is the *Expression* where they are explicitly declared, and they are usually used to hold *literal* functional *Results* (e.g. to be used as *Parameter*s, like `Msg` in example 2.1) or to express local join conditions between *Expression Argument*s.

All *Variable*s are assumed to have a *bind mode*, unless they are explicitly declared otherwise (*with* a *use* or *local mode*) or implicitly enforced to have a *use mode* by the context in which they are involved (viz. *groupedBy*, *aggregate*, *boundTo* a *FunctionalParameter*, or *boundTo* another *use*d *Variable*). An *Evaluate Request* can only be issued for a specific *Expression* if all the *Variable*s the *Expression use*s are actually bound to *literal* values[13]. The *Variable*s with a *bind* or *local mode* may be unbound at the time of the *Evaluate Request*. The *mode* of a *Variable* poses no restriction on the *Result*s of an *Evaluate* (for instance, a *bind Variable* may be bound in a *Result* but is not required to be so).

An explicit *Variable* declaration may also include restrictions to the domain of the *Variable*, namely: it may be restricted to having a specific *literal* value (which may be understood as a simple *binding*), or it may be declared that its value always *equals* one (or more) *Expression*s (as shown in example 2.1 for *Variable* `Msg`). Declaring that a *Variable equals* several *Expression*s defines the domain of the *Variable* as the intersection of the sets of *literal* values that each *Expression returns*. Notice that *equals* (and *test component*) *Expression*s can only produce new bindings for *local Variable*s, as such, the *Evaluate Request* for these *Expression*s can only be issued when all *Variable*s they *use* or *bind* are actually bound to literal values.

[11] For an *AbstractExpression*, these explicit declarations are restricted to the *Variable*s already implicitly involved in it or in one of its *Argument Expression*s. This restriction stems from the fact that the full semantics of *AbstractExpression*s *is* to be defined by their parametric *ExpressionConstruct*s (together with any *OperatorArgument*s, in case of algebraic *Operator*s).

[12] Notice that this *rename* information is not to be used to perform any form of blind textual replacement in the *literal* text before an *Evaluate*. Instead, this *rename* information is to be included in the *Substitution*s used when issuing the *Evaluate Request* (as illustrated in example 2.7).

[13] Notice that, actually, the *Variable*s *use*d by different *Expression*s define a partial evaluation order among them (most relevant when there are multiple conditions in a rule).

2.1.2.5 Additional details

For the sake of completeness of the presented $\boldsymbol{r^3}$ ontology, it is worth mentioning some additional details that, although not fully supported by the current prototype, are also included in the current version of the ontology.

Figures 2.1 and 2.4 include a *Register* operation that may be used to supply *Definition*s of *Language*s and *Engines* (usually to an *AlgebraEngine*, a *RuleEngine*, or a *BrokerEngine*). This operation is in fact supported by the current prototype, but disregarding any inconsistency matters that it may introduce.

Additionally, the $\boldsymbol{r^3}$ ontology also includes a form of macro substitution (as defined in figure 2.7) for *literal* values using *data* constants. The XQuery variation of example 2.1 illustrates the use of a *data* constant. This concepts are not supported by the current prototype and in fact they may be dropped in future versions.

Furthermore, if an *Engine* creates or updates resources, as a result of a *Request*, these resources may be included in any *Response*. In doing so, the *Engine* makes those resources *available* to the *Client*, as shown in figure 2.8, and so they become *AvailableResource*s (and *SharedResource*s as any public resource in the Semantic Web). For instance, during a *Load* conversation (as the one in example 2.3), chances are that the *Engine* will keep a materialized copy of the *Loaded RulePackage* (like `ecarls:rs1`) and the *ECAEngine* may choose to share that copy with the *Client* by means of an *AvailableRulePackage*.

Often these *AvailableResource*s will be persistent ones (with an associated URI), at least while the *Request* is not *Terminated*. These resources may even have a dynamic (evolving) nature (for instance `ecarls:rs1` may include information about the current state of evaluation of the different rules by including any *active RuleInstance*s) and the *Client* may retrieve updated descriptions, whenever he wishes to. These *AvailableResource*s are released (and usually their representation becomes unavailable) when the *Request* is *Terminated*. Currently, the $\boldsymbol{r^3}$ prototype does not expose any *AvailableResource*s, but internally it already relies upon *RuleInstance*s to keep the evaluation state of reactive rules (viz. using *Assertion*s to stand for *test component*s).

Notice that figure 2.4 does not include any functionality regarding consulting resources (and their representations). Nonetheless, consulting resources is not only needed to retrieve *AvailableResource*s, it is also core to the $\boldsymbol{r^3}$ API, since, as said before, a *Message* may be an incomplete RDF graph containing references to external resources. This means that in order to fully understand a *Request* (or a *Response*) an *Engine* (or *Client*) may have to retrieve additional resource representations[14]. Such functionality (of retrieving resource representations) is core to the (Semantic) Web itself and not specific to rule engines.

Since version v0.12 (see figure 2.8), the ontology also includes an incomplete set of specializations of *Interface* and *Expression* to be refined in future versions. These specializations are currently used for some minor implementation details (viz. ECA-ML integration), but are not yet consistently used throughout the ontology.

As a final note, it should be noticed that, the $\boldsymbol{r^3}$ ontology (although focused on reactive rules) addresses the general concept of a rule (viz. figure 2.3). Future developments will require at least a model for deductive rules (e.g. for specifying higher level events, queries or actions). In fact the current implementation already allows deductive rules (viz. Prova, XChange and Xcerpt) but only in a *Native* form. As such, upcoming versions of the presented ontology must

[14]How (and to what extent) this is actually done is up to the receiver, as long as it manages to understand the *Message* (correctly).

be compared and validated against results achieved by more general projects (not focused on reactive rules), both at the ontology level (e.g. SBVR [35], WRL [8, 23]), and at the language level (e.g. R2ML [52], RuleML [16], SWRL [26], SWSL [9, 10]). Naturally, any results achieved by the W3C RIF [56] Working Group may also prove relevant in this regard.

2.1.3 On the way to full r^3, growing up...

In this section, after a brief historic introduction (in section 2.1.3.1) regarding the evolution that led to the current r^3 ontology (v0.10), we summarize the current state (v0.12) of the prototype (in section 2.1.3.2) including its relationship to other REWERSE prototypes (in section 2.1.3.3), and we terminate by sketching future developments of the r^3 ontology (in section 2.1.3.4).

Additionally, several examples and Java excerpts of the implementation of some r^3 language engines are included (in appendix A). UML diagrams for other versions (past and future) of r^3 can be found at r^3 site), together with some notes on the changes between the different versions.

r^3 is an ongoing research project: the previous incarnation (version 0.10) was included as part of the I5-D5 deliverable [3] (*A first prototype on evolution and behaviour at the XML level* - M30), and was refined (version 0.12) towards the present deliverable I5-D7 (*Completion of the prototype scenario* - M30..36).

Within REWERSE, the final goal of r^3 is I5-D9 (*Prototype on the RDF/OWL level* - M36..42). As such, the r^3 ontology (version 0.20) constituted a valuable starting point for I5-D6 (*Reactive rule ontology: RDF/OWL level* - M30..36). Furthermore, work on B-Domain, based on r^3, has already started in close cooperation with WGA2 towards I5-D8 (*Realistic information integration application* - M36..42).

Beyond REWERSE, it is worth mentioning our goal of taking a WIDER (Web Integrated Development tools for Evolution and Reactivity) perspective upon evolution and reactivity on the Semantic Web (cf. [7]). Since the beginning, WIDER has constituted a major driving force for r^3, as it shares and builds upon r^3 goals, and further aims to capture and demonstrate the functionality needed in a development environment (and the support that must be built into the related execution model), in order to make Semantic Web application building, based on reactive rule-based models, viable (and as simple as possible).

All this makes r^3 an actual moving target. In fact, the r^3 prototype is expected to be updated to support version v0.2x of the r^3 ontology in the (very) near future (backward compatibility is desirable but not mandatory). The most up-to-date information about r^3 is to be found online (viz. [53]).

2.1.3.1 From childhood (v0.0x)...

The original motivation for using an OWL-DL ontology definition as the foundation for r^3 instead of an XML markup (viz. ECA-ML [2]) was that the heterogeneous approach made it quite difficult to fully formalize the XML markup. The level of expressiveness of the available standards (viz. DTD or XML Schema) was not enough. Instead of pushing these standards to the limit (or defining a new abstract syntax) it seemed only natural to resort to an adequate standard like OWL-DL (re-using already defined abstract syntaxes, e.g. RDF, and related concrete markups, e.g. RDF/XML). The last r^3 preview version (v0.00)[15] clearly shows this origin by even sketching a set of transformation rules from the ontology to ECA-ML.

[15] See http://rewerse.et/I5/r3/ for further details on this and other versions.

The first r^3 pre-release version to include an OWL-DL definition (and not only UML diagrams) was version v0.01. This version had a very brief life and was made public only to allow work on [6] to start. It was quickly replaced by version v0.02. This version was the base of quite useful discussions with WGI1 participants at the 2006 Rewerse Annual Meeting regarding the use of UML to define XML markups and OWL-DL ontologies.

Version v0.03 was the first full-blown version actually committed to the ontology level (and also the first to include *Language* definitions). Before this version, the "legacy" of ECA-ML was always present and language elements (operators, parameters or arguments) were identified by local names in a global language namespace and were not actual RDF resources (eventually identified by a global URI).

For what is worth, the interested reader may find the UML diagrams for all these versions at the r^3 site (viz. `http://rewerse.net/I5/r3/DOC/index.html`).

2.1.3.2 Through adolescence (v0.1x)...

Prototype: Available implementation. The r^3 prototype is implemented as a Java 5 Servlet [47], with minimal use of JavaServer Pages [46] for the prototype frontend. The r^3 Servlet provides an HTTP POST interface (embedding also Apache Axis [40] 1.3, for SOAP 1.1 support), and is packaged as a WAR[16] file, thus allowing easy installation under any servlet container[17].

The core of the r^3 ECA rule engine relies heavily upon Jena [48] 2.4 for RDF support (wrapped in Jastor-like [45, 27] Java classes). Nevertheless, the core of this engine is implemented using Prova (version 1.9, commit 14[18], patched[19]). The use of Prova embedded in a Servlet required the development of an extension to the Java/Prova integration mechanisms. The extended/patched version of Prova 1.9 is also included in the WAR package.

The prototype frontend allows to browse through different Use-Case scenarios formed by Groups of Examples, inspecting the actual requests that would be posted. Installed on an application server, it allows the actual posting of requests to that same server and the inspection of the synchronous responses. It is also possible to unpack the WAR file and browse the files offline. This offline configuration has weaker security restrictions and allows the posting of requests to any application server that you are aware of (just by redirecting the URL of the r^3 Server in the frontend). Currently the Use-Case scenarios available are B-Domain related and an ad-hoc collection of development tests.

Prototype: Communication details. Founding the r^3 API on an abstract messaging layer, like the one introduced in section 2.1.2.2, provides an architecture where other more specific layers may be semantically plugged providing support for: many concrete syntaxes (or serializations), alternative protocols, and even ontology reasoners. All that is assumed is that any message, at the time it gets delivered to the core of an r^3 *Engine*, is a fully materialized resource description of a *Message* (detailed in figure 2.4), and so the rule engine can focus on

[16] r3.war: including r3.jar with sources, released under the Apache License - version 2.0.

[17] Just copy the r3.war file to the `webapps` directory of your Apache Tomcat installation (tested with version 5.5.12).

[18] Regretfully, support for Prova 1.9 was dropped (before final release, cf. `http://prova.ws/forum/viewtopic.php?t=160`), in favor of the new upcoming version 2.0 (which includes a major rewriting of the inference engine and has not yet been released). The available Prova version (1.8) is more than 2 years old and does not constitute a viable alternative.

[19] `http://prova.ws/forum/viewtopic.php?t=149`

what constitutes its true added value (realizing reactive behavior), and rely on (re-using) other Semantic Web technologies for the rest.

For sure, a prototype like r^3, in order to provide some actual functionality, must make some choices and eventually commit to concrete syntaxes and protocols, and for that it must include specific code to support its choices, but if the internal architecture of the prototype conforms to the abstract messaging layer keeping a clean separation between these external layers and the core of the engine, the latter will be a re-usable "abstract" library (that actually implements the behavior of reactive rules).

Currently in r^3 we have chosen to implement an external layer (as said before, using a Java Servlet based on Axis 1.3) supporting simultaneously a SOAP 1.1 and a pure HTTP REST [24] style interface. The body of an HTTP POST request (SOAP wrapped or not) may be an ECA-ML [2] XML document (viz. the restricted ECA-ML subset defined in `http://rewerse.net/I5/r3/DOC/2005/eval.xsd`) or any of the XML based RDF serializations supported by Jena 2.4. Supporting N3 and other non-XML serializations (e.g. using CDATA XML nodes) would require a more involved architecture given the XML Element orientation of Axis (probably the recently released Axis2 [41] would be the best choice to achieve this).

A preliminary WSDL 1.1 [61] specification (`http://rewerse.net/I5/r3/DOC/2005/r3.wsdl`) defines the actual SOAP [58] 1.1 binding (`document` style - not `rpc` - with a `literal` body - not `encoded`). This specification is based on an XML Schema [62] (`http://rewerse.net/I5/NS/r3/2005/r3.xsd`) that defines a markup based on RDF/XML for the r^3 ontology. This XML Schema is obtained automatically from the OWL-DL definition of the r^3 ontology, using a generation tool still under development (which is not the focus of our work but that may prove relevant as a proof-of-concept for other REWERSE Working Groups, viz. WGI1).

It is worth noting that both the protocol and the markup could contribute to the actual serialization of *Message*s. For instance, one could use WS-Addressing [60] in the SOAP header, identifying a *Request* - `wsa:MessageID` - *Message* (to which a *Terminate* - `wsa:RelatesTo` - or *Response* - `wsa:ReplyTo` - *relatesTo*), and the specific subclass - `wsa:Action` - of the *Message*, reserving the SOAP body for providing the serialization of other properties of the *Message* (like parameters, for instance). Similar results may be achieved by introducing proprietary HTTP headers. The external (concrete) layers would be responsible for mapping all this information into a materialized resource description of a *Message*.

Also, these external layers may be capable of reasoning (this is something that is currently being considered for the r^3 prototype) improving their Semantic (Web) potential. For instance, once obtained the explicit RDF graph contained in a message, it is possible to scan that graph for any resources used only as objects of tuples (excluding those that are also used as subjects), and go grab their representation from the Web, thus building an extended graph (that would for instance include ontology information), and right before submitting it to the core of the engine (with the help of an ontology reasoner, like Pellet [51]) materialize also derived tuples.

Finally, the external layers can even be made adaptive by extending the *Interface* class with properties describing the actual markup and protocol supported. Currently the r^3 ontology only allows the specification of a *markup* URI, but a richer definition (e.g. introducing a new Markup class, instead of the *markup* property) must be considered in the future, eventually including markup transformations (e.g. XSLT [65]). The current r^3 implementation supports, in its external layers, a limited form of adaptive functionality by recognizing two different *markup*s (viz. `http://rewerse.net/I5/NS/r3/2005` for RDF/XML serializations; and `http://www.semwebtech.org/eca/2006/eca-ml` for ECA-ML); and by issuing local (in-memory) calls to *Install* and *Evaluate* (avoiding the overhead of remote calls) for locally supported

*component Language*s.

Prototype: Component Languages. The r^3 prototype includes, not only the main ECA rule engine, but also, several expression sub-engines supporting some particular rule *component Language*s. All these languages have been integrated using the r^3 development library (included also in the prototype). The most relevant *Language*s are the ones that integrate the Prova, XChange and Xcerpt languages.

An expression engine for (a demonstrative subset of) the SNOOP [22] event algebra also exists [6]. This *Language* was implemented using a previous r^3 version (v0.02) and still has to be upgraded to the current r^3 version.

It is also worth noting that, besides the *component Language*s included in r^3, the Göttingen prototype [12] (with which the r^3 prototype integrates, eventually restricted to the formalized subset of ECA-ML) also includes several component languages, as further detailed in section 2.1.3.3.

The definition of all the *component Language*s currently integrated in the prototype (viz.[20] `http`, `prova`, `xcerpt`, `xchange`, `xquery`, `xpath` and `util`) is being revised and may change in the near future. The OWL-DL ontologies[21] (and Java classes[22]) that currently define (and implement) these *Language*s are available online.

HTTP support. The `http` *Language defines* a set of *Functor*s, viz. `get`, `post`, `put` and `delete`. Any of these constructs *uses* an absolute request `uri` and *binds* a response `status` and `status-reason`. Additionally each of `put` and `post` *uses* a literal `body`. Only textual (viz. text/*) and XML application (viz. application/xml, application/*+xml) response content-types are supported and the response body is returned after being converted into an XML fragment according to the actual content-type (e.g. text/plain yields an XML text node). Any other response content-type is taken as an error, pretty much like failing to establish a connection or getting an HTTP response status not in the ok (2xx) range (with the exception of "Not Found"/404 and "Gone"/410 that are considered a failure, returning an empty set of results). The extension of these *Functors* so that each of them *binds* a set of parameters relating to (the most common) HTTP headers, is being considered[23].

For any of the above constructs, exists a similar *OpaqueConstruct* (with the same name prefixed with a 'v') that, instead of using a `uri`, *digs* a `vuri`. For these *OpaqueConstruct*s, variables may also be used as URI query parameters[24] (if renamed to properly URI encoded strings prefixed with a non-encoded '`=`', variable values must not be encoded); or bound to request/response HTTP headers[25] (if renamed to strings that are valid HTTP headers with the addition of a '`:`' prefix).

Also available is an `opaque` construct that *digs* a `literal` HTTP request[26] and *binds* a

[20] `http://rewerse.net/I5/NS/r3/2005/eval/⟨name⟩#⟨name⟩`

[21] `http://rewerse.net/I5/NS/r3/2005/eval/⟨name⟩`

[22] `net.rewerse.i5.r3.eval.⟨name⟩.Evaluator`

[23] Actually, e.g., `post` already *binds* a `soapaction`.

[24] Unbound query parameters are included in the query without the equal character.

[25] Unbound headers are omitted in the request, and bound if present in the response. Bound headers are included in the request and checked (if present) in the response. Beware when using bound general headers (or entity headers in a POST) since these relate to a specific HTTP message (or entity body) and may have different values in the request and in the response causing the construct to fail.

[26] Any included HTTP version is ignored.

response `status`, `status-reason` and `status-class`[27]. This construct does not deal with any redirections (3xx), it succeeds for any response status (contrary to the others that only succeed for 2xx response status) and fails if it is not possible to establish a connection or if an invalid response content-type is returned. It uses variables in the same way the other *OpaqueConstruct*s do.

Under consideration is a `transform` *Operator* that *takes* a `source` body (instead of a literal `body`), having a behaviour similar to the `post` and `put` constructs (to distinguish between them it *uses* a `method` parameter: `POST`, by omission, or `PUT`). Additionally, a `modified uri` event (with an optional poll `interval`) and a `modified-since` test are also being considered.

In appendix A.1 excerpts from the Java implementation of this HTTP engine are included together with some *Request* examples.

Prova support. The `prova` *Language* included in the $\boldsymbol{r^3}$ prototype *defines* a `native` construct that *digs* a `literal` Prova rule; and an `opaque` construct that *digs* a `literal` Prova goal and generates all the possible bindings (for the involved variables) satisfying the given goal (variable values are represented as string values[28] or XML elements/documents, when appropriate). Any of these `prova` constructs *uses* a `rulesdb` that identifies a particular rulebase (viz. Prova shell) to be used (empty by omission, denoting a default one; if given, it may be an URL to be retrieved for initialization of the rulebase).

No asynchronous functionality is currently provided, but *Functor*s for `rcvMsg` and `sendMsg` are under consideration[29].

In appendix A.2 excerpts from the Java implementation of this Prova engine are included together with some *Request* examples.

Xcerpt support. The `xcerpt` *component Language defines* a `native` construct that *digs* a `literal` Xcerpt rule; and an `opaque` construct that *digs* a `literal` Xcerpt query (trivially succeeds, if empty) producing several variable bindings. Any of these constructs *uses* a `rulesdb` that identifies a particular rulebase to be used (empty by omission, denoting a default one; if given, it may be an URL to be retrieved for initialization of the rulebase).

Additionally `opaque` *uses* an Xcerpt `construct` term to build and return a result for each produced tuple (empty by omission, in which case nothing is returned).

Two additional *Functor*s are also available to obtain the Xcerpt `term` or `program` corresponding to a given XML `document`.

Finally, two other constructs are currently under consideration: an `eval` *Aggregator* that *uses* an Xcerpt `construct` term to build the aggregated result based on the involved variables (if omitted, a `tupleset` of `tuple`s with the aggregated variables could be returned, for each group); and a `transform` *Operator* similar to the `opaque` construct, but that instead of evaluating a literal query against a rulebase, *takes* a `source` *Argument* and *uses* an Xcerpt `match` pattern to filter it (trivially succeeding, if empty) and possibly produce additional variable bindings (an Xcerpt `construct` term may also be provided to specify a transformation of the `source` to be returned, defaults to identity).

[27] `status-class` is the first digit of `status`.

[28] For what is worth, it should be mentioned that several problems remain unresolved relating to the representation of numbers vs. strings, and also that quotes (single or double) in Prova strings must be balanced and have no escape mechanism.

[29] Actually *Functor*s `rcvMsg` and `sendMsg` are implemented but restricted to using the Prova `self` protocol (which does not allow remote messages).

Excerpts of the Java implementation of this Xcerpt engine are included in appendix A.3 together with some *Request* examples.

XChange support. The `xchange` *component Language defines* a `native` construct that *digs* a `literal` XChange rule; a `detect` *OpaqueConstruct* that *digs* an XChange event `query` and signals any events matching it; a `raise` *Functor* that *uses* a `recipient` and an `event` body to generate an XChange event (if `recipient` is omitted it generates a local event); and an `execute` *OpaqueConstruct* that *digs* an XChange `transaction` and executes it.[30]

It should be stressed that all these *ExpressionConstruct*s (even if they are only action related, e.g. `execute`) return their results asynchronously, and so any `xchange` *Client* must always provide a *notifyTo Interface* in order to receive any possible results.

Excerpts of the Java implementation of this XChange engine are included in appendix A.4 together with some *Request* examples.

Xcerpt and XChange TCP servers. The *LanguageEngine*s that support Xcerpt and XChange languages are mainly wrappers around the existing TCP servers for both languages, so in order to use these languages the appropriate TCP servers must be running (the XChange wrapper requires also the Xcerpt TCP server).

Currently the set of TCP ports used for integrating with $\boldsymbol{r^3}$ is fixed. The Xcerpt and XChange TCP servers must be started with the appropriate parameters, viz. '`xcerptd 15003`' and '`xchange -p4711 -nhttp://localhost:4711`'. The available implementations for Xcerpt and XChange TCP servers are research prototypes (just like $\boldsymbol{r^3}$), so versioning is a bit elusive. Both prototypes are developed using Haskell [44], and binary versions may not be available for all platforms, or if available they might be a bit outdated.

The current version of the $\boldsymbol{r^3}$ wrappers requires the use of the most recent version of the sources (as of September 2006) for both prototypes. For a Windows (32 bits) platform the compiled binaries[31] are available for both servers (compiled using GHC [43] 6.4). Beware nevertheless that there seems to exist some issue preventing the XChange server from writing to disk (a Permission Denied error occurs). Otherwise, e.g. for a Unix platform, you may check the respective Xcerpt and XChange sites for a compatible compiled version for your preferred platform (which is not yet available at the time of this writing, at least not with the required changes). If an adequate binary is not yet available, you may have to compile it yourself.

Currently, to compile a compatible binary of any of the prototypes, you will have to grab the latest development snapshots available in the respective Subversion repositories. The current $\boldsymbol{r^3}$ version was tested with revision 10 of XChange (available in the XChange repository, but you may need to contact the XChange team in order to get access). Regarding Xcerpt, the $\boldsymbol{r^3}$ wrapper requires revision 595 (available in the Xcerpt repository, anonymous access is allowed), but beware that you will need to compile an updated version of XcerptDaemon.hs[32] (already approved, but not yet committed, by the Xcerpt team).

[30] Alternatively, mainly for compatibility with ECA-ML, it is also available an `opaque` construct that *digs* a `literal`. This `literal` may stand for a `detect`, `raise` or `execute` using for this purpose a syntax similar to XChange rules (starting with a line containing `ON`, `RAISE` or `TRANSACTION`, respectively; containing the appropriate XChange term; and ending with a line containing `END`).

[31] `http://rewerse.net/I5/r3/TST/install/xc.zip`

[32] `http://rewerse.net/I5/r3/TST/install/XcerptDaemon.hs.txt`

XQuery and XPath support. The r^3 prototype also supports XQuery and XPath as *component Language*s. For each of the two *Language*s (viz. `xquery` and `xpath`) it implements, based on Saxon 8.7 [57], an `opaque` construct that *digs* a `literal` XQuery or XPath query.

This construct returns the XML literal results of evaluating the opaque query for all the input *Substitution*s. XQuery external variable declarations (for all the variables included in the input *Substitution*s) are added if needed[33]. Additionally it is also possible to specify a `base-uri` or a context `document`[34].

By default a `raw format` is used and literal results are returned as string values or XML elements (when appropriate); unless one explicitly *uses* a `wrap format`. In the latter case, each result is always wrapped with Saxon elements (e.g. `result:sequence / element / attribute / atomic-value`) providing details of its type and value.

In appendix A.5 excerpts of the Java implementation of this XQuery engine are included together with some *Request* examples.

Basic utilities support. The r^3 prototype further includes the `util` *Language* that *defines* an introspective `bound` *Functor* with a *LogicalParameter* `value`. This *Functor* succeeds if the `value` parameter is bound, returning its alternative values[35].

Also available is a `replace-all` *Functor* with three *FunctionalParameter*s (viz. `text`, `pattern` and `replacement`) for regular expression text replacement (based on java.util.regex package[36]).

Furthermore another introspective *OpaqueConstruct* is currently under consideration (viz. `opaque` that *digs* a `literal` RDF/XML serialization of an r^3 *Message*, viz. a *Request*/*Terminate* is to be issued and the synchronous *Response* returned, and a *Response* is to be validated and "echoed"[37]).

The `util` *Language* may be extended in the future (if needed[38]). Some useful *Aggregator*s may be added (e.g. sum, avg, min, max, count). Also some basic comparison *Functor*s (e.g. eq, neq, lt, lte, gt, gte) and some basic *Operator*s (e.g. and/and-then, or/or-then, not, select-when-otherwise/if-then-else) may be added.

Building Component Engines using Java. Since the beginning of the work on the r^3 ontology and the r^3 prototype, it became clear that there was a considerable amount of code

[33] In `xpath`, generated variable declarations are always added, whereas in `xquery`, these are added only if a related error occurs during compilation of the XQuery expression.

[34] In `xpath`, any namespace prefix used in the context `document` is added to the set of static namespace prefixes available.

[35] Given `_:getW a :Expression; :is util:bound; :having [a :Parameter; :is util:value; :boundTo "W"]`, notice that `[a :Variable; :name "X"; :boundTo "W"]` is equivalent to `[a :Variable :name "X"; :equals _:getW]` (provided `W` is bound). But beware that `[a :Expression; :is _:op; :takes [a :Argument; :is _:oparg; :solve _:expr]]` is not equivalent to `[a :Expression; :is _:op; :with [a :Variable :name "W"; :mode :local; :equals _:expr]; :takes [a :Argument; :is _:oparg; :solve _:getW]]`; in the former `_:op` may consider non distinct values returned by `_:expr`, whereas the latter uses a variable (and variables retain only distinct values).

[36] Beware of \ and $ occurrences in the `replacement`, particularly if this parameter is *boundTo* a variable, you may need to escape them. You may be better of using the (less powerful, but safer) *Functor* `util:replace(util:text, util:old-text, util:new-text)` which is also available for literal text replacement.

[37] Eventually, echoing a *Response* seems to be the only functionality that cannot be achieved using the `http` *Language*.

[38] The non-introspective `util` constructs under consideration must be carefully considered, since most of them may not be needed if the `xquery` and `xpath` *Language*s are extended to export also their functions and operators [64].

that would be shared by r^3 main rule engines and the different r^3 expression sub-engines (hopefully to be developed for an unlimited set of *component Language*s).

To some extent, this sharing of code, may be solved by an adequate distributed component-based architecture. This is the case for the requirement to evaluate expression arguments (which is not trivial) that is common to all algebraic languages and shared by the main engines (given precisely the algebraic nature of those languages). This requirement may be solved/shared by the introduction of *BrokerEngine*s to be used both by the main rule engines and any algebraic engines. Nevertheless, other (not so minor) details like actually invoking a broker engine (or, e.g., dealing with variables and unification, and joining substitution sets) are hardly solved by distributed architectures.

Eventually, all fully compliant r^3 expression engines will share an enormous amount of code that is already part of an r^3 main engine. In order to minimize this issue and ease, as much as possible, the process of implementing and testing new expression engines, r^3 facilitates a Java library that abstracts away matters like communication protocols (HTTP, SOAP); binding variables and generating alternative solutions; or even the r^3 ontology itself (and dealing with the Jena RDF models).

The final goal is to allow developers of expression engines to focus on the specificities of the languages to implement; freeing them from r^3 details which are to be abstracted by tailored evaluation context components (under development at the time of this writing).

Currently, the functionality of the r^3 development library is still limited, nevertheless you may use it if you are willing to do so. It provides, e.g., a `net.rewerse.i5.r3.test.Tester` class for offline testing (without requiring the use of a Java application server) and even a `net.rewerse.i5.r3.test.dumpster` *Engine* for receiving asynchronous *Response*s. Actually this library is being used to develop all the, previously described, *component Language*s included in the r^3 prototype (as illustrated by the excerpts of Java code included in appendix A). Beware though, the r3 development library is being revised. Documentation will be made available as soon as it becomes a bit more stable. At the time of this writing, both the prototype and the development library are included in a single downloadable package[39], but that will change in the near future. Meanwhile, please contact us if you plan to use this library to develop any component language. We will do our best to fully support you!

2.1.3.3 Keeping in good company...

In the following, we try to outline what r^3 is, and what r^3 is not, by relating it to the work being developed by other REWERSE (viz. WGI5) participants, trying to evidentiate the complementary nature of r^3, and also at the same time trying to identify some of the research areas to which r^3 reaches out (without aiming at solving the related issues). Regarding these research areas/issues, and related work, r^3 takes a cooperative stand; hoping to become a useful Use-Case scenario where external results (either within or outside REWERSE) may be integrated and experimented.

Currently, much work in the research field of reactivity in the Semantic Web tends to follow a vertical approach designing specific reactive languages addressing particular issues (e.g. [36] for detecting events in - and updating - RDF documents, or [32] for triggers at OWL level). We believe that in order to achieve full reactivity at the global scale of the Semantic Web a complementary horizontal approach (like the one used by r^3) is required. Such approach actually promotes separation of concerns which may prove to be quite effective in such a broad

[39] `http://rewerse.net/I5/r3/TST/install/r3.jar`

research area. Particularly by separating the research on reactive behavior from the research about its different specificities like, for instance: how to propagate and detect events in the Semantic Web (that is an exclusive matter for *EventDetector*s, e.g. [6] or [18]); what is a good set of algebraic operators for composing events or actions (which is a matter for the appropriate *AlgebraEngine*s, e.g. [6] or [11]); how to support transactions in the Semantic Web (that is first of all a matter for *ActionProcessor*s but eventually will present additional requirements on reactive engines); and defining XML and RDF update languages and events (confined to *ActionProcessor*s and *EventDetector*s, respectively).

For most of these specific matters, the vertical approach is gradually providing answers that must be verified and experimented from a global point of view, actually leading to a whole that is more than the simple sum of the parts. Frameworks using this horizontal approach are particularly convenient for this purpose, bringing together and validating other more specific research results. Also, these frameworks provide an important foundation to investigate other issues of an higher level (e.g. user-oriented issues like collaborative and distributed integrated development environments for reactivity [7]) that are also required to realize the vision of a reactive Semantic Web. To achieve this latter goal, integration among the existing frameworks is also important and something that we plan to pursue taking the $\boldsymbol{r^3}$ perspective (most significantly [12], which is an XML based prototype based on the same general ECA framework, and to some extent [17] or even [59]).

GAU Göttingen, Informatik: ***Databases and Information Systems Group.*** The first public prototype (presented in [12]) trying to realize the objectives of the General ECA framework (detailed in [2]) was developed in Göttingen by the *Databases and Information Systems Group*. This prototype, pretty much like $\boldsymbol{r^3}$, is an ongoing WGI5 research project.

Since its first incarnation, this prototype has been fully based on the ECA-ML markup and among its main concerns includes: supporting languages/engines both framework aware and unaware, and defining/optimizing an all encompassing architecture including also event propagation/detection in the Semantic Web. The $\boldsymbol{r^3}$ prototype takes a complementary approach, where concrete markups, optimization and detailed architecture are not core concerns (e.g. all $\boldsymbol{r^3}$ sub-engines which are not framework aware are required to be wrapped by $\boldsymbol{r^3}$ compatible semantic layers), but instead expressiveness and semantic integration take the lead. Additionally, the more conservative approach taken by the Göttingen team (e.g. being markup based, starting only with opaque components, and grow from there one step at a time) has produced some quite interesting results on the level of specific component languages (viz. [32] and [11]). The decision to take a sematic approach in $\boldsymbol{r^3}$, already having in mind the future goals of WGI5 (namely I5-D9), was only possible thanks to the early results achieved by the Göttingen team, and we are quite confident that both prototypes will benefit from this early bilateral approach.

The $\boldsymbol{r^3}$ approach tries to maximize separation of concerns and focus as much as possible on a model for reactive rule evaluation, taking into consideration (but not focusing on) the specificities of the different component languages. For instance, matters like event detection/propagation in the Semantic Web do not have a direct impact on $\boldsymbol{r^3}$, besides requiring the ability to deal with asynchronous evaluations (an event specification is just an expression that will produce asynchronous results when submitted for evaluation to an appropriate event engine). Semantic Web event detection/propagation is a concern for event engines (viz. atomic event detectors/matchers, event brokers or algebraic event composers); as much as Semantic Web transactions are a concern for action engines. $\boldsymbol{r^3}$ is expected to work with different event (and action) engines (like [6]), regardless of the specific event detection/propagation (and

transaction management) algorithms/architectures they use. $\boldsymbol{r^3}$ is not expected to enforce any particular algorithm/architecture at this level, which, of course, also means that $\boldsymbol{r^3}$ does not propose any particular solution for these research issues. $\boldsymbol{r^3}$ simply does not try to solve these issues. Instead, $\boldsymbol{r^3}$ tries to integrate external research results, by defining how they should work together and by providing an adequate environment for testing these results. By focusing on this integration level, $\boldsymbol{r^3}$ aims at defining an abstract (parametric) model for reactive behaviour that may also constitute a good platform for higher level models leading, for instance, to WIDER [7] perspectives on evolution and reactivity on the Semantic Web.

The fact that $\boldsymbol{r^3}$ is semantically based, and as such not enforcing a particular concrete syntax/markup, allowed the easy integration of the two prototypes (reduced to a matter of translation between a concrete XML markup - ECA-ML - and the abstract RDF model/syntax of $\boldsymbol{r^3}$). This translation was implemented for the formalized subset of ECA-ML. In fact, $\boldsymbol{r^3}$ already formalizes at an abstract/semantic level matters that become harder to fully formalize at the concrete/syntactical level of ECA-ML (e.g. how to represent non-opaque expressions, which currently ECA-ML defines mainly by using a set of guidelines).

It is worth noting that, the preliminary results of this integration seem to suggest that, given the ontological description of rule component languages based on the $\boldsymbol{r^3}$ ontology, an automatic bi-directional translation could be achieved for the full ECA-ML (with the possible exception of some quirks contained in ECA-ML that stem from its syntactical nature, like marking up variables using non-XML constructs - '$' prefixes - within opaque texts). If such a reliable automatic translation could be achieved, ECA-ML could be fully formalized based on the $\boldsymbol{r^3}$ ontology.

Currently, the Göttingen prototype is growing closer to the semantic level of $\boldsymbol{r^3}$, and the first ontologies describing Languages, Services and LSR (Languages and Services Registry) can be found on the companion deliverable I5-D6 – "Reactive rule ontology: RDF/OWL level". The integration with the $\boldsymbol{r^3}$ ontology seems more than natural, and is to be pursued in the next period. In fact, these ontologies extend concepts already present in the $\boldsymbol{r^3}$ ontology, which means that just by making explicit this extension their expressivity would grow immediately.

H Skövde, IKI: *Software Systems Research Group.* RuleCore is an industry product that since its beginning, as an open-source project, is being developed in close cooperation with the *Software Systems Research Group* at Skövde.

Besides providing a graphical editor for reactive rules with some integrated testing and monitoring capabilities, RuleCore (being an industry product) has some interesting features related to engine startup, shutdown, crash and recovery. These features may prove relevant in the context of $\boldsymbol{r^3}$.

At the rule engine level, probably its main focus (and power) lies in its highly flexible complex event detector. Full integration (both at the atomic and algebraic level) between RuleCore event detector and $\boldsymbol{r^3}$ seems possible[40] and is to be pursued in the next months.

LMU München, Informatik: *Programming and Modelling Languages Unit.* The XChange language and prototype is being developed by the *Programming and Modelling Languages Unit* at Munich, and takes yet another perspective on reactive evolution for the Semantic Web. It differs from the other WGI5 prototypes by trying to maximize the benefits of having a single uniform language for ECA rules. Despite this mono-language perspective, the XChange

[40] With the possible exception of RuleCore `States` that have to be carefully considered.

work is not only complementary but also quite relevant to $\boldsymbol{r^3}$ and to the general ECA framework both at the rule component level and at the rule level.

At the rule component level XChange is currently integrated in $\boldsymbol{r^3}$ as an event and action language. Indirectly it is also integrated as a query/condition language through the Xcerpt language (since the condition component of an XChange rule is no different from an Xcerpt query). Currently the definition of both rule component languages is formed only of atomic (and opaque) constructs, yet both languages contain algebraic operators (for any of the components: events, queries and actions), and these should be modelled as such. The current expression engines for both languages are simple wrappers around the existing Haskell TCP servers and do not allow the level of integration that would be required for these algebraic operators. Also, separating the event algebra (by far the most promising XChange sub-language with an algebraic nature), could disrupt the homogeneity of the language and hinder the vision of XChange as a single and unified language for reactivity. Such possibility is open, and the $\boldsymbol{r^3}$ team is ready to cooperate, if at any point the XChange team wants to research the global applicability of XChange event query composition operators as algebraic operators to compose other event languages.

Still at the rule component level, raising implicit events upon concrete actions, and dealing with transactions, are still open issues in XChange. Regarding these issues, current XChange concerns are related to evolution at the XML level. Although $\boldsymbol{r^3}$ concerns regarding these matters are at a more abstract level, any results achieved by XChange may prove relevant to $\boldsymbol{r^3}$.

Also relevant to $\boldsymbol{r^3}$ are results achieved by XChange at the rule level, particularly those that could lead to higher or different levels of synergy/integration between the different rule components. Any such results would probably question the structural model or the integration model used by $\boldsymbol{r^3}$, leading to adequate extensions. Currently, for any XChange rule it is possible to write a fully equivalent $\boldsymbol{r^3}$ rule using the available rule component languages. Nevertheless, the latest results achieved by the XChange team (summarized in [21]) already identify possible extensions to the language. Some of those extensions are already present in $\boldsymbol{r^3}$, other confirm the the need for a richer $\boldsymbol{r^3}$ ECA model already induced by Prova contextual reactive rules.

TU Dresden: *Biotechnological Centre (Biotec).* The Biotechnological Centre (Biotec) at Dresden uses the general-purpose open source programming language Prova (originally presented in [28]). The Prova language aims at integrating three programming paradigms, viz. logic (Prolog), imperative (Java) and reactive (based on event-driven goals/rules).

Prova has been chosen as the programming language for the core of the main $\boldsymbol{r^3}$ engine, and a rule component language is also available allowing Prova to be used to build query components (datalog-like) and also as an action language. The $\boldsymbol{r^3}$ team has been following closely the development of Prova and actively cooperating for more than a year with the Prova team towards the release of Prova 1.9. As a result of this work some extensions to the mechanisms of integration between Prova and Java have been proposed.

Prova is by itself a reactive language and Prova reactive rules include also a form of contextual reactive rules (born out of actual development requirements) that provide a declarative way to selectively activate specific reactive rules in a particular context. This concept of contextual reactive rules is not present in the $\boldsymbol{r^3}$ model and is an open issue.

2.1.3.4 Towards maturity (v0.20)...

Version v0.20 will have a new namespace (viz. `http://rewerse.net/I5/NS/2006/r3#`) and compatibility with v0.12 will not be mandatory.

This will be the first version to model rule languages (and not only rule component languages), allowing for instance the definition of contextual rules. Some other relevant changes to be considered in this version include:

- rule variables will be re-introduced and ground resources removed; macro substitution only seems to make the prototype more complex, and appropriate restriction of rule variables (as originally intended) may yield similar results;
- the result of an evaluation may group a set of substitutions for each value, and this grouping must be considered for event results when creating ECA rule instances (only one per group); as a motivating example consider the re-use of `ex:ev1` (included in example 2.1) to define a rule that, instead of reacting to a single cancellation (like `ex:r1`), collects all cancellations since `office:registration_open` until `dept:term_terminated`;
- solutions may be considered as an informative alternative extending (and thus always including) the normative substitutions currently used when evaluating an expression;
- opaque constructs should not be restricted to a single opaque parameter and should be allowed to take arguments;
- the introduction of type information may also be considered.

2.2 Use-Case Scenario

The use-case scenario considered for testing the $\boldsymbol{r^3}$ prototype is based on the scenario *Updates and evolution in bioinformatics data sources* described in [5].

2.2.1 Scenario overview

In bioinformatics there are many publicly accessible data sources, which are often mirrored locally and integrated with other data. The bioinformatics use case discusses four specific data sources: PubMed, a database of 12.000.000 biomedical literature abstracts, GeneOntology, an ontology for molecular biology, with 19.000 concepts, PDB, a database with some 25.000 protein structures, and SCOP, the Structure Classification of Proteins, which groups PDB structures according to their evolutionary relationships. The scenario is concerned with mirroring these data sources locally, keeping them consistent and integrating them, and its relation and usefulness to the work being developed in working group A2 is clear. These public bioinformatic data sources can be classified as primary and secondary (i.e. derived from one or more primary data sources, e.g. SCOP or Astral[41]). The scenario, besides the four specific data sources, also considers online applications that integrate them, viz. (Gene)Ontology-based Literature search, GoPubMed, which given the appropriate interfaces may also be considered as secondary data sources. Users of such data sources (final users or applications) keep local copies of these primary and secondary databases and often derive tertiary data sources. Keeping local and

[41] The ASTRAL Compendium for Sequence and Structure Analysis, `http://astral.berkeley.edu/`

remote databases in sync and consistent is an important problem, which requires techniques to deal with evolution and reactivity.

PubMed PubMed, `http://www.pubmed.gov/`, the main biomedical literature database references over 12.000.000 abstracts. It has grown by some 500.000 in 2003 alone. Besides biology it covers fields such as medicine, nursing, dentistry, veterinary medicine, the health care system, and the preclinical sciences. PubMed contains bibliographic citations and author abstracts from more than 4,600 biomedical journals published in 70 countries. Abstracts date back to the mid-1960's. Coverage is worldwide, but most records are from English-language sources or have English abstracts. PubMed is available in XML.

PDB Another source, which is widely used in the A2 group, is PDB, `http://www.rcsb.org/pdb/`, the protein databank. PDB is a repository of the atomic coordinates of proteins and nucleic acids. PDB entries contain among others, besides the coordinates, the resolution at which the coordinates have been obtained, the authors (who submitted the data), literature references (some recorded in PubMed), the species the data is coming from. PDB is updated every week and is available as XML and flat file.

SCOP Also widely used in the A2 group, SCOP, `http://scop.mrc-lmb.cam.ac.uk/scop/`, classifies PDB structures according to their evolution. SCOP contains four main structural classes, which are refined into some 1000 structural families of proteins. SCOP is updated every 6 months and is available as flat file. Inconsistencies introduced by PDB updates, given their weekly update rate, are a possibility not to be discarded lightly.

GeneOntology GeneOntology (GO), `http://www.geneontology.org/`, is a controlled, hierarchical vocabulary. GO has been designed for the annotation of genes. It comprises over 19.000 terms organized in three sub-ontologies for cellular location, molecular function and biological process. GO was initially created to reflect gene function of fruitflies, but has expanded to encompass many other genomes as well as sequence and structure databases. The hierarchical nature of GO allows one to quickly navigate from an overview to very detailed terms. As an example, there are maximally 16 terms from the root of the ontology to the deepest and most refined leave concept in GO. GO is available in free text, XML and as database. The XML version is updated on a monthly basis. The deliverable A2-D2 contains further details on GO.

GoPubMed GoPubMed, `http://www.gopubmed.org/`, a tool developed by TU Dresden in the A2 group, uses GO to structure large amounts of relevant literature to realize the concept of *Ontology-based Literature search*[42]. GoPubMed submits keywords to PubMed, extracts GO-terms from the retrieved abstracts, and presents the relevant sub-ontology for browsing. GoPubMed has a number of advantages, e.g. users get a high-level overview of the whole search result and are not forced to view multi-dimensional and thus often incomparable articles in a one-dimensional list. The latest versions of GoPubMed also integrate with Wikipedia.

[42]MeshPubMed, `http://www.meshpubmed.org/`, also based on PubMed and developed by TU Dresden, realizes the same concept of *Ontology-based Literature search* using the hierarchical vocabulary "Medical Subject Headings", MeSH, `http://www.nlm.nih.gov/mesh/`.

2.2.2 Use-Cases overview

This scenario, cf. [5], includes the following two specific use-cases.

Use Case 2.2.1 (Caching and Actuality of data in GoPubMed, PubMed, and GO) *GoPubMed is a distributed application: A query entered in GoPubMed is submitted on-the-fly to the remote PubMed site, which returns relevant articles. These are then annotated by GoPubMed with relevant terms from the GO, a local copy of which is residing at the GoPubMed site. To integrate the three sources, the GoPubMed application needs to exhibit reactive behaviour. On the event of a user query, a request is sent to PubMed. On the event of an answer from PubMed, a local cache is consulted. If the abstracts are not cached, then GoPubMed sends another message to PubMed requesting the abstracts. On receipt of them, they are annotated. Finally, the results are compiled an presented to the user. Overall, there are different distributed data sources, which are communicating with each other using event-condition-action rules.*

Use Case 2.2.2 (Mirroring, Actuality, and Consistency of data in SCOP and PDB) *The original SCOP data is published on a website hosted in Cambridge. A researcher may have a copy of SCOP on his laptop besides a copy hosted at his university. The copy on the laptop is not up-to-date, so that the researcher usually uses the remote database, but when offline he is forced to use the local laptop copy. The researcher wants to transparently access SCOP and this access needs to handle the preference of the remote SCOP copy over the local SCOP copy.*

A reactive agent acts as a wrapper of the original SCOP site and data and upon the event of a new release it informs a local agent, who updates the local SCOP copy.

Updates of PDB can lead to inconsistencies as SCOP is derived from PDB and as PDB is updated weekly, while SCOP is only updated every six months. If a PDB entry gets withdrawn between two SCOP releases, then the constraint is violated that every SCOP entry should have a PDB entry it is derived from. The constraint can be satisfied if we know which PDB entry replaces a withdrawn PDB entry. Then the local SCOP copy can be updated accordingly.

Relationship to I5-D8: Realistic information integration application These use cases constitute the starting point for the upcoming WGI5 I5-D8 deliverable (due at month 42) dedicated to the *development of realistic information integration application in bioinformatics in collaboration with working group A2*. The work on B-Domain, reported here, is a first attempt towards integrating the different components needed to realize these use-cases, eventually identifying and anticipating future needs/limitations.

2.3 Current state

The current implementation of B-Domain is available online and allows the user to monitor new PDB structures. Given a criterion specified at the level of the advanced query functionality available at the PDB site, a storage (viz. a personalized PDB mirror) is created to keep all the new structures that satisfy that criterion.

2.3.1 Available prototype, quick start

B-Domain Site: `http://di150.di.fct.unl.pt:15080/b-domain/`.

r^3 Site: `http://rewerse.net/I5/r3/`.

r^3 Download: `http://rewerse.net/r3/TST/install/`.

r^3 Prototype: `http://di150.di.fct.unl.pt:15080/r3/TST/`.

Check the storyboard available at `http://di150.di.fct.unl.pt:15080/b-domain/demo`, or try it yourself at `http://di150.di.fct.unl.pt:15080/b-domain/monitor_form.jsp`:

- Define the monitor criterion[43] that new PDB entries must satisfy in order to be included in your personal mirror;
- Submit and B-Domain will create a storage for your own personal mirror and will also load the appropriate ECA rules that will ensure the appropriate monitoring of new PDB entries cf. the provided criterion;
- Bookmark the returned URL for the created storage;
- You may see the ECA rules responsible for the actual update of your storage in the r^3 ECA engine model (`http://di150.di.fct.unl.pt:15080/r3/service`), or by analyzing the latest logged requests in the r^3 dumpster (`http://di150.di.fct.unl.pt:15080/r3/service/dumpster`);
- Visit the bookmarked URL periodically to see the new PDB entries conforming to your criterion (you will have to wait for the next PDB release[44] - updated every Tuesday/Wednesday - actually containing new structures that fit your monitor criterion).

2.3.2 Description

PDB provides an advanced query facility (`http://www.rcsb.org/pdb/search/advSearch.do`) which is commonly used by bioinformatic researchers to locate relevant structures. Also, every week the new structures added to PDB are listed in an RSS feed (`http://www.rcsb.org/pdb/rss/LastLoad`). B-Domain combines these two functionalities providing an infra-structure by which researchers may define their own personalized mirrors of PDB. These personalized mirrors should contain all the new structures relevant for each researcher, and for each new structure (depending on its "kind") mirror the relevant information (either from PDB or other sources, e.g. PubMed).

The current implementation allows a researcher to create a storage (viz. personalized mirror) by specifying a criterion to filter the relevant new structures. This criterion is currently specified by an XML document (cf. `http://di150.di.fct.unl.pt:15080/b-domain/schemas/2007/criteria.xsd`) and mimics some of the PDB advanced query parameters. Upon submission of a criterion, an URL is returned. The researcher may later use this URL to browse to its personalized mirror.

The currently available set of PDB advanced query parameters is only a restricted set of the full set available at PDB, and is meant primarily as a proof-of-concept. Nevertheless a complete survey of the existing parameters has been conducted and work is on the way towards defining an OWL ontology to be used as a model for defining the criteria instead of the current XML Schema. The final goal is not to develop a B-Domain front-end for the PDB advanced query

[43] An example of a monitor criterion is available at `http://di150.di.fct.unl.pt:15080/b-domain/schemas/examples/criteriaExample2.xml`.

[44] RCSB Protein Data Bank - This week's new structures: `http://www.rcsb.org/pdb/rss/LastLoad`.

Example 2.10 *To illustrate the kind of plans that B-Domain may generate, consider the following very simple plan formed by two rules that use the B-Domain language and that are chained together by an XChange event.*

```
ex:r1 a :ECARule;
  :event [ a :Expression;
    :is b-domain:newStructure;
    :boundTo [ a :Variable; ;name "PDBID" ] ];
  :test [ a :Expression;
    :is b-domain:satisfies;
    :having [ a :Parameter;
      :is b-domain:pdbId; :boundTo "PDBID" ];
    :having [ a :Parameter;
      :is b-domain:criteria; :boundTo "MyCriterion" ] ];
  :action [ a :Expression;
    :is xchange:raise;
    :having [ a :Parameter;
      :is xchange:event;
      :literal """
        newpdb {
          id {var PDBID},
          crit{var MyCriterion},
          for{var MyStorage}
        }""" ] ].
ex:r2 a :ECARule;
  :event [ a :Expression;
    :is xchange:detect;
    :having [ a :Parameter;
      :is xchange:event;
      :literal "newpdb {{ id {var Id}, for{var St} }}" ] ];
  :action [ a :Expression;
    :is b-domain:store;
    :having [ a :Parameter;
      :is b-domain:storage; :boundTo "St" ];
    :having [ a :Parameter;
      :is b-domain:pdbId; :boundTo "Id" ] ].
```

functionality, but instead to make this functionality available to be used (at an RDF level) in different front-ends allowing the creation (and browsing) of personalized mirrors. As possible front-ends, two main options are currently under consideration, viz. the Chemera[45] application and the Personal Reader[46] framework, both developed within REWERSE by members of WGA2 and WGA3, respectively).

The personalized mirrors are maintained through the use of appropriate reactive rules. For this, B-Domain provides a domain specific language and generates reactive rules realizing a (more or less) complex plan that is responsible for detecting relevant updates and retrieve the relevant information to be stored/updated. Currently the plans generated by B-Domain are far too simple (cf. example 2.10) for the intended use. They store a simple link for the PDB page dedicated to the new structure. These plans are to be improved with actual replication of information available not only at PDB but also at SCOP and PubMed.

[45] http://www.cqfb.fct.unl.pt/bioin/chemera/Chemera/Intro.html

[46] http://www.personal-reader.de/

2.3.3 Coverage of E&R Use-Cases

Regarding use-case 2.2.1, it embodies the concept of reactive querying where synchronicity drives the use of reactive rules. Since reactive rules have an asynchronous nature, this use-case must be considered very carefully and it is not clear at this point if a reactive model will constitute an appropriate solution to support this use-case.

The current implementation is restricted to a proof-of-concept related to use-case 2.2.2. A more comprehensive coverage of the mirroring concerns contained in this use-case is to be expected in the upcoming moths, and if possible consistency concerns should also be addressed.

2.3.4 Interoperability with other WGI5 Prototypes

$\boldsymbol{r^3}$ component structure accounts for different serializations for requests and responses. Currently, $\boldsymbol{r^3}$ supports RDF/XML (fully materialized according to the $\boldsymbol{r^3}$ ontology, integration with an OWL-DL reasoner is expected in the near future), and ECA-ML (to the extent formally specified by MARS). Supporting other serializations is a matter of providing/implementing appropriate bidirectional translators to/from the $\boldsymbol{r^3}$ ontology. This actually means that rules and expressions can be submitted using any of the supported serializations (responses use the same serialization of the respective requests).

Furthermore, the $\boldsymbol{r^3}$ ECA engine doubles as a broker engine for any language engine known to it (viz. previously registered). These languages and engines have to be described according to the $\boldsymbol{r^3}$ ontology (examples are available online for some of the MARS languages/engines).

Given all this, full interoperability with the MARS prototype, although not fully tested yet, should not be an issue as long as the MARS prototype associates the $\boldsymbol{r^3}$ namespace with an $\boldsymbol{r^3}$ ECA engine (viz. MARS recognizes an $\boldsymbol{r^3}$ language supported by an $\boldsymbol{r^3}$ broker).

Interoperability with the XChange prototype is achieved by modelling XChange and Xcerpt as component (and native rule) languages at an opaque level. Achieving interoperability at an abstract level, e.g. allowing the composition of XChange and Xcerpt with other languages, is possible.

Finally, it is worth stressing that interoperability is a top priority for $\boldsymbol{r^3}$ and that the $\boldsymbol{r^3}$ team is fully committed towards providing the best support possible to anyone interested (within or outside REWERSE). The actual integration of Prova as an opaque language, or the integration of ruleCore at an abstract level (which is being pursued in close cooperation with the industry), are examples of this commitment.

2.3.5 Future work

B-Domain criteria are to be taken to an ontology level and are to cover as much as possible of the PDB advanced query parameters, considering also its extension to PubMed, SCOP and GO parameters.

B-Domain generated plans are expected to retrieve and mirror information from PDB, SCOP and PubMed. The extent of mirrored information is to be personalized. Personalization depending on further criterion is desirable.

B-Domain personalized mirrors must be provided in an RDF/XML compatible markup (according to an OWL ontology). Integration with RSS feeds is desirable.

Integration with Chemera application and the Personal Reader framework is to be pursued, and submitted if possible to real use by bioinformatic researchers collecting feedback about the actual applicability of the results.

Reactive query scenarios (viz. integration with GoPubMed, use-case 2.2.1) are to be further researched, either realizing them or identifying the limitations that prevent the use of reactive models in such query-based scenarios.

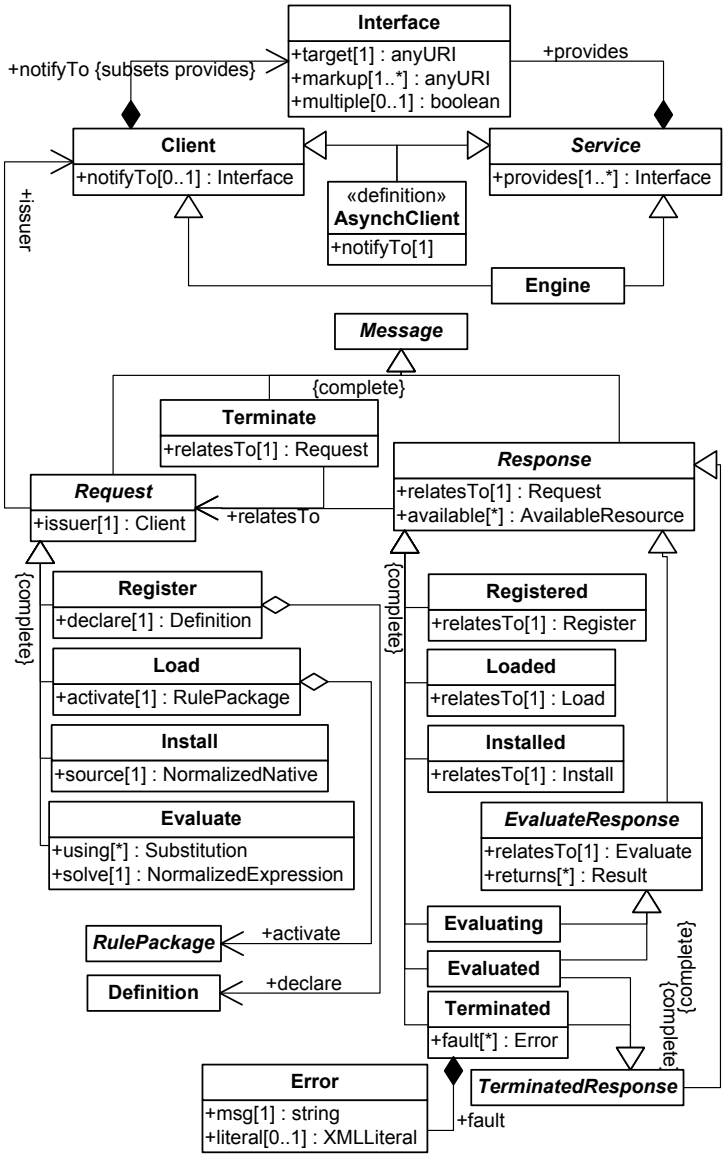

Figure 2.4: Services and Messages (v0.12)

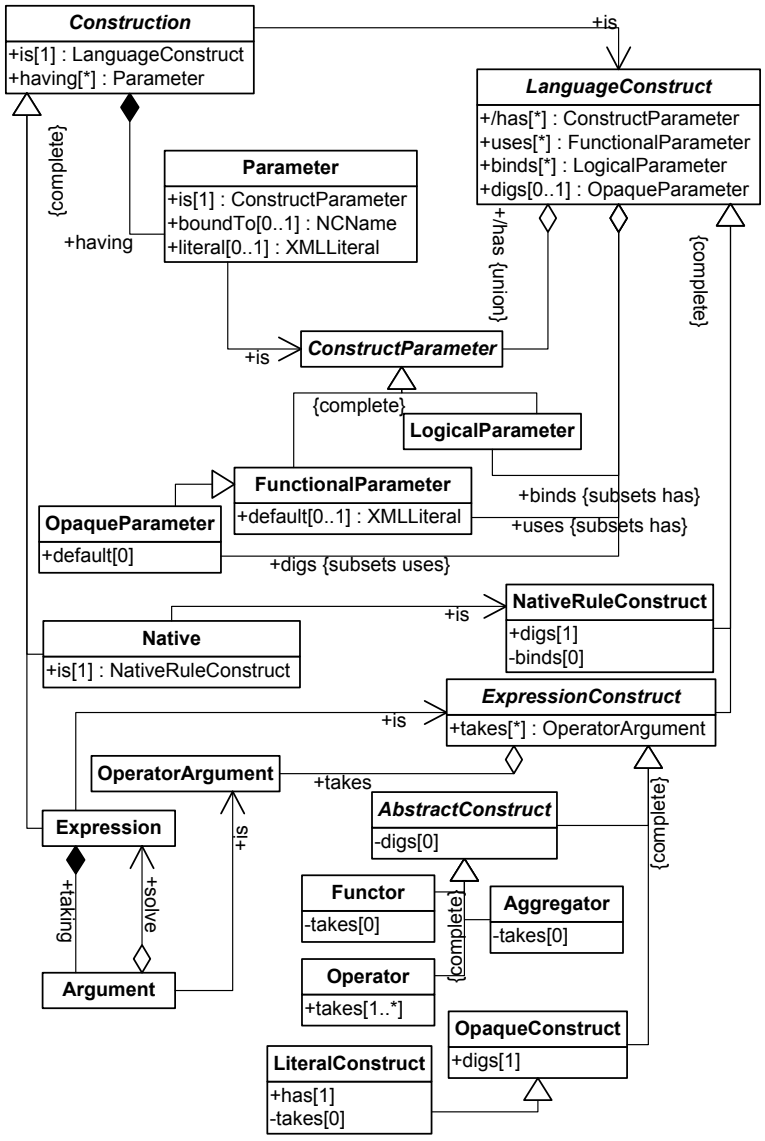

Figure 2.5: Languages (v0.12)

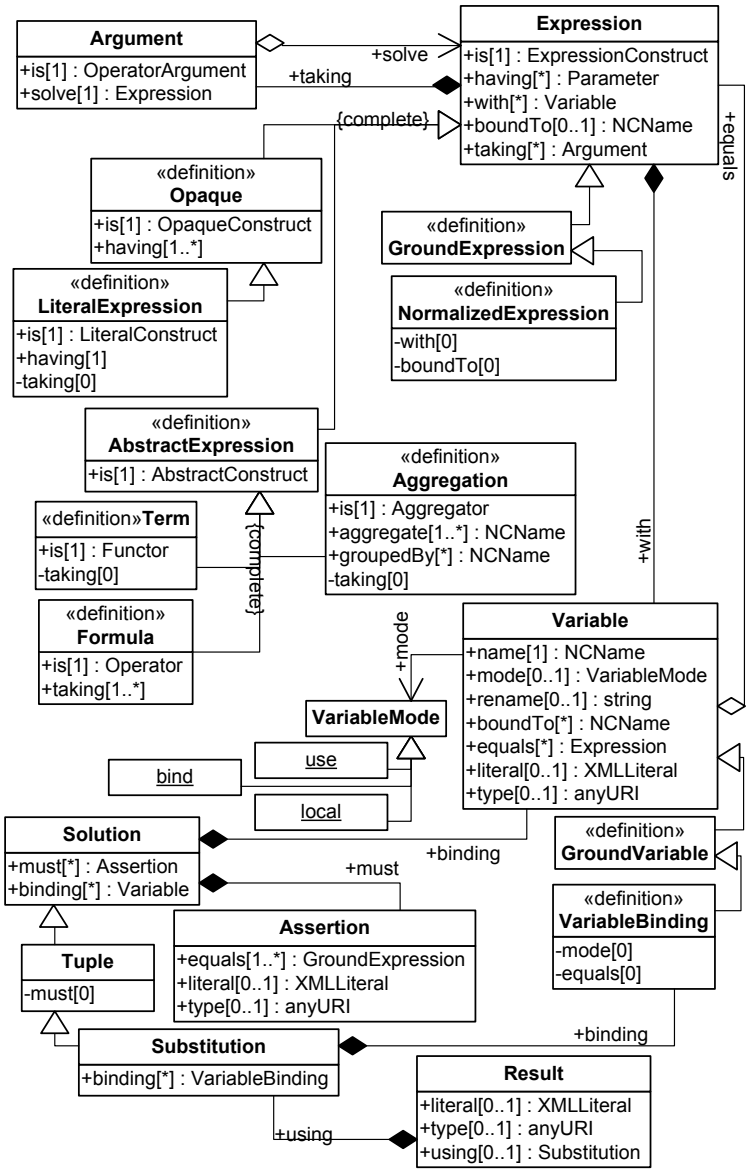

Figure 2.6: Expressions (v0.12)

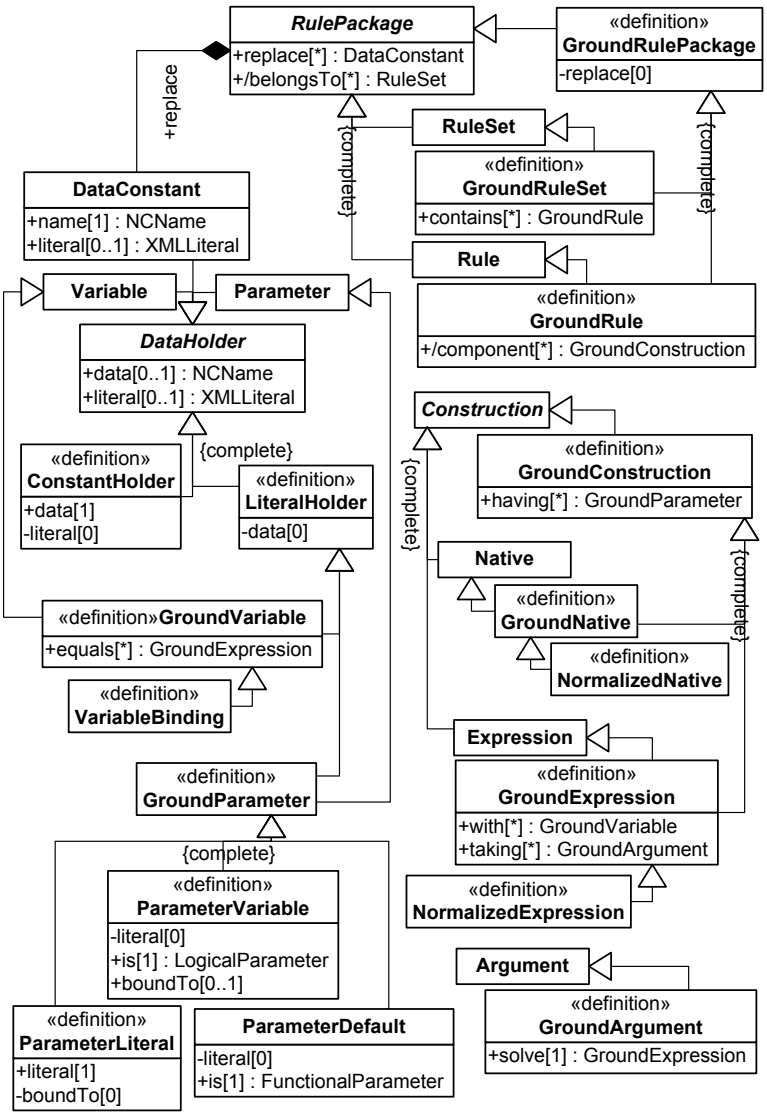

Figure 2.7: Ground Resources (v0.10)

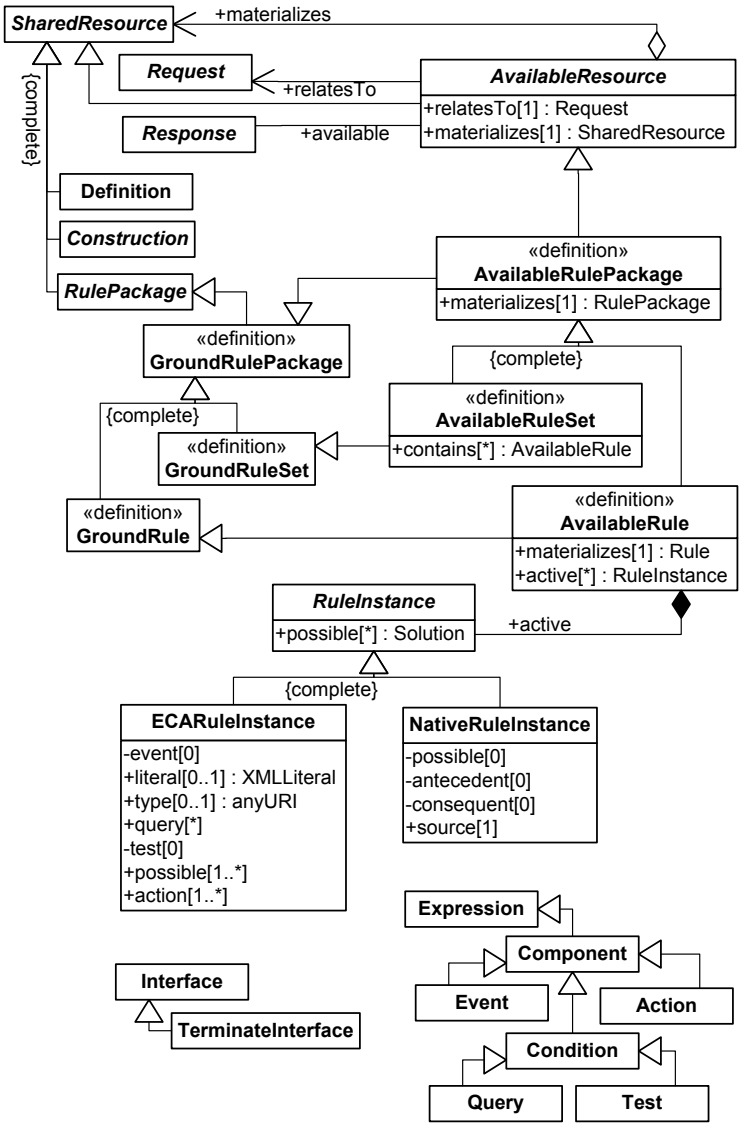

Figure 2.8: Available Resources (v0.12)

Chapter 3

XChange for the Eighteenth Century Studies Society

In this chapter we describe a demonstration of the XChange language in another of the scenarios presented before in our deliverable [5], namely the *Project's information system and portal*. Instead of implementing directly over the REWERSE project portal, we demonstrate on a fictitious scientific community of historians called the Eighteenth Century Studies Society (ECSS). Further details on the XChange implementation can be found wither from this working group's site (http://rewerse.net/I5/) or directly at http://reactiveweb.org/

3.1 XChange Language and Prototype

The reactive, rule-based language XChange [20] has been developed to respond to the need for both local (at a single Web node) and global (distributed over several Web nodes) evolution and reactivity on the Web, and has been extensively reported in previoues deliverables. Borrowing ideas from active database systems [39], XChange is a language of Event-Condition-Action (ECA) rules. XChange is tailored for the distributed nature of the Web and for common Web data formats by allowing event-based communication and by embedding the versatile Web query language Xcerpt [38, 37].

The demonstration described in this chapter shows how XChange can be applied to programming reactive Web sites where data evolves locally and, through mutual dependencies, globally. The setting we consider are several distributed Web sites of a fictitious scientific community of historians called the Eighteenth Century Studies Society (ECSS), itself an adaptation of the *"Project's information system and portal"* usecase scenario described in our deliverable [5]. As we shall see in more detail in section 3.2, ECSS is subdivided into participating universities, thematic working groups, and project management. Universities, working groups, and project management have each their own Web site, which is maintained and administered locally. The different Web sites are autonomous, but cooperate to evolve together and mirror relevant changes from other Web sites. For example, Web sites maintain information about personal data of members; a change of member data at a university entails further changes at the Web sites of the management and some working groups.

3.1.1 An overview of XChange

We start with a very short overview of the language XChange emphasizing general ideas especially those related to distributed programming with ECA rules. The demo itself will be described in the next section. The prototype of XcChange, which has been described in our previous deliverable, can be found online at FALTA.

XChange provides the following benefits over conventional approaches based on general-purpose programming languages to implement reactive behavior as needed in the demo:

(i) XChange reactive rules are highly declarative. They allow programming on a high abstraction level and are easy to analyze for both humans and machines.

(ii) The various parts of a rule all follow the same paradigm of specifying patterns for XML data, thus making XChange an elegant, easy to learn language.

(iii) Both atomic and composite events can be detected and relevant data extracted from events. Composite events, temporal combinations of events, are an important requirement in composing an application from different services.

(iv) XChange embeds an XML query language, Xcerpt, allowing to access and reason with Web data.

(v) XChange provides an integrated XML update language for modifying Web data.

(vi) XChange reactive rules enforce a clear separation of persistent data (Web resources) and volatile data (events). The distinction is important for programmers: the former relates to *state*, while the latter reflects *changes in state*.

(vii) XChange's high abstraction level and its powerful constructs allow for short and compact code.

We give in this section a very short introduction to the language XChange emphasizing general ideas especially those related to distributed programming with ECA rules. The demo itself will be described in the next section.

An XChange program is located at one Web site and consists of one or more (re)active rules of the form *Event query — Web query — Action*. Such an ECA rule has the following meaning: When events answering the event query are received and the Web query is successfully evaluated, the action is performed. Both event query and Web query can extract data through variable bindings, which can then be used in the action. With this, we can see that both event and Web queries serve a double purpose of detecting *when* to react and –through binding variables– *how* to react. For querying data, as well as for updating data, XChange embeds and extends the Web query language Xcerpt.

XChange programs at different Web sites can coordinate each other by sending and receiving events. Events are communicated in a push-manner as XML messages. Push communication has several advantages over pull communication: it allows faster reaction, avoids unnecessary network traffic through periodic polling, and saves local resources.

Example The rule in Figure 3.1 runs at the management's Web site and reacts to changes in the working group affiliation of a member. The event query detects `change-member` events, where the member is not part of the working group "Church and Reformation." The Web query then tests if this member has previously been a member of this working group. If this is the case, an event message is sent to the working group's Web site, requesting the member to be deleted.

We now take a closer look at the individual parts of XChange rules and explain the example in more detail, starting with how XML data of both events and Web resources is queried.

```
ON   xchange:event {{
       change-member {{
         memberId { var ID },
           newData {{
            workingGroups {{
              without wg {"Church and Reformation"}
     }}  }}  }}  }}
FROM in { resource {"http://ecss.org/members.xml"},
       members {{
         member {{
           id { var ID },
             desc wg {"Church and Reformation"}
     }} }}  }
DO   xchange:event {
       xchange:recipient { "http://churchreform.net" },
         delete-member {
           memberId { var ID }
     } }
END
```

Figure 3.1: ECA propagating the removal of a member from a working group

3.1.2 Queries as Patterns

Both event queries and Web queries are based on describing *patterns* for XML data. For conciseness, query patterns as well as construction patterns and update patterns are represented in a term-like syntax. In the term syntax, square brackets denote that the order of the children of an XML element is relevant, curly braces denote that the order is not relevant.

Both partial (i.e., incomplete) or total (i.e., complete) query patterns can be specified. A query term $\boldsymbol{t}$ using a partial specification (denoted by double brackets [[ ]] or braces **{{ }}**) for its subterms matches with all such terms that (1) contain matching subterms for all subterms of $\boldsymbol{t}$ and that (2) might contain further subterms without corresponding subterms in $\boldsymbol{t}$. In contrast, a query term $\boldsymbol{t}$ using a total specification (denoted by single brackets [ ] or braces **{ }**) does not match with terms that contain additional subterms without corresponding subterms in the query $\boldsymbol{t}$.

Query terms contain variables for selecting subterms of data terms that are bound to the variables. Accordingly, the result of a query are bindings for the free variables in that query.

More advanced constructs for describing query patterns are available, e.g., in our example we use `without` to query the *absence* of subterms and `desc` to query for subterms that are not immediate children of the parent term but *descendants* at arbitrary depth. Non-structural conditions (e.g., comparisons on integer variables) can be specified by adding a `where`-clause to queries.

3.1.3 Event Queries

Each Web site monitors the incoming event messages (XML representations of events) to check if they match an event query of one of its XChange rules. Atomic event queries are single query terms (as described above) and detect and react to single incoming event messages. In the example rule, we have an atomic event query which binds the variable $\boldsymbol{ID}$ to the content of the `memberId`-element.

Often, situations that require a reaction by a rule are not given by a single atomic event, but a temporal combination of events. For this, XChange supports composite event queries [19], which are built by combining event queries with composition operators like `andthen` (ordered

sequence of events), `and` (unordered conjunction of events), or `without` (absence of events) and temporal restrictions like `within` (all events happen within a given length of time).

3.1.4 Web Queries

The condition part of XChange rules queries data from Web resources such as XML documents or RDF documents. The keyword `in` together with `resource` is used to specify the URIs of the documents that are queried.

Queries can be combined into conjunctions (`and`) and disjunctions (`or`), thus allowing to access multiple documents and complex conditions. Also, negation (as failure) is supported (`not`).

3.1.5 Actions: Updates and Raising New Events

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

Raising New Events Events to be raised are specified as a construction pattern for the new event messages. Construction patterns (also called construct terms) are similar to query patterns; however only complete patterns (single brackets/braces) can be used. Variables are replaced by the bindings obtained previously in the event and Web queries.

Grouping and aggregation is supported through constructs like `all` ***ct*** `group-by var` ***X***, which will be "replaced" by one construction of ***ct*** for each binding of the variable ***X***.

Construction patterns for events must contain an element `xchange:recipient` which specifies the recipient Web node's URI. Note that this can be a variable.

Updates Updates to Web data are specified as so-called update terms. An update term is a (possibly incomplete) query pattern for the data to be updated, augmented with the desired update operations. An update term may contain different types of update operations: An insertion operation specifies a construct term that is to be inserted, a deletion operation specifies a query term for deleting all data terms matching it, and a replace operation specifies a query term to determine data items to be modified and a construct term for their new value.

Complex Actions Actions can be combined with disjunctions and conjunctions. Disjunctions specify alternatives, only one of the specified actions is to be performed successfully. (Note that actions such as updates can be unsuccessful, i.e., fail). Conjunctions in turn specify that all actions need to be performed. The combinations are indicated by the keywords `or` and `and`, followed by a list of the actions enclosed in braces or brackets.

3.2 Use-Case Scenario

The ECA rule language XChange was applied to the scenario constituted by the distributed Web data of ECSS. Data evolves locally and updates are propagated globally by means of event messages. The Web sites of universities, working groups, and management are autonomous, but cooperate to evolve together and mirror relevant changes from other Web sites.

The different Web sites maintain XML data about members, publications, meetings, library books, and newsletters. Data is often shared, for example a member's personal data is present at his home university, at the management node, and in the working groups he participates

***r*1**: `ON` change member
`DO` update LMU data
***r*2**: `ON` change member
`DO` forward to management
***r*3**: `ON` change member
`DO` update management data
***r*4**: `ON` change member (w/WG3)
`IF` was not member of WG3
`DO` send add member to WG3
***r*5**: `ON` change member (w/o WG2)
`IF` was member of WG2
`DO` send remove member to WG2
***r*6**: `ON` remove member
`DO` update WG2 data
***r*7**: `ON` add member
`DO` update WG3 data

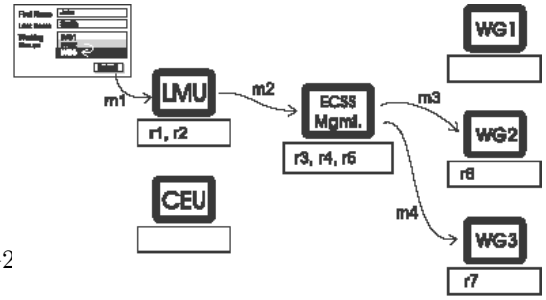

Figure 3.2: Changing a member's personal data (including working group affiliation)

in. Such shared data needs to be kept consistent among different nodes; this is realized by communicating changes as events between the different nodes using XChange ECA rules.

Events that occur in this community include changes in the personal data of members, keeping track of the inventory of the community-owned library, or simply announcing information from email newsletters to interested working groups. These events require reactions such as updates, deletion, alteration, or propagation of data, which are implemented using XChange rules. The rules run locally at the different Web nodes of the community, allowing for the processing of local and remote events.

For a concrete example, consider changing a member's personal data including his working group affiliation. The information flow is depicted in Figure 3.2. The initial change is entered by using a Web form at the member's home university LMU. The form generates event message ***m*1**. One ECA rule (***r*1**) reacts to this event and locally updates the member's data at LMU accordingly. Another ECA rule (***r*2**) forwards the change to the management node.

The management node has rules for updating its own local data about the member (***r*3**) and for propagating the change to the affected working groups (***r*4** for adding, ***r*5** for deleting a member). In the example, the member changes the working group affiliation from WG2 to WG3. Accordingly, event ***m*4** is sent to WG3 by rule ***r*4** and ***m*3** is sent to WG2 by ***r*5**.

The working groups finally each have rules reacting to deletion and insertion events (***m*2** and ***m*3**) to perform the requested updates (here: ***r*6** at WG2 and ***r*7** at WG3).

In this description we have restricted ourselves to this one example of changing member data. The implemented scenario realizes full member management of the community, a community-owned and distributed virtual library (e.g., lending books, monitions, reservations), meeting organization (e.g., scheduling panel moderators), and newsletter distribution. These other tasks are also implemented by ECA rules that are in place at the different nodes. For presentation purposes, the prototype includes facilities for displaying the rules of each node and logging received and sent events.

Bibliography

[1] José Júlio Alferes and Ricardo Amador. r^3: Towards a foundational ontology for reactive rules. Available at `http://rewerse.net/I5/r3/DOC/2006/submission.pdf`.

[2] José Júlio Alferes, Ricardo Amador, Erik Behrends, Mikael Berndtsson, François Bry, Gihan Dawelbait, Andreas Doms, Michael Eckert, Oliver Fritzen, Wolfgang May, Paula-Lavinia Pătrânjan, Loc Royer, Franz Schenk, and Michael Schroeder. Specification of a Model, Language and Architecture for Reactivity and Evolution. deliverable I5-D4, Centro de Inteligncia Artificial - CENTRIA, Universidade Nova de Lisboa, 2005.

[3] 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 behaviour at the XML-Level. deliverable I5-D5, Centro de Inteligência Artificial - CENTRIA, Universidade Nova de Lisboa, 2006.

[4] 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)*, volume 3703 of *Lecture Notes on Computer Science*, pages 101–115. Springer, 2005.

[5] José Júlio Alferes, Mikael Berndtsson, François Bry, Michael Eckert, Nicola Henze, Wolfgang May, Paula-Lavinia Pătrânjan, and Michael Schroeder. Use-cases on evolution. deliverable I5-D2, Centro de Inteligncia Artificial - CENTRIA, Universidade Nova de Lisboa, 2005.

[6] José Júlio Alferes and Gastn E. Tagni. Implementation of a Complex Event Engine for the Web. In *Proceedings of First International Workshop on Event-driven Architecture, Processing and Systems, Chicago, USA (18th September 2006)*, 2006.

[7] Ricardo Amador and José Júlio Alferes. Web Integrated Development tools for Evolution and Reactivity (WIDER). `http://www.ricardoamador.com/research/program.aspx`, 2005. PhD Proposal.

[8] Jürgen Angele, Harold Boley, Jos de Bruijn, Dieter Fensel, Pascal Hitzler, Michael Kifer, Reto Krummenacher, Holger Lausen, Axel Polleres, and Rudi Studer. Web Rule Language (WRL). `http://www.w3.org/Submission/2005/SUBM-WRL-20050909/`.

[9] Steve Battle, Abraham Bernstein, Harold Boley, Benjamin Grosof, Michael Gruninger, Richard Hull, Michael Kifer, David Martin, Sheila McIlraith, Deborah McGuinness,

Jianwen Su, and Said Tabet. Semantic Web Services Language (SWSL) 1.0. `http://www.daml.org/services/swsf/1.0/swsl/`.

[10] Steve Battle, Abraham Bernstein, Harold Boley, Benjamin Grosof, Michael Gruninger, Richard Hull, Michael Kifer, David Martin, Sheila McIlraith, Deborah McGuinness, Jianwen Su, and Said Tabet. Semantic Web Services Language (SWSL) 1.1. `http://www.daml.org/services/swsf/1.1/swsl/`.

[11] Erik Behrends, Oliver Fritzen, Wolfgang May, and Franz Schenk. Combining ECA Rules with Process Algebras for the Semantic Web. In *Proceedings of Second International Conference on Rules and Rule Markup Languages for the Semantic Web, Athens, Georgia, USA (10th–11th November 2006)*, 2006.

[12] Erik Behrends, Oliver Fritzen, Wolfgang May, and Daniel Schubert. An ECA Engine for Deploying Heterogeneous Component Languages in the Semantic Web. In *Proceedings of Workshop Reactivity on the Web, Munich, Germany (31st March 2006)*, LNCS, 2006.

[13] Tim Berners-Lee. Notation 3 (N3). `http://www.w3.org/DesignIssues/Notation3`.

[14] Tim Berners-Lee. Semantic Web - talk. `http://www.w3.org/2000/Talks/1206-xml2k-tbl/`, December 2000.

[15] Tim Berners-Lee. Putting the Web back in Semantic Web - talk. `http://www.w3.org/2005/Talks/1110-iswc-tbl/`, November 2005.

[16] Harold Boley, Benjamin Grosof, Michael Sintek, Said Tabet, and Gerd Wagner. *RuleML Design*. RuleML Initiative, http://www.ruleml.org/, 2002.

[17] F. Bry and P.-L. Pătrânjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. In *20th ACM Symp. Applied Computing*. ACM, 2005.

[18] François Bry and Michael Eckert. A High-Level Query Language for Events. In *Proceedings of First International Workshop on Event-driven Architecture, Processing and Systems, Chicago, USA (18th September 2006)*, 2006.

[19] François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan. Querying composite events for reactivity on the Web. In *Proc. Intl. Workshop on XML Research and Applications*, number 3842 in LNCS, pages 38–47. Springer, 2006.

[20] François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan. Reactivity on the Web: Paradigms and applications of the language XChange. *Journal of Web Engineering*, 5(1):3–24, 2006.

[21] François Bry, Michael Eckert, Paula-Lavinia Pătrânjan, and Inna Romanenko. Realizing Business Processes with ECA Rules: Benefits, Challenges, Limits. In *Proceedings of 4th Workshop on Principles and Practice of Semantic Web Reasoning, Budva, Montenegro (10th–11th June 2006)*, LNCS. REWERSE, 2006.

[22] Sharma Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering*, 14(1):1–26, 1994.

[23] Jos de Bruijn, Holger Lausen, Axel Polleres, and Dieter Fensel. The WSML rule languages for the Semantic Web. In *W3C Workshop on Rule Languages for Interoperability, 27-28 April 2005, Washington, D.C., USA*, 2005.

[24] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[25] Foundation for Intelligent Physical Agents. FIPA ACL message structure specification. Technical Report SC00061G, http://www.fipa.org, Dec. 2002.

[26] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. `http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/`.

[27] Aditya Kalyanpur, Daniel Pastor, Steve Battle, and Julian Padget. Automatic Mapping of OWL Ontologies into Java. In *Proceedings of Sixteenth International Conference on Software Engineering and Knowledge Engineering (SEKE), Banff, Canada, (20th–24th June 2004)*, 2004.

[28] Alexander Kozlenkov and Michael Schroeder. PROVA: Rule-based Java-Scripting for a Bioinformatics Semantic Web. In *Proceedings of First International Workshop, DILS 2004, Leipzig, Germany (March 25-26 2004)*, LNCS, 2004.

[29] Wolfgang May, José Júlio Alferes, and Ricardo Amador. Active rules in the Semantic Web: Dealing with language heterogeneity. In *International Conference on Rules and Rule Markup Languages for the Semantic Web (RuleML)*, volume 3791 of *Lecture Notes on Computer Science*, pages 30–44. Springer, 2005.

[30] 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 Applications of Semantics (ODBASE)*, volume 3761 of *Lecture Notes on Computer Science*, pages 1553–1570. Springer, 2005.

[31] Wolfgang May, José Júlio Alferes, and François Bry. Towards generic query, update, and event languages for the Semantic Web. In *Principles and Practice of Semantic Web Reasoning (PPSWR)*, volume 3208 of *Lecture Notes on Computer Science*, pages 19–33. Springer, 2004.

[32] Wolfgang May, Franz Schenk, and Elke von Lienen. Extending an OWL Web Node with Reactive Behavior. In *Proceedings of 4th Workshop on Principles and Practice of Semantic Web Reasoning, Budva, Montenegro (10th–11th June 2006)*, LNCS. REWERSE, 2006.

[33] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[34] Object Management Group. *Unified Modelling Language (UML) 2.0 Superstructure*. OMG, 2004. http://www.omg.org/cgi-bin/doc?ptc/2004-10-02.

[35] Object Management Group. *Semantics of Business Vocabulary and Business Rules (SBVR)*. OMG, 2006. http://www.omg.org/cgi-bin/doc?dtc/2006-03-02.

[36] George Papamarkos, Alexandra Poulovassilis, and Peter T. Wood. Event-condition-action rules on RDF metadata in P2P environments. *Elsevier Computer Networks journal*, October 2006. To appear.

[37] S. Schaffert and F. Bry. A practical introduction to Xcerpt. In *Int. Conf. Extreme Markup Languages*, 2004.

[38] Sebastian Schaffert and François Bry. Querying the Web reconsidered: A practical introduction to Xcerpt. In *Proc. of Extreme Markup Languages Conf.*, 2004.

[39] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, San Francisco, CA, USA, 1996.

[40] Apache Axis. `http://ws.apache.org/axis/`.

[41] Apache Axis2. `http://ws.apache.org/axis2/`.

[42] DB2 universal database. `http://www.ibm.com/db2/`.

[43] The Glasgow Haskell Compiler. `http://haskell.org/ghc/`.

[44] Haskell - A purely functional language. `http://haskell.org/`.

[45] Jastor - Typesafe, Ontology Driven RDF Access from Java. `http://jastor.sourceforge.net/`.

[46] JavaServer Pages Technology. `http://java.sun.com/products/jsp`.

[47] Java Servlet Technology. `http://java.sun.com/products/servlets`.

[48] Jena Semantic Web framework for Java. `http://jena.sourceforge.net/`.

[49] Java message service (JMS). `http://java.sun.com/products/jms/`.

[50] Web ontology language (OWL). `http://www.w3.org/2004/OWL/`.

[51] Pellet OWL Reasoner. `http://www.mindswap.org/2003/pellet/`.

[52] The REWERSE I1 Rule Markup Language. `http://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=node/6`.

[53] Resourceful Reactive Rules (r^3). `http://rewerse.net/I5/r3/`. Centro de Inteligência Artificial - CENTRIA, Universidade Nova de Lisboa.

[54] Resource description framework (RDF). `http://www.w3.org/RDF/`.

[55] RDF/XML syntax specification (revised). `http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/`.

[56] Rule Interchange Format (RIF) W3C Working Group. `http://www.w3.org/2005/rules/`.

[57] SAXON - The XSLT and XQuery Processor. `http://saxon.sourceforge.net/`.

[58] Simple object access protocol (SOAP). `http://www.w3.org/TR/soap`.

[59] SweetRules: Tools for Semantic Web rules and ontologies. http://sweetrules.projects.semwebcentral.org/, 2006.

[60] Web services addressing (WS-Addressing). http://www.w3.org/Submission/ws-addressing/.

[61] Web services description language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[62] XML Schema. http://www.w3.org/XML/Schema/.

[63] W3C XML query (XQuery). http://www.w3.org/XML/Query/.

[64] XQuery 1.0 and XPath 2.0 Functions and Operators. http://www.w3.org/TR/xpath-functions/.

[65] XSL transformations (XSLT) 2.0. http://www.w3.org/TR/xslt20/.

Acknowledgements

This research has been co-funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

Appendix A

Component Engines. Examples and implementation in r^3

The Java excerpts here presented are based on version 0.12-rc and are bound to become outdated very shortly. Nevertheless together with the included XML examples they may help the interested reader to get a better insight on the expected features for the r^3 library, for which full documentation is to be postponed until a more stable version is reached.

A.1 HTTP Engine

Java Excerpts:

```
package net.rewerse.i5.r3.eval.http;
  ...
public class Evaluator extends ExprEvaluator {
  ...
protected void evaluate(Context ctx) throws Exception {
  String op = ctx.opname();

  boolean usevars = op.startsWith("v");
  String puri = usevars ? "vuri" : "uri";
  HTTPRequest req = new HTTPRequest(
    ctx.text("method"), ctx.text(puri), ctx.text("body"));
  String[] phs = null;
  boolean trstatus;
  if (op.equals("opaque")) {
    usevars = true;
    trstatus = false;
    req.parse(ctx.text("literal"));
  } else { // op.equals("get/post/put/delete/transform")
    if (usevars) op = op.substring(1);
    trstatus = true;
    if (op.equals("transform")) {
      if (!req.method.equals("POST") && !req.method.equals("PUT"))
        throw new Exception(
          "Operator http:transform requires http:method PUT or POST");
      throw new Exception("Operator http:[v]transform not implemented");
      // Naive view: req.body = ctx.opcomposite().arg-eval("source");
      // TODO: implement AlgebraContext with multiple results,
      //     each with its set of ?distinct? tuples to join
    } else {
      req.method = op.toUpperCase();
    }
    phs = new String[] {
      "soapaction", "content-type", "content-length"};
    for (String hn: phs) {
      String hv = ctx.text(hn);
      if (hv != null) req.headers.add(hn+": "+hv);
    }
  }

  URI u = new URI(req.uri);
  String bqry = u.getRawQuery(), frag = u.getRawFragment();
  for (Tuple t: ctx.getUsing()) {
    String qry = bqry, charset = ctx.text("uri-query-values-encoding");
    ArrayList<String> headers = new ArrayList<String>(req.headers);
    Hashtable<String,String> vhs = new Hashtable<String,String>();

    if (usevars) {
      for (Variable v : t.getBinding()) {
        String n = v.getName(), r = ctx.rename(n);
        char c = r.charAt(0);
        if (c != ':' && c != '=') continue;
        r = r.substring(1);
        String txt = v.getLiteral();
        if (c == ':') {
          if (txt != null) headers.add(r+": "+txt);
          vhs.put(n, r.toLowerCase());
        } else {
          if (txt != null) txt = URLEncoder.encode(txt, charset);
          qry = r+(txt==null ? "" : ("="+txt))+
                  (qry==null ? "" : ("&"+qry));
        }
      }
    }
    u = new URI(u.getScheme(), u.getUserInfo(),
                u.getHost(), u.getPort(), u.getPath(), null, null);
    String uri = u.toURL().toString();
    if (qry != null) uri += "?"+qry;
    if (frag != null) uri += "#"+frag;
    R3Utils.URLStream us = new R3Utils.URLStream(
      req.method, uri, headers, req.body, trstatus, true);
    if (trstatus && !us.isok()) {
      us.close();
      if (us.isnotfound()) {
        if (usevars) continue; else break; // fail
      }
      throw new Exception(
        "Unexpected HTTP status "+us.status+ " "+us.reason);
    }
    ctx.startResult(usevars ? t : null);
    if (usevars) {
      for (String vn: vhs.keySet()) {
        String h = us.headers.get(vhs.get(vn));
        if (h != null) ctx.addBinding(vn, h);
      }
    }
    if (phs != null) {
      for (String hn: phs) {
        String hv = us.headers.get(hn);
        if (hv != null) ctx.output(hn, hv);
      }
    }
    if (!trstatus)
      ctx.output("status-class", ""+((int)(us.status/100)));
    ctx.output("status", ""+us.status);
    ctx.output("status-reason", us.reason);
    ctx.finishResult(us.asString(true).trim());
    if (!usevars) break;
  }
}
```

```
  ...
}
```

Examples:

```
<Evaluate
  xmlns="http://rewerse.net/I5/NS/r3/2005#"
  xml:base="http://rewerse.net/I5/NS/r3/2005/eval/http"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <issuer><Client/></issuer>
    <solve><Expression><is rdf:resource="#opaque"/>
        <having><Parameter>
          <is rdf:resource="#status"/>
          <boundTo>Status</boundTo>
        </Parameter></having>
        <having><Parameter><is rdf:resource="#status-class"/>
          <boundTo>StatusClass</boundTo>
        </Parameter></having>
        <having><Parameter><is rdf:resource="#status-reason"/>
          <boundTo>StatusReason</boundTo>
        </Parameter></having>
        <having><Parameter><is rdf:resource="#literal"/>
          <literal rdf:parseType="Literal">
POST http://di150.di.fct.unl.pt:15080/r3/service/prova HTTP/1.1
content-type: text/xml

<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Body>
<request xmlns=""
    xmlns:log="http://www.semwebtech.org/lang/2006/logic#">
  <subject>http://dummy.nop/ddd/solve</subject>
  <opaque>a(X,Y)</opaque>
  <log:variable-bindings>
    <log:tuple>
      <log:variable name="X" />
      <log:variable name="Y" />
    </log:tuple>
  </log:variable-bindings>
</request>
</soapenv:Body>
</soapenv:Envelope>
          </literal>
        </Parameter></having>
    </Expression></solve>
    <using><Substitution>
        <binding><Variable>
          <name>SOAPAction</name><rename>:SOAPAction</rename>
          <literal rdf:parseType="Literal"
          >"http://rewerse.net/I5/NS/r3/2005#Evaluate"</literal>
        </Variable></binding>
        <binding><Variable>
          <name>Type</name><rename>:Content-Type</rename>
        </Variable></binding>
        <binding><Variable>
          <name>Length</name><rename>:Content-Length</rename>
        </Variable></binding>
        <binding><Variable><name>Status</name></Variable></binding>
        <binding><Variable><name>StatusClass</name></Variable></binding>
        <binding><Variable><name>StatusReason</name></Variable></binding>
    </Substitution></using>
</Evaluate>
```

A.2 Prova Engine

Java Excerpts:

```
package net.rewerse.i5.r3.eval.prova;
  ...
public class Evaluator extends NativeEvaluator {
  ...
protected void evaluate(Context ctx) throws Exception {
  String rules = normalize(ctx.text("rulesdb"));
  String opn = ctx.opname();
  if (opn.equals("opaque")) {
    solve(ctx, rules, ctx.text("literal"));
  } else if (opn.equals("sendMsg")) {
    sendMsg(ctx, rules);
  } else { // opn.equals("rcvMsg")
    addRcvMsg(ctx, rules);
  }
}

protected void solve(Context ctx, String rules, String goal)
  throws Exception
{
  // collect variables with a bad name
  ArrayList<String> badv = new ArrayList<String>();
  for (String n : ctx.vars())
    if (!Character.isUpperCase(ctx.rename(n).charAt(0))) badv.add(n);
  // if there are any bad variables
  if (!badv.isEmpty()) {
    // compute a conflict free prefix
    String vpref = varPrefix(ctx, goal);
    // and rename those bad variables
    int vind = 0;
    for (String bn : badv) ctx.rename(bn, vpref+(vind++));
  }

  // build list of goals to solve
  String lgoals = "";
  ArrayList<Object> lobjs = new ArrayList<Object>();
  for (Tuple t : ctx.getUsing()) {
    ArrayList<String> vns = new ArrayList<String>();
    ArrayList<String> vvs = new ArrayList<String>();
    for (Variable v : t.getBinding()) {
      String n = v.getName(), r = ctx.rename(n);
      Object o = asObject(v.getLiteral());
      if (o != null) {
        vns.add(r);
        vvs.add("_"+lobjs.size());
        lobjs.add(o);
      }
    }
    lgoals += ":- solve(ctxderive("+vns.toString()+", "+
        vvs.toString()+", "+goal+")).\n";
  }

  // solve goals
  List lrs = null;
  Communicator sh = provaShell(rules);
  synchronized (sh) {
    sh.consultSync(
        "ctxderive(X,X,[P|As]) :- derive([P|As]).\n", "ctxderive", null);
    lrs = sh.consultSync(
        lgoals.toString(), "ctxderive", lobjs.toArray());
    sh.unconsultSync("ctxderive");
  }

  // collect bindings
  for( Iterator itrs = lrs.iterator(); itrs.hasNext(); ) {
    ProvaResultSet rs = (ProvaResultSet)itrs.next();
    Exception ex = rs.getException();
    if (ex != null) throw ex;
    for( Iterator itr = rs.iterator(); itr.hasNext(); ) {
      org.mandarax.kernel.Result r =
          (org.mandarax.kernel.Result)itr.next();
      ctx.startResult();
      Map m = r.getResults();
      Set ks = m.keySet();
      for (Object ko : ks) {
        VariableTerm vk = (VariableTerm)ko;
        Object vo = m.get(ko);
        ConstantTerm c = (ConstantTerm)vo;
        String vn = ctx.var(vk.getName());
        if (vn == null || vn.startsWith("_"))
          continue; // skip new or unnamed vars
        ctx.addBinding(vn, asString(c.getObject()));
      }
      ctx.finishResult("");
    }
  }
}
  ...
protected void createNative(Context ctx, String id)
  throws Exception
{
  String rules = normalize(ctx.text("rulesdb"));
  provaShell(rules).consultSync(ctx.text("literal"), id, null);
}

protected void freeId(Context ctx, String id) throws Exception {
  String rules = normalize(ctx.text("rulesdb"));
  boolean create = false;
  provaShell(rules, create).unconsultSync(id);
}
  ...
}
```

Examples:

```
<register>
  <subject>http://dummy.nop/ddd/1</subject>
  <opaque>
  a('1',bbb).
  a(2,aaa).
  a(3,bbb).
```

```
  </opaque>
</register>

<request xmlns:log="http://www.semwebtech.org/lang/2006/logic#">
  <subject>http://dummy.nop/ddd/solve</subject>
  <opaque>a(X,Y)</opaque>
  <log:variable-bindings>
    <log:tuple>
      <log:variable name="X" />
      <log:variable name="Y" />
    </log:tuple>
  </log:variable-bindings>
</request>

<deregister>
  <subject>http://dummy.nop/ddd/1</subject>
</deregister>

<request xmlns:log="http://www.semwebtech.org/lang/2006/logic#">
  <subject>http://dummy.nop/ddd/solve</subject>
  <Expression
    xmlns="http://rewerse.net/I5/NS/r3/2005#"
    xml:base="http://rewerse.net/I5/NS/r3/2005/eval/prova"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <is rdf:resource="#opaque" />
    <having><Parameter>
      <is rdf:resource="#rulesdb" />
      <literal>http://rewerse.net/I5/r3/TST/templates.prova</literal>
    </Parameter></having>
    <having><Parameter>
      <is rdf:resource="#literal" />
      <literal>a(X)</literal>
    </Parameter></having>
  </Expression>
  <log:variable-bindings>
    <log:tuple>
      <log:variable name="X" />
    </log:tuple>
  </log:variable-bindings>
</request>
```

A.3 Xcerpt Engine

Java Excerpts:

```
package net.rewerse.i5.r3.eval.xcerpt;
  ...
public class Evaluator extends XcerptEvaluator {
  ...
protected void evaluate(Context ctx) throws Exception {
  String opn = ctx.opname();
  if (opn.equals("program")) {
    // program(document)
    ctx.finishResult(xcerptClient().convertProgramXML2Xcerpt(
        ctx.literal("document")));
    return;
  } else if (opn.equals("term")) {
    // term(document)
    ctx.finishResult(xcerptClient().convertTermXML2Xcerpt(
        ctx.literal("document")));
    return;
  } else if (opn.equals("eval")) {
    // eval(?construct)+aggregate+groupedBy
    throw new Exception("xcerpt:eval aggregator not implemented");
  } else if (opn.equals("transform")) {
    // transform(?construct, match)+takes(source)
    throw new Exception("xcerpt:transform operator not implemented");
  } else { // opn.equals("opaque")
    // opaque(?rulesdb, ?construct, literal)
    solve(ctx, ctx.text("rulesdb"));
  }
}

void solve(Context ctx, String rules) throws Exception {
  XcerptClient cli = xcerptClient();
  rules = getRules(cli, rules);

  String tupleset = "tupleset";
  while (rules.contains(tupleset)) tupleset += "_";

  String lresult = "result";
  ArrayList<String> tpl = new ArrayList<String>();
  for (String n : ctx.vars()) {
    String r = ctx.rename(n);
    if (lresult.equals(r)) lresult += "_";
    tpl.add("optional "+r+" { var "+r+" }");
  }

  String frm = ""
    + "    "+tupleset+" {{ tuple {\n"
    + "        "+join(tpl, ",\n        ")+"\n"
    + "    } }}";
  String query = notEmpty(ctx.text("literal"));
  if (query != null) {
    query = "  and {\n"+frm+",\n\n    "+query.trim()+"\n  }";
  } else {
    query = frm;
  }

  String construct = notEmpty(ctx.text("construct"));
  if (construct != null)
    construct = lresult+" { "+construct+" }";
  construct = ""
    + "  out { resource { \"stdout:xml\" },\n"
    + "    "+tupleset+" { all tuple {\n"
    + "        "+join(construct, tpl, ",\n        ")+"\n"
    + "    } }"+"\n"
    + "  }";

  String goal = "\nGOAL\n"+construct+"\nFROM\n"+query+"\nEND\n";
  String init = "\n"
    + "CONSTRUCT\n"
    + tupleset+" ["+join(getTuples(cli, ctx), ", ")+"\n"
    + "]\n"
    + "END\n";

  String res = cli.evaluateProgram(goal+init+"\n"+rules);
  if (!res.trim().equals(
      "<xcerpt:error xmlns:xcerpt=\"http://xcerpt.org\">"+
      "no results"+
      "</xcerpt:error>"))
  {
    buildResult(res, lresult, ctx);
  }
}


package net.rewerse.i5.r3.eval.xcerpt;
  ...
public class XcerptEvaluator extends NativeEvaluator {
  ...
protected Collection<String> getTuples(
  XcerptClient cli, Context ctx) throws Exception
{
  ArrayList<String> tsl = new ArrayList<String>();
  ArrayList<String> tl = new ArrayList<String>();
  for (Tuple t : ctx.getUsing()) {
    for (Variable v : t.getBinding()) {
      String n = v.getName(), r = ctx.rename(n);
      String vs = v.getLiteral();
      if (vs != null) {
        if (vs.trim().length() == 0)
          vs = r+" [ \"\" ]";
        else
          vs = cli.convertTermXML2Xcerpt("<"+r+">"+vs+"</"+r+">");
        tl.add("\n        "+vs);
      }
    }
    tsl.add("\n    tuple ["+join(tl, ", ")+"\n    ]");
    tl.clear();
  }
  return tsl;
}

protected void buildResult(
  String res, String lresult, Context ctx) throws Exception
{
  Document tmpd = xmlBuilder().parse(
      new ByteArrayInputStream(res.getBytes()));
  Element el = tmpd.getDocumentElement();
  NodeList lr = el.getChildNodes();
  for ( int i=0, szr=lr.getLength(); i<szr; i++ ) {
    Node nr = lr.item( i );
    if ( nr.getNodeType() != Node.ELEMENT_NODE ) continue;
    ctx.startResult();
    String r = null;
    NodeList lt = nr.getChildNodes();
    for ( int j=0, szt=lt.getLength(); j<szt; j++ ) {
      Node nt = lt.item(j);
      if ( nt.getNodeType() != Node.ELEMENT_NODE ) continue;
      String nm = nt.getLocalName();
      String vl = asString(nt, true).trim();
      if (nm.equals(lresult)) {
        r = vl;
      } else {
        nm = ctx.var(nm);
        if (nm != null) ctx.addBinding(nm, vl);
      }
    }
    ctx.finishResult(r);
  }
}
```

```
  ...
}
```

Examples:

```
<register>
  <subject>http://dummy.nop/xcerpt/bib</subject>
  <opaque>
CONSTRUCT
  bib [
    book [
      title [ "Dummy33" ],
      price [ "33" ]
    ],
    book [
      title [ "Dummy44" ],
      price [ "44" ]
    ],
    book [
      title [ "Dummy55" ],
      price [ "55" ]
    ]
  ]
END
  </opaque>
</register>

<register>
  <subject>http://dummy.nop/xcerpt/rev</subject>
  <opaque>
CONSTRUCT
  reviews [
    entry [
      title [ "Dummy33" ],
      price [ "66" ]
    ]
  ]
END
  </opaque>
</register>

<request xmlns:log="http://www.semwebtech.org/lang/2006/logic#">
  <subject>http://dummy.nop/xcerpt?ex=00</subject>
  <opaque>
    and {
      bib {{
        book {{
          title { var T  },
          price { var Pa }
        }}
      }}
      ,
      reviews {{
        entry {{
          title { var T },
          price { var Pb }
        }}
      }}
    }
  </opaque>
  <log:variable-bindings>
    <log:tuple>
      <log:variable name="T" />
      <log:variable name="Pa" />
      <log:variable name="Pb" />
    </log:tuple>
  </log:variable-bindings>
</request>

<deregister>
  <subject>http://dummy.nop/xcerpt/bib</subject>
</deregister>

<deregister>
  <subject>http://dummy.nop/xcerpt/rev</subject>
</deregister>

<request xmlns:log="http://www.semwebtech.org/lang/2006/logic#">
  <subject>http://dummy.nop/xcerpt?ex=01</subject>
  <opaque>
    in {
      resource {"file:bib.xml"},
      bib {{
        book {{
          title { var T  },
          price { var Pa }
        }}
      }}
    }
  </opaque>
  <log:variable-bindings>
    <log:tuple>
      <log:variable name="T" />
      <log:variable name="Pa" />
    </log:tuple>
  </log:variable-bindings>
</request>

<request xmlns:log="http://www.semwebtech.org/lang/2006/logic#">
  <subject>http://dummy.nop/xcerpt?ex=4</subject>
  <Expression
    xmlns="http://rewerse.net/I5/NS/r3/2005#"
    xml:base="http://rewerse.net/I5/NS/r3/2005/eval/xcerpt"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <is rdf:resource="#opaque" />
    <having><Parameter>
      <is rdf:resource="#rulesdb" />
      <literal>http://rewerse.net/I5/r3/TST/templates.xcerpt</literal>
    </Parameter></having>
    <having><Parameter>
      <is rdf:resource="#construct" />
      <literal>
  book [
    title   [ var Title  ],
    price-a [ var PriceA ],
    optional price-b [ var PriceB ]
  ]
      </literal>
    </Parameter></having>
    <having><Parameter>
      <is rdf:resource="#literal" />
      <literal>
  or {
      bib {{
        book {{
          title { var Title  },
          price { var PriceA }
        }}
      }}
      ,
    books-with-prices [[
      book-with-prices [
        title   [ var Title  ],
        price-a [ var PriceA ],
        price-b [ var PriceB ]
      ]
    ]]
  }
      </literal>
    </Parameter></having>
  </Expression>
  <log:variable-bindings>
    <log:tuple>
      <log:variable name="Title" />
      <log:variable name="PriceA" />
      <log:variable name="PriceB" />
    </log:tuple>
  </log:variable-bindings>
</request>

<request xmlns:log="http://www.semwebtech.org/lang/2006/logic#">
  <subject>http://dummy.nop/xcerpt?ex=5</subject>
  <opaque>
    in {
      resource {"http://rewerse.net/I5/r3/TST/templates.xml"},
      templates {{
       group {{
        attributes {{ name { var G } }},
        description { var GD },
        template {{
          attributes {{ name { var T } }},
          description { var TD }
        }}
       }}
      }}
    }
  </opaque>
  <log:variable-bindings>
    <log:tuple>
      <log:variable name="G" />
      <log:variable name="T" />
    </log:tuple>
  </log:variable-bindings>
</request>
```

A.4 XChange Engine

Java Excerpts:

```
package net.rewerse.i5.r3.eval.xchange;
```

```
  ...
public class Evaluator extends XcerptEvaluator
  implements XChangeListener
{
  ...
protected void evaluate(Context ctx) throws Exception {
  XChangeFullClient cli = xchangeClient();

  String detect = null, action = null;
  String opn = ctx.opname();
  if (opn.equals("detect")) {
    detect = notEmpty(ctx.text("query"));
  } else if (opn.equals("execute")) {
    action = notEmpty(ctx.text("transaction"));
  } else if (opn.equals("raise")) {
    String recipient = notEmpty(ctx.text("recipient"));
    action = cli.buildEventRaise(recipient, notEmpty(ctx.text("event")));
  } else { // opn.equals("opaque")
    String txt = notEmpty(ctx.text("literal"));
    if (txt == null)
      throw new Exception("xchange:literal cannot be empty");
    if (txt.endsWith("END")) {
      int nl = txt.length()-"END".length();
      txt = txt.substring(0, nl).trim();
      if (txt.length() == nl)
        // no space before END
        txt = null;
    } else {
      txt = null;
    }
    if (txt == null)
      throw new Exception("xchange:literal must terminate with 'END'");
    boolean isqry = false;
    String pref = null;
    if (txt.startsWith("ON")) {
      isqry = true;
      pref = "ON";
    } else if (txt.startsWith("TRANSACTION")) {
      pref = "TRANSACTION";
    } else if (txt.startsWith("RAISE")) {
      pref = "RAISE";
    }
    if (pref != null) {
      int nl = txt.length()-pref.length();
      txt = notEmpty(txt.substring(pref.length()));
      if (txt == null || txt.length() == nl)
        // txt empty || no space after pref
        pref = null;
    }
    if (pref == null)
      throw new Exception(
        "xchange:literal must start with 'ON'/'RAISE'/'TRANSACTION'");
    if (isqry) detect = txt; else action = txt;
  }

  String tupleset = "tupleset";
  String init = tupleset+" ["+
      join(getTuples(cli.xcerptClient(), ctx), ", ")+"\n]";
  ArrayList<String> tpl = new ArrayList<String>();
  String lresult = "result";
  for (String n : ctx.vars()) {
    String r = ctx.rename(n);
    tpl.add("optional \""+r+"\" { var "+r+" }");
    if (lresult.equals(r)) lresult += "_";
  }
  String tuple = ""
    +"        "+join(tpl, ",\n        ")+"\n";
  String from = ""
    +"    "+tupleset+" {{ tuple {\n"
    +tuple
    +"    } }}";
  tuple = ""
    +"    tuple {\n"
    +tuple
    +"    }";

  String iid = ctx.incomplete();
  String seed = cli.proxyEventMatch("id { \""+iid+"\" }, "+from);
  String seedRaise = "id { \""+iid+"\" }, "+init;
  ArrayList<String> rules = new ArrayList<String>();
  if (detect != null) {
    rules.add(""
        +"RAISE\n"
        +cli.proxyEventRaise("event { "+
            "attributes { id { \""+iid+"\" } }, "+
            "result { var "+lresult+" }, "+tuple+" }")+"\n"
        +"ON\n"
        +"andthen [\n"
        +seed+",\n"
        +"var "+lresult+" ->\n"
        +detect+"\n"
        +"]\n"
        +"END\n");
  } else { // action != null
    rules.add(""
        +"TRANSACTION\n"
        +"or [\n"
        +"  and [\n"
        +"    "+action+",\n"
        +cli.proxyEventRaise("done { "+
            "attributes { id { \""+iid+"\" } }, "+
            "result { \"\" }, "+tuple+" }")+"\n"
        +"  ],\n"
        +cli.proxyEventRaise("error { "+
            "attributes { id { \""+iid+"\" } } }")+"\n"
        +"]\n"
        +"ON\n"+seed+"\n"
        +"END\n");
  }
  cli.registerRules(iid, rules, seedRaise);
}

protected void createNative(Context ctx, String id)
  throws Exception
{
  xchangeClient().registerRule(id, ctx.text("literal"));
}

protected void freeId(Context ctx, String id) throws Exception {
  xchangeClient().freeRules(id);
}

public void received(String msg) {
  // public to be called by the xchangeClient() proxy
  try {
    Element el = getReceivedEvent(msg);
    if (el == null) return;
    String iid = el.getAttribute("id"), tp = el.getLocalName();
    boolean islast = !tp.equals("event");
    if (islast) xchangeClient().freeRules(iid);
    NodeList l = el.getElementsByTagName("result");
    int sz = l.getLength();
    if (sz > 1) return;
    Element result = sz == 1 ? (Element)l.item(0) : null;
    l = el.getElementsByTagName("tuple");
    sz = l.getLength();
    if (sz > 1) return;
    Element tuple = sz == 1 ? (Element)l.item(0) : null;
    if (result == null && tuple != null) return;
    if (tuple == null && result != null) return;
    Context ctx = incompleteEvaluation(iid);
    if (tuple != null) {
      ctx.startResult();
      try {
        l = tuple.getChildNodes();
        sz = l.getLength();
        for (int i=0; i<sz; i++) {
          Node nd = l.item(i);
          if ( nd.getNodeType() != Node.ELEMENT_NODE ) continue;
          String nm = nd.getLocalName();
          nm = ctx.var(nm);
          if (nm != null)
            ctx.addBinding(nm, asString(nd, true).trim());
        }
      } catch (Exception ex) {
        ctx.cancelResult();
        throw new Exception("Result aborted", ex);
      }
      ctx.finishResult(asString(result, true).trim());
    }
    ctx.notifyResults(iid, islast);
  } catch (Exception e) {}
}
  ...
}
```

Examples:

```
<register>
  <subject>http://dummy.nop/xchange/1</subject>
  <opaque>
RAISE
  "xchange":event {{
    "xchange":recipient { "http://localhost:4711" },
    blablub { got { var X }, "in" { var Y } }
  }}
ON
  var Y -> "xchange":event {{
    blablub { var X }
  }}
END
  </opaque>
</register>
```

```
<deregister>
  <subject>http://dummy.nop/xchange/1</subject>
</deregister>

<Evaluate
  xmlns="http://rewerse.net/I5/NS/r3/2005#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xml:base="http://rewerse.net/I5/NS/r3/2005/eval/xchange"
  rdf:about="http://dummy.nop/xchange?id=detect">
  <solve><Expression><is rdf:resource="#detect" />
    <having><Parameter><is rdf:resource="#query" />
      <literal rdf:parseType="Literal">
"xchange":event {{
  blablub {{ }}
}}
      </literal>
    </Parameter></having>
  </Expression></solve>
  <issuer><Client><notifyTo><Interface>
    <target
    >http://localhost:8080/r3/service/dumpster</target>
  </Interface></notifyTo></Client></issuer>
</Evaluate>

<deregister>
  <subject>http://dummy.nop/xchange?id=detect</subject>
</deregister>

<request xmlns:log="http://www.semwebtech.org/lang/2006/logic#">
  <subject>http://dummy.nop/xchange?id=detect2</subject>
  <reply-to>http://localhost:8080/r3/service/dumpster</reply-to>
  <opaque>
ON
  andthen [
    "xchange":event {{
      var Ev1 -> blablub {{ }}
    }},
    "xchange":event {{
      var Ev2 -> blablub {{ }}
    }}
  ]
END
  </opaque>
  <log:variable-bindings>
    <log:tuple>
      <log:variable name="Ev1"/>
      <log:variable name="Ev2"/>
    </log:tuple>
  </log:variable-bindings>
</request>

<deregister>
  <subject>http://dummy.nop/xchange?id=detect2</subject>
</deregister>

<Evaluate
  xmlns="http://rewerse.net/I5/NS/r3/2005#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xml:base="http://rewerse.net/I5/NS/r3/2005/eval/xchange"
  rdf:about="http://dummy.nop/xchange?id=theevent">
  <solve><Expression><is rdf:resource="#raise" />
    <having><Parameter><is rdf:resource="#event" />
      <literal rdf:parseType="Literal"
      >blablub { "o tal do xxx" }</literal>
    </Parameter></having>
  </Expression></solve>
  <issuer><Client><notifyTo><Interface>
    <target
    >http://localhost:8080/r3/service/dumpster</target>
  </Interface></notifyTo></Client></issuer>
</Evaluate>

<Evaluate
  xmlns="http://rewerse.net/I5/NS/r3/2005#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xml:base="http://rewerse.net/I5/NS/r3/2005/eval/xchange"
  rdf:about="http://dummy.nop/xchange?id=prxevent">
  <solve><Expression><is rdf:resource="#raise" />
    <having><Parameter><is rdf:resource="#event" />
      <literal rdf:parseType="Literal"
      >blablub { "o tal do xxx" }</literal>
    </Parameter></having>
  </Expression></solve>
  <issuer><Client><notifyTo><Interface>
    <target
    >http://localhost:8080/r3/service/dumpster</target>
  </Interface></notifyTo></Client></issuer>
</Evaluate>

<request xmlns:log="http://www.semwebtech.org/lang/2006/logic#">
  <subject>http://dummy.nop/xchange?id=mlevent</subject>
  <reply-to>http://localhost:8080/r3/service/dumpster</reply-to>
  <opaque>
RAISE
  "xchange":event {{
    "xchange":recipient { "http://localhost:4711" },
    blablub { var X }
  }}
END
  </opaque>
  <log:variable-bindings>
    <log:tuple>
      <log:variable name="X">o tal do zzz</log:variable>
    </log:tuple>
  </log:variable-bindings>
</request>

<request xmlns:log="http://www.semwebtech.org/lang/2006/logic#">
  <subject>http://dummy.nop/xchange?id=upd</subject>
  <reply-to>http://localhost:8080/r3/service/dumpster</reply-to>
  <opaque>
TRANSACTION
  or [
   in { resource {"file:travel5.xml"},
     travel {{
       delete train {{ name{var X} }},
       insert name{var X},
       currency {"EUR" replaceby "DM"}
     }}
   },
   in { resource {"file:travel5.xml"},
     travel {{
       delete name{var X},
       insert train {{ name{var X} }},
       currency {"DM" replaceby "EUR"}
     }}
   }
  ]
END
  </opaque>
  <log:variable-bindings>
    <log:tuple>
      <log:variable name="X">t1</log:variable>
    </log:tuple>
  </log:variable-bindings>
</request>
```

A.5 XQuery Engine

Java Excerpts:

```
package net.rewerse.i5.r3.eval.xquery;
  ...
public class Evaluator extends ExprEvaluator {
  ...

protected void evaluate(Context ctx) throws Exception {
  String buri = notEmpty(ctx.text("base-uri"));
  String xmldoc = notEmpty(ctx.literal("document"));
  String query = ctx.literal("literal");
  boolean wrap = !ctx.text("format").equals("raw");

  XQueryEvaluator exp = new XQueryEvaluator(
      autoDeclare(),
      buri, xmldoc, query, ctx.renames());

  LinkedList<String> pvars = new LinkedList<String>();
  for (String vr: exp.getParameters()) {
    String vn = ctx.var(vr);
    if (vn != null) pvars.add(vn);
  }
  Iteratable<String> pars = new Iteratable<String>(pvars);

  for (Tuple t : ctx.getUsing(pars)) {
    QryEvaluation eval = exp.evaluate(wrap);
    for (Variable v : t.getBinding()) {
      String n = v.getName(), r = ctx.rename(n);
      eval.setParameter(r, asObject(v.getLiteral()));
    }
    while (true) {
      String res = eval.nextResult();
      if (res == null) break;
      ctx.addResult(t, res);
    }
  }
}
  ...
}
```

Examples:

```
<request xmlns:log="http://www.semwebtech.org/lang/2006/logic#">
  <subject>http://dummy.nop/xqq</subject>
  <opaque>
  for $X in ("aaa", "bbb")
  return <doc><xxx>{$X}</xxx><yyy>{$Y}</yyy></doc>
  </opaque>
  <log:variable-bindings>
    <log:tuple>
      <log:variable name="Y">ccc</log:variable>
    </log:tuple>
    <log:tuple>
      <log:variable name="Y">ddd</log:variable>
    </log:tuple>
    <log:tuple>
      <log:variable name="Y">eee</log:variable>
    </log:tuple>
  </log:variable-bindings>
</request>

<request xmlns:log="http://www.semwebtech.org/lang/2006/logic#">
  <subject>http://dummy.nop/xqq</subject>
  <opaque>
  fn:doc($W)/*:feed/*:entry/*:link/@href
  </opaque>
  <log:variable-bindings>
    <log:tuple>
      <log:variable name="W"
      >http://code.google.com/feeds/updates.xml</log:variable>
    </log:tuple>
    <log:tuple>
      <log:variable name="W"
      >http://code.google.com/feeds/featured.xml</log:variable>
    </log:tuple>
  </log:variable-bindings>
</request>

<request xmlns:log="http://www.semwebtech.org/lang/2006/logic#">
  <subject>http://dummy.nop/xqq</subject>
  <Expression
    xmlns="http://rewerse.net/I5/NS/r3/2005#"
    xml:base="http://rewerse.net/I5/NS/r3/2005/eval/xquery"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <is rdf:resource="#opaque" />
    <having><Parameter><is rdf:resource="#format" />
      <literal>wrap</literal>
    </Parameter></having>
    <having><Parameter><is rdf:resource="#base-uri" />
      <literal>http://code.google.com/feeds/</literal>
    </Parameter></having>
    <having><Parameter><is rdf:resource="#document" />
      <literal rdf:parseType="Literal">
  <config xmlns="">
    <mode>escaped</mode>
    <entry>1</entry>
  </config>
      </literal>
    </Parameter></having>
    <having><Parameter><is rdf:resource="#literal" />
      <literal>
  xquery version "1.0";
  declare namespace atom = "http://purl.org/atom/ns#";
  declare variable $W external;
  let $m := fn:string(config/mode), $i := fn:number(config/entry)
  for $r in fn:doc($W)/atom:feed/atom:entry[$i]/atom:title[@mode=$m]
  return (fn:string($r), $r/@type, $r)
      </literal>
    </Parameter></having>
  </Expression>
  <log:variable-bindings>
    <log:tuple>
      <log:variable name="X">x1</log:variable>
      <log:variable name="W">updates.xml</log:variable>
    </log:tuple>
    <log:tuple>
      <log:variable name="X">x2</log:variable>
      <log:variable name="W">updates.xml</log:variable>
    </log:tuple>
    <log:tuple>
      <log:variable name="X"/>
      <log:variable name="W">featured.xml</log:variable>
    </log:tuple>
  </log:variable-bindings>
</request>

<request xmlns:log="http://www.semwebtech.org/lang/2006/logic#">
  <subject>http://dummy.nop/xqq</subject>
  <Expression
    xmlns="http://rewerse.net/I5/NS/r3/2005#"
    xml:base="http://rewerse.net/I5/NS/r3/2005/eval/xpath"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <is rdf:resource="#opaque" />
    <having><Parameter><is rdf:resource="#format" />
      <literal>raw</literal>
    </Parameter></having>
    <having><Parameter><is rdf:resource="#base-uri" />
      <literal>http://code.google.com/feeds/</literal>
    </Parameter></having>
    <having><Parameter><is rdf:resource="#document" />
      <literal rdf:parseType="Literal">
  <config xmlns="" xmlns:atom="http://purl.org/atom/ns#">
    <!-- using xpath -->
    <!-- all namespace prefixes used here, get declared -->
    <atom:entry>1</atom:entry>
  </config>
      </literal>
    </Parameter></having>
    <having><Parameter><is rdf:resource="#literal" />
      <literal>
  for $i in config/atom:entry
  return fn:string(
    fn:doc($W)/atom:feed/atom:entry[fn:number($i)]/atom:title)
      </literal>
    </Parameter></having>
  </Expression>
  <log:variable-bindings>
    <log:tuple>
      <log:variable name="W">updates.xml</log:variable>
    </log:tuple>
    <log:tuple>
      <log:variable name="W">featured.xml</log:variable>
    </log:tuple>
  </log:variable-bindings>
</request>
```