Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Event-Condition-Action rules for distributed content management

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Information and Knowledge Management

eingereicht von

Wolfgang Ziegler
Matrikelnummer 0425861

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer/in: O.Univ.Prof. Dipl.-Ing. Dr.techn. A Min Tjoa
Mitwirkung: Mag. Dipl.-Ing. Dr. Amin Anjomshoaa

Wien, 23.07.2010 ________________________ ________________________

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

Wolfgang Ziegler, Favoritenstraße 68/9, A-1040 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 23.07.2010

____

Wolfgang Ziegler

Abstract

For the popular open source Content Management System (CMS) Drupal the Rules extension module makes it feasible for users to configure reactions on a high level without requiring any programming expertise. To achieve that, the extension allows for the specification of reactive rules - or more precisely it leverages Event-Condition-Action rules, for which the actions are executed when a specified event occurs and the conditions are met. Finally the ability to create custom reactions constitutes an opportunity for users to rapidly adapt the behavior of Drupal based web applications.

In this thesis the existing Rules extension module is analyzed and revised in order to obtain an extensible and reusable solution, for which all identified flaws have been eliminated. Moreover the module is advanced to work across system boundaries, such that it can be utilized for the rule-based invocation of web services as well as for reacting on remotely occurring events. Therefore the solution obtains the ability to work with arbitrary data structures with the help of metadata, so that the data of remote systems can be seamlessly integrated based on metadata. Building upon this capability we present the rule-based utilization of RESTful and WS* web services and introduce Rules web hooks - a novel approach for the interaction of Drupal based web applications that exploits reaction rules to enable custom near-instant reactions on remotely occurring events.

Zusammenfassung

Das Rules Erweiterungsmodul für das weit verbreitete Open Source Content Management System (CMS) Drupal erlaubt Drupal-Benutzern individuelle Reaktionen auf einem hohen Abstraktionsgrad zu definieren, ohne dass dafür Programmierkenntnisse nötig wären. Hierfür können Benutzer Reaktionen mit Hilfe sogenannter ECA-Regeln formulieren. Diese reaktiven Regeln bestehen aus Ereignissen, Bedingungen und Aktionen, wobei die Aktionen ausgeführt werden, wenn eines der definierten Ereignisse auftritt und die Bedingungen erfüllt sind. Schließlich ermöglicht das Erstellen von benutzerdefinierten Reaktionen Benutzern das Verhalten von Drupal-basierten Webapplikationen auf schnelle Art und Weise anzupassen.

In dieser Arbeit wird das bestehende Rules Modul analysiert und überarbeitet, sodass nach Ausmerzen der erkannten Schwächen eine flexibel erweiterbare und wiederverwendbare Lösung entsteht. Darüber hinaus wird das Modul verbessert, um über Systemgrenzen hinweg zu funktionieren. Dies geschieht in dem die regelbasierte Einbindung von Web Services als auch das Reagieren auf entfernt auftretende Ereignisse ermöglicht wird. Um dies zu erreichen, wird das Modul um die Fähigkeit erweitert mit beliebigen Datenstrukturen mit Hilfe von Metadaten zu arbeiten, so dass durch die Verwendung von Metadaten die Daten von entfernten Systemen nahtlos integrierbar werden. Darauf aufbauend zeigen wir die regelbasierte Einbindung von REST und WS* Web Services und stellen Rules Web Hooks vor - ein neuartiger Ansatz für die Interaktion von Drupal-basierten Webapplikationen, welcher reaktive Regeln für die beinahe augenblickliche Reaktion auf entfernt auftretende Ereignisse einsetzt.

Contents

Contents iv

List of Figures vi

List of Tables vii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Outline of the thesis . . . 2

2 Foundations 4
- 2.1 Rules . . . 4
- 2.2 Web services . . . 8
- 2.3 Drupal . . . 13

3 Related work 25
- 3.1 XChange . . . 25
- 3.2 Drools . . . 27
- 3.3 Rule engines in Content Management Systems . . . 29

4 Objectives 31

5 Realization 34
- 5.1 Analysis . . . 34
- 5.2 Architecture . . . 39
- 5.3 Foundational APIs . . . 54
- 5.4 Implementation . . . 60

6 Use cases **74**
6.1 Editorial workflow 74
6.2 A distributed workflow 80
6.3 Other significant use cases 84

7 Evaluation **87**

8 Conclusions and outlook **89**
8.1 Future work 90

A Acronyms **94**

B API documentation **96**
B.1 Rules module 96
B.2 Rules web extension modules 111

C References **118**

List of Figures

2.1 Rules in the Semantic Web layer cake [Bra07]. 8
2.2 WS* web services architecture [NL04]. 10
2.3 User interface for defining fields of comments posted to article nodes. . . 15
2.4 Default RDF mapping used by Drupal 7. 17
2.5 An example explaining the concept applied by the Trigger module [DRD]. 19
2.6 An Event-Condition-Action rule that reacts when a user updates a node to notify the node author [DRD]. 21
2.7 User interface for the configuration of an ECA rule in Drupal [DrR09a]. . 23

5.1 An action configuration showing the use of token replacements. 36
5.2 The design of the Entity metadata module. 41
5.3 Architectural overview of the enhanced Rules module. 42
5.4 Overview over the possible argument configuration modes. 43
5.5 The architectural overview extended to include Rules web hooks. 45
5.6 The role of remote proxies for integrating remote systems. 46
5.7 The extended base module architecture including remote proxies. 47
5.8 The propagation of available variables in a rule configuration. 52
5.9 Available variables in a loop configuration. 53
5.10 The class hierarchy of the provided plugin implementations. 61

6.1 The states of the editorial workflow. 75
6.2 The node revision log listing workflow state transitions. 79
6.3 The editorial workflow extended with a separate front-end site. 80

8.1 A user interface for the enhanced Rules module. 91

List of Tables

2.1 Mapping CRUD operations to HTTP methods [BB08]. 11

5.1 The plugins provided by the Rules module and their properties. 50

CHAPTER 1

Introduction

1.1 Motivation

Many web applications, such as communication platforms, e-learning systems or online marketplaces make use of event-based programming in order to react on events such as web page updates or data being posted to a server. Hence reactivity, the ability to detect and react on events is an integral component of today's web applications [BE06]. For Drupal[1], a popular open source Content Management System (CMS) and web application framework, the Rules[2] extension module makes it feasible for regular Drupal users to specify custom reactions on a high level without requiring any programming expertise. Therefore the ability to create custom reactions enables users to rapidly adapt the behavior of Drupal based web applications and provides site builders with new opportunities for Drupal site customization, e.g., by sending notification e-mails, issuing well-adjusted page redirects or by making content publishing workflows more flexible.

To achieve that the Rules module allows for the specification of reactive rules - or more precisely it utilizes Event-Condition-Action rules, which correspond to conditionally executed actions that are triggered based on occurring events. Ideally rules are easy to understand for humans, but at the same time well-suited for processing

[1]Drupal: `http://drupal.org` (accessed 03-06-2010).
[2]Rules module: `http://drupal.org/project/rules` (accessed 03-06-2010).

and analysis by machines. Also rule-based specifications are flexible, as well as easy to adapt and alter as requirements change [BE06]. Thus the module forms an easy to use, but powerful solution providing user-defined reactions for Drupal based web sites.

The existing Rules module for Drupal constitutes the foundation for this thesis. Consequently the starting point of our work forms the advancement of the module, whereby the existing solution has to be analyzed and all identified limitations have to be addressed. In addition, to cope with the emergence of web applications and web services and their dynamic nature [BBB+07], the main objective of our work is to take the module to the next level, such that it is able to seamlessly integrate the data of remote systems and to work across system boundaries. To achieve that our work should enable the rule-based invocation of web services and offer means to simply react on events occurring on remote Drupal sites. That way the module lays the foundation for leveraging rules for a flexible, dynamic, but user controlled utilization of the emerging "Web of Data" with Drupal (cf. [CDC+09]).

Beside that, as the Drupal community is currently working on bringing a new major release - Drupal 7 - to completion, the enhanced Rules module is already developed for the upcoming version - Drupal 7. First off all this enables us to build upon the improved APIs of Drupal 7, but moreover this facilitates the creation of a well-integrated solution, which is able to exploit the innovative architecture of Drupal 7 in an optimal way.

1.2 Outline of the thesis

Chapter 2 introduces the foundational technologies and work for this thesis. Thus it gives an overview about rules in general, web service technologies and Drupal, whereby we also describe some relevant Drupal extension modules.

In chapter 3 research and systems related to our work are presented, whereas the differences compared to the Rules module are pointed out.

The objectives of our work are stated in chapter 4. Each objective is explained in detail and reasons for its relevance are given.

Chapter 5 describes the realization of the enhanced module. The chapter starts with an analysis of the objectives presented in the preceding chapter. Based upon this analysis and the objectives the module architecture is revised and elaborated in detail. This forms the foundation for the actual implementation, for which an overview is provided.

Next we show how our work can be utilized to implement some use cases in chapter 6. We present how a simple editorial workflow can be accomplished and improve the solution to work in a distributed way.

In chapter 7 we evaluate our work by revisiting the objectives as stated in chapter 4 in order to show that we have succeeded in attaining all objectives.

Finally chapter 8 concludes the thesis, whereas an overview over possible future enhancements is given.

CHAPTER 2

Foundations

This chapter presents the foundations for this thesis. Thus it gives an overview about rules, web services and the Content Management System Drupal, whereas we focus on parts which are of a particular interest for this work.

2.1 Rules

Rules are becoming a more and more important part of the IT Systems today. Many companies and academics use declarative rule languages to develop flexible applications on a high abstraction level, but also many companies use rules to manage their business logic or to model security policies. Apart from that they are frequently utilized for reasoning in the Semantic Web [BKPP07].

Categorization

Rules can be generally seen as self-contained knowledge units that involve some form of reasoning. But they are significantly different depending on their purpose. For instance they are used for

- static or dynamic integrity constraints
- derivations (e.g., define derived concepts)
- reactions (specify the reactive behavior of a system)

[WATB04, BKPP07]

Harold Boley, Michael Kifer, Paula-Lavinia Patrânjan and Axel Polleres [BKPP07] identify the following types of rules:

Deduction rules are used to to derive further knowledge from already existing knowledge by using logical inference or mathematical calculations. Usually they consist of one ore more conditions and one or more conclusions and are often specified as implications of the form $Conclusions \Leftarrow Conditions$.
Deduction rules are also called constructive rules in logical languages, derivation rules in the business rules community or views in databases [BKPP07, TW01, BM05, WATB04].

Normative rules express conditions or integrity constraints that data must fulfill in all states of the system, e.g., a rule might assert that an identification number is unique. So normative rules ensure that changes do not cause inconsistencies and that the business logic of companies is obeyed.
Normative rules are also called integrity constraints in databases or structural rules in the business rules community, whereas the term normative rules is usually used by logicans [BKPP07, BM05, WATB04].

Reactive rules describe reactive behavior and implement reactive systems by specifying actions depending on the current state of the system or based on events. They are of the form `if condition then action` or `on event if condition then action`, whereas rules of the first kind are called production rules and rules of the second Event-Condition-Action rules (or short ECA rules).

Production rules evaluate the conditions whenever a change to the underlying database occurs and execute the actions if the conditions are met. Actions usually contain operations on the underlying data, but also statements known by procedural languages like assignments or loops [WATB04, BKPP07, BM05].

Event-Condition-Action rules however are only evaluated when the specified events occur, whereas an event represents changes in the state of the world.

> Thus actions of a rule are executed if the event occurs and the conditions are met. Depending on the rule language one can specify a single event (*atomic events*) or also combinations of events (*composite events*), e.g., a temporal combination of multiple atomic events [BKPP07, TW01, BM05]. Berstel, B. and Bonnard, P. and Bry, F. and Eckert, M. and Patrânjan, P. [BBB+07] discuss both types of reactive rules thoroughly and point out the differences when using them for reactive web applications. They conclude that Event-Condition-Action rules are better suited for distributed applications relying on event based communication, whereas production rules have their strengths in logical applications with the focus on managing the state of web nodes than on the distribution aspects.

Reactive rules are also called active rules, dynamic rules or just reaction rules [BKPP07].

While the three identified types of rules - normative rules, production rules and reactive rules - differ in many aspects, it is found that they are sometimes utilized together. For instance deductive rules may be used to implement normative rules, as in databases deductive rules are used to specify constraints. Furthermore databases implement a form of reactive rules to enforce integrity constraints. Thus which type of rules is feasible to leverage largely depends on the application and on the available support of rule types [BKPP07, WATB04].

Business rules

The Business Rule Group[1] - an independent organization of business and IT professionals - defines a business rule as statement that defines or constraints some aspects of business and is intended to assert business structure or to control or influence the behavior of the business [Gro00].
Business rules represent the logic of a company, modeled as separate entities. Explicitly modeling the business logic in rules offers great flexibility as business rules can be changed very fast and without programming knowledge, e.g., by business analysts.

[1] Business Rule Group: `http://www.businessrulesgroup.org` (accessed 02-02-2010).

What makes it even more valuable is that usually the business logic changes more frequent than the application logic. As business rules are declarative statements, they are decoupled from the system what helps to increase maintainability [RD05]. An example of a business rule as described by Romanenko, I. [RPB+06] would be:

```
IF the purchase value is greater or equal to $100
THEN change the customer category to "gold"
```

The example makes use of the `IF-THEN` construct known by many business rule systems and just uses natural language for describing the parts of the rule.

As rules in general business rules can be classified in normative rules, production rules and reactive rules - as described earlier. However a fourth type - deontic assignments - is partially identified [Wag02, RD05]. According to Wagner G. [Wag02] "deontic assignments of powers, rights and duties to (types of) internal agents define the deontic structure of an organization, guiding and constraining the actions of internal agents".

Rules in the Semantic Web

Rules in the Semantic Web serve a similar purpose as business rules - both are supposed to capture semantics about the real world. However the goal of business rules is to improve human communication while in the Semantic Web the improvement of the communication between machines is the objective [SG04].

The need for rules in the Semantic Web was envisioned already at the very early stages of the Semantic Web. Hence rules play an important role in the Semantic Web layer cake as seen in figure 2.1. Nowadays rules for the Semantic Web become the focus of intense activity. With RuleML - the Rule Markup Language - a standard serialization language is provided, what forms the base for the Semantic Web Rule Language (SWRL) which combines RuleML and OWL, the Web Ontology Language [KDBBF05, BKPP07].

More recently the World Wide Web Consortium (W3C) is working on standardizing a common format for rule interchange on the web - the Rule Interchange Format (RIF) - which enables the reuse of knowledge specified in different rule languages [BKPP07].

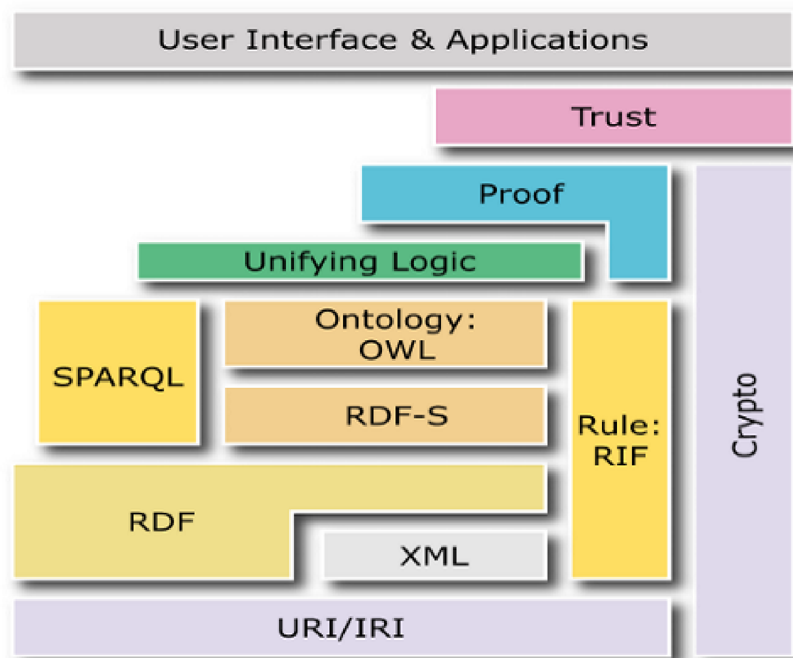

Figure 2.1: Rules in the Semantic Web layer cake [Bra07].

2.2 Web services

Web services are a common solution for application - application interaction, which can vary from simple requests like checking the weather report to complete business applications that access and combine multiple sources like an automated travel planner. There are quite some protocols and standards for building web services as well as different approaches: The Service Oriented Architecture (SOA) and the Resource Oriented Architecture (ROA) [Pap08, RR07].

Service Oriented Architecture

The Service Oriented Architecture (SOA) evolved as a principal for designing software systems to provide services to end-user applications or other services. It is the dominating mechanism for application integration and the most common architectural paradigm in enterprise level application development today.
To implement a Service Oriented Architecture organizations are providing existing, previously siloed applications as services available for other applications, thus en-

abling applications running with different technologies to communicate with each other [Pap08, A.09].

According to Newcomer, E. and Lomow, G. [NL04] the four primary characteristics for the design, implementation, and management of services in the Service Oriented Architecture are:

- **Loosely coupled.** Clients should be able to use a service without any knowledge of its internals.
- **Well-defined service contracts.** A service contract specifies the capabilities and technical access properties of a service.
- **Meaningful to service requesters.** Services and service contracts have to be defined on an appropriate level of abstraction, which is meaningful to service requesters.
- **Standards-based.** Services should rely on open standards to increase the number of potential service consumers and make use of standard or mature vocabularies in its associated domain. [NL04, A.09]

SOA is an architectural paradigm and independent of any specific technology, however the concept of SOA is often discussed in conjunction with web services as underlying technology for implementing a SOA, for which usually the WS* based web services are leveraged. For that clients must be able to find services they require - therefor the main building blocks are the service provider, the service registry and the service requester [Pap08].

WS* web services

The WS* based web services rely on the three core standards: WSDL, SOAP and UDDI. They are also called *Big Web Services*, *SOAP oriented Web Services* or *WSDL based services* [A.09, RR07]. The interaction of a service requester with a service provider is supported via SOAP and WSDL and service discovery via UDDI, which serves as service registry as shown in figure 2.2. Thus the provider usually publishes the service description using WSDL, the Web Service Description Language. The

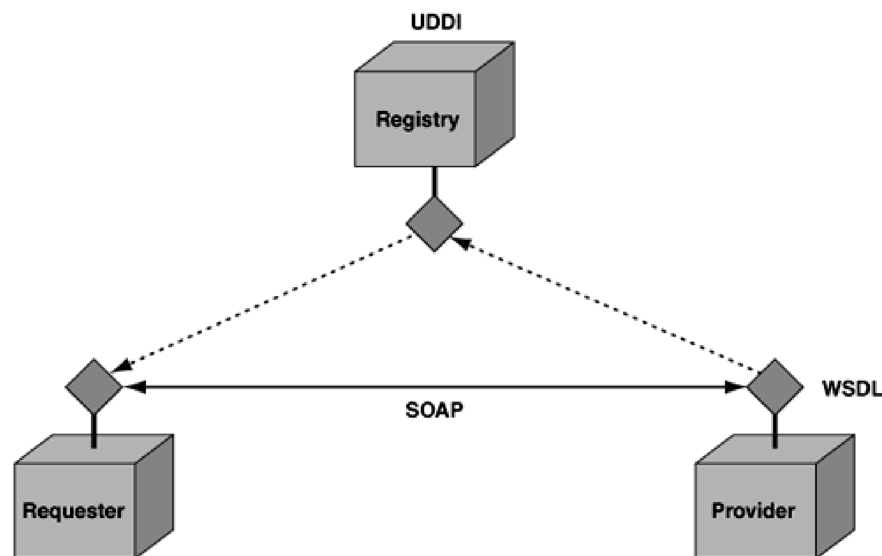

Figure 2.2: WS* web services architecture [NL04].

service requester then accesses the description using the UDDI registry and uses the service by sending a SOAP message to the provider. Hence WSDL, SOAP and UDDI can serve as the building blocks for a Service Oriented Architecture as described in the preceding section 2.2. But apart from the core specifications - WSDL, SOAP an UDDI - there are also a lot of different specifications that extend the WS* web service stack with support for security, reliability, transactions, metadata management and orchestration.
However the vision for UDDI was to become a public directory for services. Companies were supposed to register their services so that others could discover services dynamically. But the vision has not been realized, instead UDDI has more success as in-side the enterprise technology [NL04].

Event Driven Architecture

As its name implies the Event Driven Architecture (short EDA) builds upon events on which the system may react. When the Event Driven Architecture is applied to SOA, sometimes the terms event-driven SOA, SOA 2.0 or just EDA are used.
In an event-driven SOA the occurrence of an event may trigger service invocations.

Apart from that services may generate further events. Upon generation the event is passed to all involved parties which then evaluate the event and optionally react [Mic06, Jos08].
The Event Driven Architecture has the advantage to further decouple services from each other. However due to the loose coupling services are not aware of its dependencies any more what may pose maintenance difficulties [Jos08].

RESTful web services

Representational State Transfer - short REST - is an architectural style that became defacto standard for publishing web services in the Web 2.0. It was first introduced by Fielding, R. T. in the fifth chapter of his dissertation published in the year 2000 [Fie00]. Web Services that are conform to the REST principles are called RESTful web services, which serve as a light weight alternative to WS* web services described previously in the section 2.2. Compared to them RESTful web services deal only with data structures that represent the current or intended state of resources while the so called *Big Web Services* make use of full blown remote objects and remote method invocation [BB08, RR07, A.09].
A REST conform design is based upon a set of state transfer operations universal to any data storage - commonly referred to by the acronym CRUD standing for Create, Read, Update and Delete. While there is no standard defined by the W3C or similar organizations the web community commonly leverages the HTTP methods POST, GET, PUT and DELETE to map them to CRUD operations as shown in table 2.1 [BB08].

Table 2.1: Mapping CRUD operations to HTTP methods [BB08].

CRUD operation	HTTP method
Create	POST
Read	GET
Update	PUT
Delete	DELETE

While REST has become popular in the web a lot of widespread services like

the API of flickr[2] or delicious[3], which are commonly described as REST services are not really REST conform, rather they can be described as REST-RPC hybrids. They integrate well with the web for retrieving data, but for modifying data they work more like RPC style services which make use of remote method invocations [RR07].

Resource Oriented Architecture

The Resource Oriented Architecture (ROA) is a concrete application of the REST architectural design guidelines. It applies the REST principles to web technologies, thus it heavily builds upon HTTP (Hypertext Transfer Protocol) and URIs (Uniform Resource Identifier). The central concepts of the Resource Oriented Architecture are resources, their names, their representations and the references between them - whereas resources represent abstract information items, which are identified by URIs. The main characteristics of the Resource Oriented Architecture are addressability, statelessness, connectedness and the uniform interface [RR07].

Addressability means that resources are identified and accessible through a unique naming mechanism - the uniform resource identifiers (URIs) as introduced by Tim Berners-Lee in RFC 1630 [BL94, RR07].

Statelessness connotes that each request of the client is isolated from previous or subsequent requests, thus it has to include all information that is necessary to handle the request. Therefore any session state has to be kept on the client. Statelessness helps to improve visibility, reliability and scalability, but has the disadvantage that the network performance may decrease due to repetitive data sent in request series [Fie00, RR07].

Connectedness implies that resource representations may contain references to other resources, which may reside on the same system but also on external systems. That way resources may be connected to each other just like regular web pages. Fielding, R. T. called this concept “hypermedia as the engine of application state” in his dissertation [Fie00, RR07].

---

[2]flickr: `http://www.flickr.com/services/api/` (accessed 04-02-2010).
[3]delicious: `http://delicious.com/help/api` (accessed 04-02-2010).

Uniform interface. The Resource Oriented Architecture leverages HTTP methods to define a uniform interface for interactions between the consumer and the service provider. As mentioned in 2.2 HTTP methods are mapped to the four basic CRUD operations CREATE, READ, UPDATE and DELETE as shown in table 2.1. Furthermore the HTTP method HEAD is used to only retrieve the metadata of a resource and the method OPTIONS to determine the supported operations for a resource. However in practice the HTTP method OPTIONS is seldom implemented [RR07, A.09].

2.3 Drupal

Drupal is a highly modular and flexible open source Content Management System (CMS) and Framework (CMF) with the main focus on collaboration. Drupal's strengths are its extensibility and standard compliance as well as its clean code base. It is built upon the PHP scripting language and requires a database server - whereas currently the MySQL and PostgreSQL databases are supported. Drupal is designed to run on widespread and cheap web hosting servers, but to be highly scalable at the same time. Thus it is among the three top open source CMS systems in terms of market share and it is leveraged at IBM, NASA, Yahoo, Sony, MTV among others. Recently also the flag ship website of the U.S. government *Whitehouse.gov* has been relaunched with Drupal [vD09, CCPD08, Hoj07, Pet09, Buy09].

Drupal core

The main distribution of Drupal contains only the most widely used features and is commonly referred to as Drupal core. It is publicly developed on the website drupal.org[4], which is like Drupal built by the community but backed by the Drupal Association[5] - a non profit organization dedicated to help Drupal to flourish. Drupal.org also provides free facilities for developing open source Drupal extensions and thus it serves as the main source for Drupal extensions - so called Drupal mod-

[4] drupal.org: `http://drupal.org` (accessed 04-02-2010).
[5] Drupal Association: `http://association.drupal.org` (accessed 04-02-2010).

ules. Those modules interact with Drupal core and other modules by a hook system, which makes it possible to extend the existing functionality and so forms the base for a vast number of useful extensions [Pet09, vD09].
As of now the latest Drupal release is Drupal 6, however Drupal 7 is already under active development since the first Drupal 6 release in February, 2008. The first alpha release of Drupal 7 has already been published and currently the community is working towards the stable Drupal 7 release [DrA10].

From nodes to entities

The main building block of Drupal are so called nodes, which are also referred to as content. They serve as flexible site content that may be created by site administrators or regular site users. Site builders can control the structure of several different node types and extend their functionality with the help of modules. Thus nodes are used for blog entries, writing articles, posting receipts, representing tasks and much more while comments, files, ratings, etc. may be attached to nodes.
All that great functionality can be built with the help of nodes thanks to Drupal's powerful API around nodes [vD09]. However with Drupal 7 the community has started to introduce the concept of *entities* in Drupal, which is the start of a common API for various Drupal data objects, such as nodes, user accounts, taxonomy terms, comments and others [DrE09]. This move enables modules to build upon this common API to easily support all kind of Drupal entities instead of building upon the node API only. Consider a rating module: Instead of providing a way to rate nodes only, the same module built based on the concept of entities could be utilized to rate nodes, comments, taxonomy terms or even other users.

CCK, Field API

As of Drupal 6 each node has some basic properties like the node title and the body, which is the main content of the node. While modules may extend this properties the Content Construction Kit - short CCK - is a popular Drupal module that makes it possible for site builders to define custom properties - so called fields - just via the module's user interface. For that modules may provide further field types and

Figure 2.3: User interface for defining fields of comments posted to article nodes.

widgets, which then site builders can make use of. That way the module makes it possible to construct arbitrary structured nodes and so it has quickly become one of the most widespread Drupal modules [Pet09, vD09]. Given the importance of the module it was a natural choice for the community to integrate the CCK module into Drupal core. However the module lacked usable public APIs for programmatical configuration of CCK fields. So for inclusion in core the module was reworked what resulted in the creation of the field API.
The field API, which is already part of Drupal 7, is not only better programmatically configurable, moreover it has been improved to not only work with nodes, but to build upon the concept of entities. That way it is possible in Drupal 7 to make any entity *fieldable*, what means the entity can be extended via the UI or programmatically by fields. Hence Drupal 7 enables site builders to define custom fields for entities like nodes, comments, taxonomy terms and user accounts out of the box. Figure 2.3 shows the user interface site builders may use to define custom fields for comments, which may vary by the type of the node that is commented. Such a combination of configured fields for an entity is referred to as *bundle* [Pet09, Né10, DrA10].

In Drupal 7 a lot of existing functionality has been refactored to utilize the field API. For example the taxonomy module, which ships with Drupal core, provides a way to categorize and tag nodes in Drupal 6. For Drupal 7 the association of nodes with taxonomy terms has been revised to be a field. As a consequence one is now able to reuse the so provided field type to tag and categorize any fieldable entity just by the definition of a new field [Né10].

Views

According to the usage statistics published on drupal.org [DrU10a] the Views module is currently the most popular Drupal module. Views makes it possible for site builders to generate listings and even searches for nodes, comments, users, taxonomy terms, etc. simply via the provided configuration interface. The module directly builds upon the database and implements a SQL query builder for joining database tables and applying user defined condition and sort filters. Furthermore the module is highly extensible so modules may define further base tables, tables to join to, selectable fields, filters and sort criteria as well as different display styles [Pet09, DrV09].

RDFa

Drupal's founder Dries Buytaert has called for semantically annotating Drupal's data using RDF at his keynote *State of Drupal* of the Drupalcon 2008 in Boston. Since that time the community has been exploring ways to support RDF in Drupal [Pet09]. For Drupal 7 an RDF module has been added to Drupal core, which semantically annotates Drupal's markup. For this task the module makes use of RDFa, which provides a set of attributes for embedding RDF in XHTML [AC08]. For that purpose the module comes with a meaningful mapping of Drupal's data to well known public vocabularies as shown in figure 2.4. Furthermore the RDF module provides ways for other modules to customize the provided default mapping as well as to provide mappings for added fields or additional data properties defined by modules [Cor10b, DrA10].

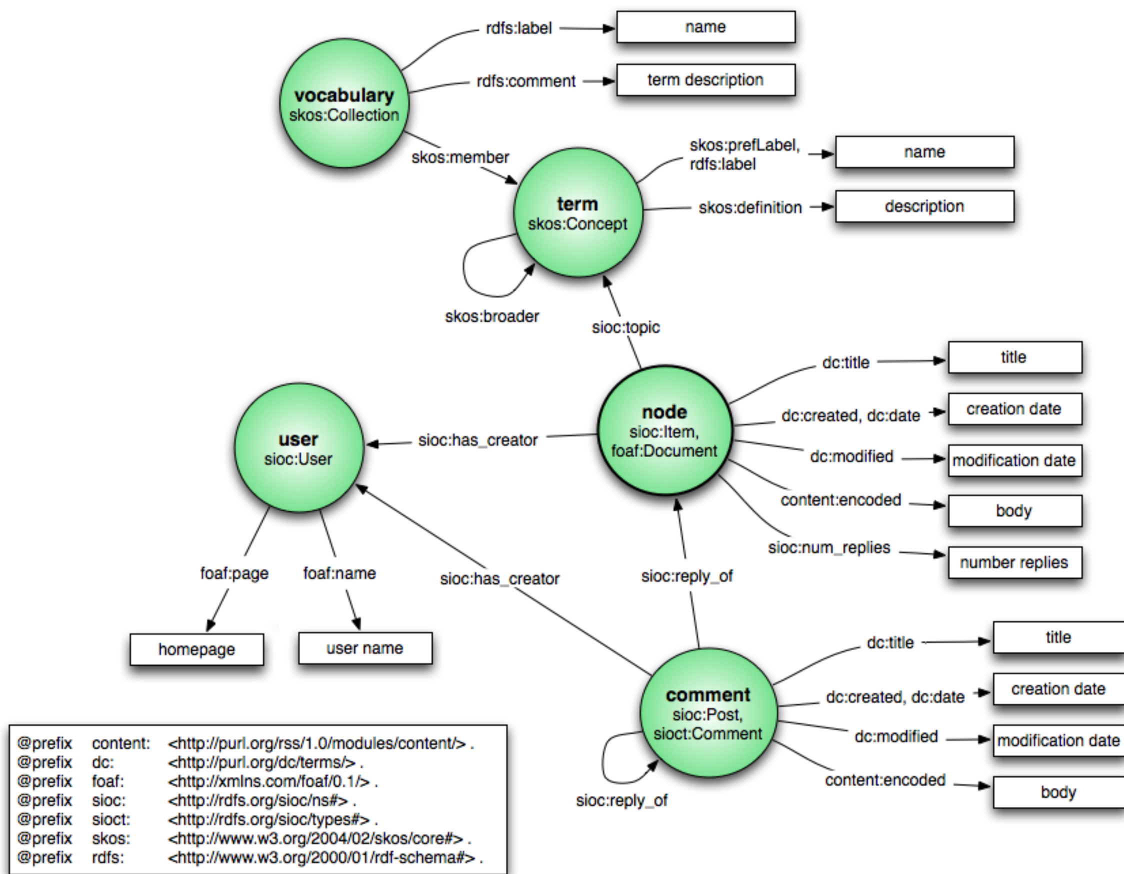

Figure 2.4: Default RDF mapping used by Drupal 7 as of 12-01-2010, created by Corlosquet, S. [Cor10b].

Replacement tokens

The Token module provides an API for other modules that makes it possible to easily integrate a variety of replacement tokens. The module integrating tokens - called *token consumer* module - provides a string containing replacement patterns and the available Drupal data objects to the token module, which then applies the replacements by making use of the Drupal data objects. The Token module provides hooks that permit other modules to provide further token replacements - those modules are called *token provider* modules [DrT, DrT09].
The Token module is available for Drupal 5 and 6 and has been integrated in Drupal core for Drupal 7 as it quickly became very popular [DrU10a] and its uses are

manifold. For instance modules leverage it to allow users to create custom messages containing relevant data tokens, to automatically create meaningful path aliases or to customize the server side file path of file uploads per user. Also Drupal 7 relies on the token replacements for the e-mails sent during account registration [DrA10]. For example the subject of the welcome e-mail in Drupal 7 as entered by the user looks like this:

```
Account details for [user:name] at [site:name]
```

So `[user:name]` gets replaced with the account name of the newly created user and `[site:name]` with the name of the Drupal site [DrA10, DrT09].

Services

The Services module allows Drupal based web sites to provide web services for integrating external applications with Drupal. For that multiple server modules are available, which enable the module to provide XML-RPC, JSON, JSON-RPC, REST, SOAP and AMF interfaces. The Services module provides a bunch of services for Drupal core, however modules are able to add custom services which then are automatically exposed by the enabled server module [DrS].
From our experience the long time only available version 1.x of the module only supported RPC style services - also the corresponding REST server is better described as a kind of a REST-RPC hybrid as explained in section 2.2. However recently the developers have started working on a new Drupal 6 compatible version 2.x of the module, which together with the REST server module offers REST conform web services. Also this version provides a new hook that make it possible for other modules to provide additional resources [DrS].

Triggers and actions

While working with Drupal it is often required to react when something notable happens on a Drupal site. The Trigger module, which is part of Drupal core, makes it possible to fire actions on certain system events, such as when a node is being created or updated. A fictional example explaining the concept is shown in figure 2.5: When

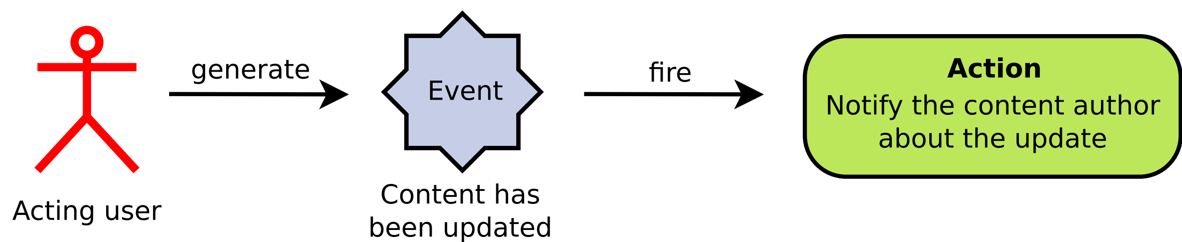

Figure 2.5: An example explaining the concept applied by the Trigger module [DRD].

the user changes a node the event "Content has been updated" is generated and the action is used to notify the content author about the update.

Drupal internally uses hooks to allow modules to react or to adapt something. The Trigger module builds upon this hooks to expose events via an interface to site builders. In turn site builders may use the interface to associate actions with available events, so the module executes the actions whenever the associated event occurs. The module itself refers to the events available as *triggers*, whereas the list of available events may be extended my modules [vD09]. Some examples of events available in Drupal core are:

- After saving a new post.
- After a user has logged in.
- When content is viewed by an authenticated user.

Actions that can be used with the Trigger module can be provided by modules and boil down to normal PHP functions that are executed whenever the action fires. Actions can be configurable, in which case one needs to configure the action before it is actually usable [vD09]. For example some actions that are available in Drupal core are:

- Unpublish comment.
- Promote post to front page.
- Redirect to URL - with a configurable URL.
- Send e-mail - with a configurable recipient, subject and message.

Each action has to be of a certain type, which is used to group the actions in the interface. Also an action has to specify whether it is compatible with all possible triggers or a list of triggers it is compatible with [vD09, Dro10].

Rules

The Rules modules allows site builders to define conditionally executed actions based on occurring events, known as Event-Condition-Action rules as described in section 2.1. It is a replacement with more features for the Trigger module in Drupal core - which has been described previously in section 2.3 - and constitutes the basis for this thesis. The module is the successor of the *Workflow-ng* module, which is available for Drupal 5 since 2007 [DrW]. It is available for Drupal 6 and according to the drupal.org usage statistics [DrU10a] it has already more than 15.000 users.

As stated in the documentation [DRD] the Rules module is a "rule-based event-driven action evaluation system". Thus just like with the Trigger module it is possible to fire actions on certain system events, however as the module relies upon Event-Condition-Action rules it is also possible to specify conditions. A rule fires and so executes the actions only when the conditions are met. Figure 2.6 shows an example for an ECA rule, which reacts when a user updates a node and notifies the node author about the update, but only if the acting user is different to the node's author.

The module comes with a bunch of useful events, conditions and actions. However any module may extend the available events, conditions and actions [DRM]. Most events are based on Drupal internal hooks similar to the Trigger module, however modules may also define new events without the existence of a hook. The module introduces a separate API for providing events, conditions and actions in addition to the API for actions provided by Drupal core and used by the Trigger module. This step allows the module to support more sophisticated actions while it is also permits the use of regular Drupal core actions [DrR09a, DRD, Dro10]. By building on a separate API the Rules module is able to offer a variety of features:

Condition groups serve as a way to configure logical operations like `AND` and `OR`. Thanks to the possibility to negate the result of condition groups this also permits the configuration of `NAND` and `NOR` operations [DRD].

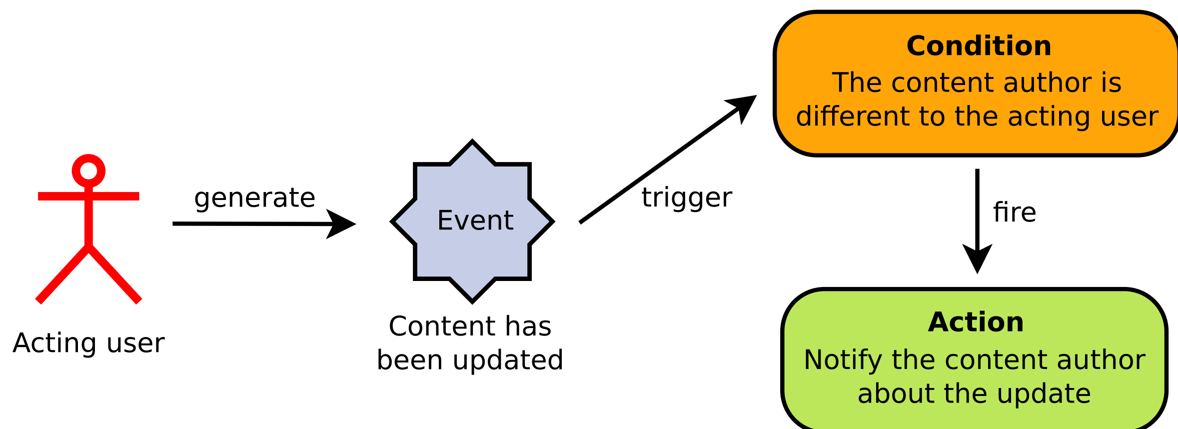

Figure 2.6: An Event-Condition-Action rule that reacts when a user updates a node to notify the node author [DRD].

Modularity. The module provides a couple of hooks which allow modules to provide further events, conditions and actions for the functionality introduced by the module [DRD]. The code needed by modules to properly implement this hooks is commonly called *Rules integration*. We pick up this terminology for the rest of the thesis.

Exportable configuration. Rule configurations can be imported and exported simply via the interface. This is useful during site development and to share rule configuration with others. Additionally it is possible for modules to provide default configurations, which users may override with the user interface unless the providing module has disabled this functionality [DRD].

Input evaluation. Rules transparently integrates input evaluation systems like the Token module described in section 2.3 or the evaluation of embedded PHP code. For that it makes all variables available for rule evaluation also available to the input evaluation system. Modules may provide new input evaluation systems to Rules, which then users may make use of when configuring conditions or actions.
The transparent integration removes the need for developers providing Rules integration to care themselves about supporting input evaluation systems. That way the developer is able to concentrate on the main purpose of his code while

still the input evaluation systems in place are supported. Hence the transparent integration avoids the need for the Rules integration developers to introduce module dependencies just for being able to support an input evaluation system like the Token module in the provided Rules integration [DrR09a, DRD].

Rule Sets serve as reusable components that can be invoked directly by modules or by rule configurations with the help of actions. Each set has to be configured to work with a defined list of arguments, which the caller has to provide. A set contains one or more rules that are only evaluated once the rule set is explicitly invoked [DRD, DrR09a].

Scheduling. The Rules module offers a flexible scheduling system which makes use of Rule Sets. In addition to the action for directly invoking a rule set an action to schedule the evaluation of a set is provided. That way the module enables scheduling of arbitrary rule configurations.
Once the action to schedule the evaluation of a rule set is executed Rules saves a new task together with the provided arguments in the database. For the scheduled execution of the task the module relies on Drupal's integration with the cron system, which permits repetitive code execution. Once a task is scheduled, the arguments are restored and the rule set is evaluated as usual [DRD, DrR09a].

Intelligent data handling. The Rules module introduces an extensible data type system, which is leveraged to intelligently deal with Drupal's data. Most important, modules may specify how data is permanently saved, which is used by the module to automatically save data objects when actions have changed them. This enables multiple actions to update the same data object while the system saves all changes at once what is critical for the module to work efficiently.
Furthermore when the system saves arguments needed to schedule the execution of rule sets to the database, it saves only the identifiers for so called identifiable data types. Once the arguments are restored the data is loaded by passing the identifier to the load routine provided by the data type. That way the amount of the data that has to be stored to save the arguments is reduced significantly and it is ensured that the scheduled rule sets operate with up-to-date data [DrR09a, DRD].

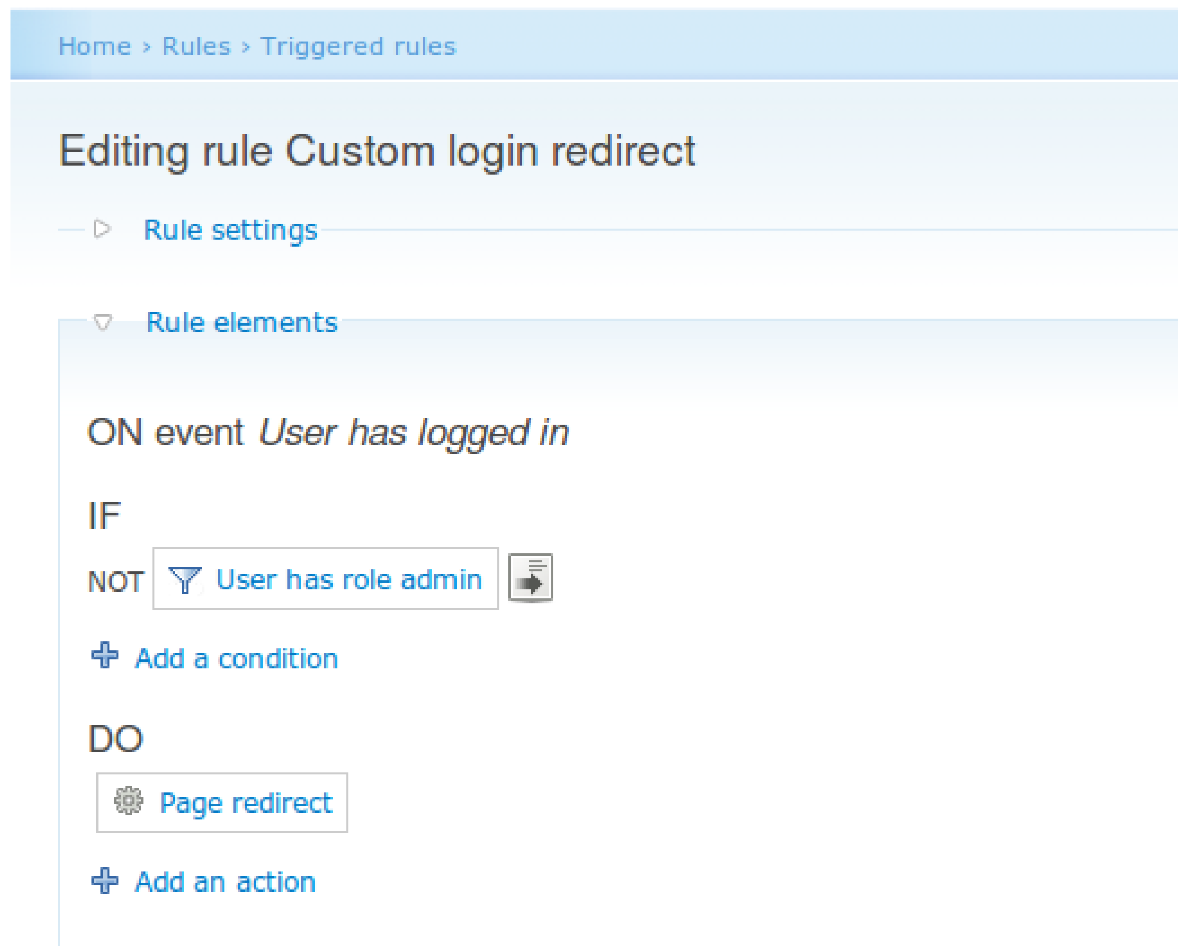

Figure 2.7: User interface for the configuration of an ECA rule in Drupal [DrR09a].

Administration UI

The rules module provides a simple but useful user interface that allows site builders to create and manage their rules as shown in figure 2.7. Action and condition configuration is embedded in the configuration workflow of a rule, thus it is not required for users to pre-configure actions in another interface to make them available for configuration - as it is the case when using the Trigger module [Dro10]. Configured reaction rules and rule sets may be categorized with tags what is helpful to keep the overview over a big rule base [DrR09a].

Comparison to the Trigger module

While the feature set of the Rules module and the Trigger module differ, there are also more fundamental differences. In contrast to the Trigger module the Rules module requires that modules that provide Rules integration describe the arguments required by conditions and actions as well as the variables an event provides. That way it is possible for the system to allow the configuration of conditions and actions wherever the necessary variables are available - thus on any event that provides a suiting variable. Furthermore it is possible for actions to provide new variables - e.g., for loaded data objects - which in turn may be used by subsequent actions or rules [Zie08, DRD].
This argument based configuration ensures that the provided conditions and actions can be reused wherever possible, thus conditions and actions written for the Rules module constitute highly reusable components [Zie08].
As described in section 2.3 the Trigger module relies upon an explicit definition of compatible events for actions. However the list of available events is extensible, thus the action cannot know and specify all events it should support beforehand. Analogously the list of actions is extensible, so the module providing the event cannot know the list of actions it could fire either. As a consequence for the Trigger module the availability of core actions for events cannot be optimal [Zie08, DrR09a].

CHAPTER 3

Related work

In this chapter we introduce research and systems related to our work and point out the differences compared to our approach. A similar system - the Trigger module part of Drupal core - has been already described in section 2.3, whereas we have pointed out the differences to the Rules module during the introduction of the Rules module.

Event-Condition-Action rule languages are topic of scientific research, thus there are several research originating ECA rule languages, e.g., XChange[1], the General Semantic Web ECA Framework[2], Prova[3] or ruleCore[4] [BKPP07]. From those examples we have a closer look at the Event-Condition-Action rule language XChange as its distributed approach is of a particular interest for our work.

3.1 XChange

XChange is a high-level and rule-based reactive web language, following a declarative approach for reactivity on the web. It allows programming both locally and distributed over several web nodes. Its development has been part of the research

[1] XChange: http://reactiveweb.org/xchange/ (accessed 24-05-2010).

[2] General Semantic Web ECA Framework: http://www.dbis.informatik.uni-goettingen.de/eca/ (accessed 24-05-2010).

[3] Prova: http://www.prova.ws/ (accessed 24-05-2010)

[4] ruleCore: http://www.rulecore.com/ (accessed 24-05-2010)

project of the REWERSE project[5] funded by the EU Commission and Switzerland [BEGP06].

XChange makes use of Event-Condition-Action (ECA) rules, whereas an XChange program may consist of multiple ECA rules. Each rule consists of an event query, a web query - forming the condition part - and an action. That means when an event answering the event query is received and the web query can be successfully evaluated, the action is performed. A XChange program runs locally at some web site, accessing local and remote data as reaction to events, whereas the reaction may trigger further reactions, locally or remote. Therefore the language is tailored for the distributed nature of the web [BEGP06, Pat05].

To achieve that XChange embeds the web query language Xcerpt. Xcerpt is a rule-based query and transformation language for the web, making use of a pattern-based approach for selecting data items and constructing new data. As an example, the following Xcerpt query matches data terms having a root labeled `accommodation` with a subterm labeled `hotel` [Pat05, Sch04]:

```
accommodation {{
  hotel {{ }}
}}
```

XChange makes use of Xcerpt for expressing the web query, but also for querying single occurrences of incoming events, whereas the integrated update language is based on Xcerpt. For the communication between web sites event messages are utilized, which contain information about the sender web site and the event. The communication follows a push strategy, so web sites inform other web sites about occurred events, whereas subscription might take place implicitly, e.g., booking a flight could implicitly subscribe to related events [Pat05, BEGP06].

Compared to the Drupal Rules module, XChange is designed to work in a distributed way from the ground up [BEGP06] and focuses on the creation of a formal, textually written rule language, whereas the Rules module's primary goal is not a formal language, but moreover to provide a simple, web based user interface integrated

[5] REWERSE project: `http://rewerse.net` (accessed 24-05-2010).

in Drupal's administration back-end. Furthermore XChange, as presented in the dissertation of Patrânjan, P. L. [Pat05], focuses on dealing with XML data structures while the Rules module puts its emphasis on a modular design, such that it is able to deal with Drupal's data structures natively and support for dealing with any data accessible with PHP could be added.

3.2 Drools

Drools[6] 5 is a business logic integration platform for rules, workflow and event processing. Drools, and its subprojects, are community releases from the JBoss Enterprise Business Rules Management System (BRMS)[7], which forms the commercial product. Therefore the Drools platform constitutes a good example demonstrating how rule engines are utilized in the business world. Drools is written in Java and consists of four main projects:

Drools Guvnor is a web-based governance system, traditionally referred to as Business Rules Management System (BRMS). A BRMS provides business use focused end users tools for rule creation, management, deployment, collaboration and analysis [dro09, Bro09].

Drools Expert is a forward chaining inference based rules engine, which can be classified as a production rule system. It focuses on knowledge representation to express propositional and first order logic in a concise, non-ambiguous, declarative manner, whereas the inference engine matches facts and data against the production rules such that conclusions resulting in actions are inferred [dro09].

Drools Fusion enables event processing behavior by supporting Complex Event Processing (CEP), which is about detecting and selecting interesting events from the event cloud including the detection of event relationships [dro09].

---

[6] Drools: `http://www.jboss.org/drools` (accessed 24-05-2010).

[7] JBoss JBoss Enterprise BRMS: `http://www.jboss.com/products/platforms/brms/` (accessed 24-05-2010).

Drools Flow is a workflow or process engine that allows integrating processes and rules, whereas flow charts are utilized to describe the order of steps to be executed. While processes and rules are usually considered two different paradigms for defining business logic, Drools Flow provides an advanced integration of processes and rules. Therefore rules can be used to define parts of the business logic. Furthermore it features a generic process engine supporting multiple process languages such as the RulesFlow language from the Drools project or WS-BPEL[8] - a standard targeted towards web service orchestration [dro09, Bro09].

Drools itself does not integrate web services or data from remote systems, however the JBoss Enterprise SOA Platform[9] integrates JBoss Rules - the JBoss version of the Drools rules engine - into a platform for finding, integrating and orchestrating SOA business services, enterprise applications, and other IT assets into automated business processes [jbo10].

The Drools platform illustrates how rule engines are leveraged in enterprises and thus unveils therefor needed capabilities of a rule-based system. Hence it shows valuable features the Rules module could implement for Drupal to be able to serve the needs of organizations in order to foster the adoption of Drupal in enterprises and organizations, e.g., by providing advanced integration of processes and rules like Drools Flow. Related to that Paschke, A. and Kozlenkov, A. [PK08] show a rule-based approach to describe service-oriented business processes and implement a distributed execution middleware for rule-based process specifications with the help of reaction rules, whereas they conclude that the realization of business processes by means of rules provides an expressive orchestration language.

[8]WS-BPEL: `http://www.oasis-open.org/committees/wsbpel/` (accessed 24-05-2010).

[9]JBoss Enterprise SOA Platform: `http://www.jboss.com/products/platforms/soa` (accessed 01-06-2010)

3.3 Rule engines in Content Management Systems

As far as we can see rule engines appear to be mostly implemented in Content Management Systems with a focus on a flexible workflow management, however popular PHP-based open source systems like Joomla[10], WordPress[11] or TYPO3[12] do not feature a comparable rule engine as of the writing.

Alfresco[13] - a Java based enterprise Content Management System - features an interface for defining business rules. The system provides some built-in rules, but also allows the creation of custom-tailored business rules without requiring any programming expertise. For example the rules can be leveraged to automatically organize or version documents, send notifications to specific people or to dynamically add properties to a document. Apart from that also Alfresco's built in workflow component is driven by the rule system.

Alfresco's business rules are defined per space - Alfresco's notion of folders - and consist of a set of configured actions and conditions applied to documents. The execution of a rule may be triggered on one of the available events - being *Inbound*, *Outbound* or *Update* - whereas the events relate to a document and the space of the rule [Sha09].

Plone[14] is a flexible, open source content management solution written in Python, whereas Plone 3 ships with a rules engine called *Content rules*, which allows one to associate automatic actions to site editing events. The system makes use of rules that are combinations of triggers, conditions and actions to be executed when users do something on the site, whereas a rule may contain multiple conditions and actions. The rules can be configured with the help of a global graphical management interface, however for a rule to actually apply it needs to be assigned to one ore more Plone folders.

[10]Joomla: http://www.joomla.org (accessed 01-06-2010)
[11]WordPress: http://wordpress.org/ (accessed 01-06-2010)
[12]TYPO3: http://typo3.org (accessed 01-06-2010)
[13]Alfresco: http://www.alfresco.com (accessed 01-06-2010)
[14]Plone: http://plone.org/ (accessed 01-06-2010)

Plone provides a set of usable conditions, actions and triggers, but the system is extensible as well. An example that one could configure with the provided set of conditions and actions would be a rule that sends an e-mail to a specified mail address any time a page is modified [plo, KSA+09].

Sitecore CMS[15]: The Sitecore Content Management System is a product built upon the .NET framework. It comes with a rules engine and provides a graphical configuration interface for it. The rules may be configured with the help of some provided actions and conditions, whereas it is possible to extend the solution with custom conditions and actions. Sitecore uses event handlers to invoke the rules, e.g., when an item is inserted or saved. Apart from that the rules engine drives a feature called conditional rendering, which allows runtime manipulation of presentation components by the mean of rules. For that special Conditional Rendering Actions and Conditional Rendering Conditions can be utilized in Conditional Rendering Rules [sit09].

All inspected rule engines utilized in Content Mangagement Systems as well as Drupal's Rules module work similarly, such that users may use it to define custom reactions that are executed based upon some events or triggers when the specified conditions are met. Only the Sitecore CMS rules engine is additionally utilized for implementing a way of conditional rendering. Alfresco's and Plone's rule engine both have in common that rules are assigned and apply to one or more folders (or spaces in Alfresco's terminology) only, such that the rules can be easily leveraged to adapt the system per folder. However none of the systems is able to work across system boundaries, nor is support for a rule-based interaction with remote systems included.

[15]Sitecore CMS: `http://www.sitecore.net/en/Products/Sitecore-CMS.aspx` (accessed 01-06-2010)

Chapter 4

Objectives

In order to enhance the existing Rules module for Drupal introduced in section 2.3, this chapter discusses the objectives of our work. Each objective is explained in detail and reasons for its relevance are given.

O1: Bring Rules to the web

To cope with the emergence of web applications and to enable users to build upon available web services the module should be able to communicate with remote sites across system boundaries. Thus as a first step the rule-based invocation of web services should be possible for users without any additional programming to be required. Next to provide enhanced communication between several Drupal powered web sites the module should offer means to simply react on events occurring on remote Drupal sites. For that both the server and the client side should be completely rule driven in order to achieve great flexibility.

O2: Integrate WS* web services

Enterprise applications often rely on the principles of the Service Oriented Architecture (SOA) for providing their software services within and across organizational boundaries [NRD06]. As noted in section 2.2, the introduction of SOA, usually WS* based web services are leveraged to implement the Service Oriented Architecture. Currently the Services module presented in section 2.3 offers means to provide SOAP

services with Drupal, however there is currently no known way to easily invoke WS* based web services without the need of programming. Thus enabling the Rules module to support the rule-based invocation of any WS* based web services helps to foster the use of Drupal in the Service Oriented Architecture and so in enterprises.

O3: Consider significant use cases

For the enhanced module being useful in practice, significant use cases should be considered from the beginning. For that we have identified the following possible use cases:

1. *Distributed content deployment*
   Site content of a specific content type created on a Drupal site is replicated to another Drupal site by means of rules, whereas all fields of the content type shall be accessible.

2. *Taxonomy sharing*
   A taxonomy created on a Drupal site is replicated to another site by means of rules, such that both sites can make use of the same taxonomy.

3. *A workflow based review process*
   Writers create the content, which has to be reviewed afterwards. Then a list of content, which is to be reviewed, is shown to reviewers. They can approve the content or set it to the state "needs work". A list of content that needs work is shown to writers, so they can overhaul content which needs work and set it to "needs review" again. Once content is approved by a reviewer, it is published automatically. Also all changes to the content are logged, so that all changes are visible to reviewers.

4. *Automated publishing and expiration of content*
   Content can be configured to be automatically published or expired at a given time. Optionally content can be published manually too. In this case it is still possible to schedule automatic expiration of the content. When content expires, its publication is revoked.

O4: Be extensible and reusable

As described in section 2.3 the Rules module is already extensible such that Drupal modules may provide further events, conditions, actions, default rule configurations or data types. Apart from that any parts of a rule configuration, which are conditions, actions but also logical conjunctions, should be separate usable components. Next modules should be able to provide new components, thus extending the *rule language*. For one thing that warrants optimal reusability of module provided rules integration, e.g., could modules reuse conditions while not making use of any other rule-related functionality. For another thing the Rules module could itself be split up into its core components and more sophisticated language constructs. This is valuable as it might become necessary for a possible future inclusion of the module's core components in Drupal core.

O5: Prepare for decent user interfaces

While a user interface (UI) itself is not part of this work, we should prepare for decent user interfaces being developed later on. That means in particular that enough metadata has to be available at configuration time, such that meaningful options can be provided to the user. For components being really reusable on its own (see Objective 5), also the user interface of single components should be reusable and any for that necessary preparations should be taken into account.

CHAPTER 5

Realization

This chapter starts with an analysis of the objectives presented in the last chapter. Based upon this analysis and the objectives the module architecture is revised and elaborated in detail. This forms the foundation for the actual implementation, for which we provide an overview and elaborate on important aspects.

5.1 Analysis

Working with arbitrary data

In order to be able to work with any remote data source as required for Objective 1 and Objective 2, we need a unified way to deal with arbitrary, possible being remote, data. In particular for the data to be actually usable, we need means to traverse through the data and to access specific data properties, regardless in which format the data is represented. Based on that we need actions and conditions to perform desired operations like comparing data and modifying data.
To achieve that we could generate an appropriate condition or action for each data operation and for each data property of all available data sources. However, already on Drupal sites having a lot of fields, this would generate quite a bunch of conditions and actions - and even a lot more once remote sources are used - which the user would have to choose from. In order to make it easier for the user to overview the available conditions and actions, we think it is better to provide one generic condition or action per operation to be performed on a data property, and

to allow the user to configure the data property on which to perform the operation on.

For being able to provide a decent user interface later on as stated by Objective 5, we need some information about data we work with, so we can use this information to provide a user interface for the user. Therefore we need to know not only about the data structure at design time, but also we need to be able to show the available data properties to the user designing the rules, such that the user is aware of the purpose and the meaning of the data he works with. For that a label and descriptive sentence should be available for each data property.
Also we need to know about the data type at design time in order to be able to provide valuable operations one can perform on a property. E.g., for comparing numeric data values with a user configured value, we want to offer different comparison operators than for comparing textual data values. Also, to provide a decent user interface, we need suiting input forms for user provided data values dependent on the data type.

Specifying argument values

As described in section 2.3 the Rules module for Drupal 6 features a flexible input evaluation system, which allows users to make use of replacement tokens provided by the Token module. As the Token module is very popular [DrU10a] there are manifold replacement patterns available covering nearly all data available to Drupal. This and the opportunity to use token replacements in all the textual input elements used during condition or action configuration, enables one to specify dynamic data values as argument values by using token replacements as shown in figure 5.1.

While this way of using replacement patterns for specifying dynamic data values is working fine for textual arguments, it doesn't work in case the configuration form makes use of different input elements for specifying input values, e.g., there is no way to enter a replacement token when the configuration form uses checkboxes, radio buttons or a select list for specifying a value. However the use of improved data input widgets is critical for providing a decent user experience.

Apart from that Rules currently supports dynamic argument values by allowing the user to select one of the available variables of a matching data type, as one can see in figure 5.1 for the "Content" argument. Whether this argument configuration

Editing action *Schedule publishing of a node*

Label: *

Schedule publishing of a node

Customize the label for this action.

Arguments configuration

Content:

created content ▼

▷ Token replacement patterns

▷ PHP Evaluation

Identifier: *

node_[node:nid]

User provided string to identify the task. Existing tasks for this rule set with the same identifier will be replaced.

Scheduled evaluation date: *

[node:field_datefield-value]

Format: *2010-05-11 12:40:02* or other values in GMT known by the PHP strtotime() function like "+1 day".
You may also enter a timestamp in GMT. E.g. use

```
<?php echo time() + 86400 * 1; ?>
```

together with the PHP input evalutor to specify a date one day after the evaluation time.

Weight:

0 ▼

Adjust the weight to customize the ordering of actions.

Save Delete

Figure 5.1: An action configuration showing the use of token replacements for specifying a dynamic data value for the "Scheduled evaluation date" argument [DrR09a].

mode or an input form for specifying concrete values is used, depends on the data type needed by the argument to be configured. Thus Rules automatically provides the select box of available variables for arguments of the type *node*, but uses an input form for *date* arguments.
However in some cases it would be helpful to be able to work with variables of a certain type, but to still have the possibility to specify concrete values on design time, if preferable. In history the lack of providing these two input modes at the same time for the same data type has lead to extensive discussion of the preferable configuration mode for certain data types like taxonomy terms [DrR09b] or the flags, provided by the Flag module[1] [DrFb].

Thus to solve both problems, we need to have form widgets that ease inputting concrete values on the one hand, but on the other hand we need the flexibility to choose the specification of dynamic values by configuring a source for the data value. From now on we refer to these two configuration modes as *direct data input* and *data selection*.

In order to be able to support both configuration modes for parameters, we need to know about all existing parameters of a condition or action. However for the existing Rules module the explicit description of a needed argument is optional if direct data input is used for a parameter [DRD]. Thus for the enhanced module, the explicit description of all parameters has to be required.

Selecting data sources

While using token replacements as a way of specifying dynamic values for arguments as shown in figure 5.1 is working for textual arguments, it poses problems for other data types as the Token module is designed for offering *textual* replacement tokens. So usually the Token module formats the data in a human readable way, e.g., by converting timestamps to readable dates or by formatting numbers to be grouped by thousands. However for the result being usable as an argument value, we are not interested in the textual token of the data, we need the plain, unformatted data value. Next to provide a data selection mode usable for any parameter, it needs to support

[1] Flag module: `http://drupal.org/project/flag` (accessed 04-05-2010).

non-textual data like entities too and we need metadata about the data type of the selected data, so that we are able to restrict the selectable data items to items of the data type needed by the argument in question only, e.g. restricted to numeric values or to comment entities.

Therefore we need a way for selecting arbitrary data sources, similar to the Token system, but instead of being tied to textual replacements, the system has to return plain data values and provide metadata about the data in question. For the simple reason that selectors used by the Token system like `[node:title]` are already known by Drupal users, it seems natural to utilize the same way of specifying selectors for selecting arbitrary data sources for configuring arguments.

Avoid PHP evaluation

In the current version of the Rules module one needs to make use of the provided PHP input evaluator to achieve certain functionalities, e.g., to apply a custom offset to a dynamic date value or to select a data source for CCK fields not having a text input element as form widget. However, the use of the PHP `eval()` function is considered dangerous and could easily lead to security vulnerabilities [vD09], thus if possible it should be avoided. Next letting site builders, usually not familiar with PHP, write PHP code is always risky and could quickly lead to fatal errors or unexpected side effects.
Thus in order to enable users to get along without PHP evaluation, we have to ensure that the common usage scenarios are supported *without* the use of any PHP input evaluation making use of the `eval()` function.

Improve the export format

As already mentioned the existing Rules module allows exporting of created rules. For that the module basically generates some PHP code that re-creates the rules once the code is evaluated during import. That way the exported code fulfills his purpose, but it is not easy to read.

However an easy to read export format would be valuable as it could even serve as a simple way to document created rules. Next when the exported rules are used by

modules to provide default configurations in order to maintain rule configurations in code, a better readable export format would even help to make changes to the exported rules more visible - provided a version control system is used to track the code changes. Furthermore to avoid the use of PHP evaluation - as requested in the preceding section - importing rule configurations should not require the PHP evaluation of the exported rules.

Multi-valued data

As described in section 2.3 the Rules module features an extensible data type system allowing modules to add support for their data types. However the system is not capable of dealing with variables representing multiple values, but it turned out that the capability to support looping and generic data lists is an often requested feature [DrR]. Indeed it would be useful in a lot of cases, e.g., consider the possibility to loop over the list of comments that have been posted to a node, or just over the taxonomy terms of a node in order to apply an action to each of the items. Obviously the uses for this feature are manifold, thus it should be integrated into the enhanced module.

Object-oriented programming

In order to fulfill Objective 4 of being extensible and reusable and to ease supporting sophisticated features like looping and generic data lists, leveraging object-oriented programming could help creating easily usable APIs, while ensuring good code encapsulation and reusability. Also with Drupal 7 requiring PHP 5.2 or higher [Drub] the object oriented improvements of PHP 5 can now be fully utilized, paving the way for an object oriented module design.

5.2 Architecture

In this section we present the architecture of the enhanced Rules module based upon the preceding analysis, such that the objectives presented in chapter 4 are fulfilled. We start with an architectural overview and outline the structure of rule configurations. Then in order to enable remote Drupal sites to react on events occurring, we introduce the concept of *web hooks*. Lastly we show how we utilize the foundational module

architecture to integrate remote systems in a way we enable rule-based web service invocations and reactions upon remote events with the help of reaction rules.

Architectural overview

As described previously in section 2.3 with Drupal 7 the concept of *entities* has been introduced, which is the start of a common API for different Drupal data objects being nodes but also user accounts, taxonomy terms, comments and others. However the entity API of Drupal 7 does not yet provide unified ways to create, save or delete entities. However this was topic of discussion for Drupal 8 at the recently held core developer summit in San Francisco [cor10a].

Entity metadata

In order to be able to perform operations like creating, updating or deleting for any entity in Drupal 7, we need to create an abstraction layer that provides us with the needed unique interface. To obtain support for the existing core entities, we have to provide some code translating between the introduced interface and the existing API of each entity - what can be thought of entity metadata.
As analyzed previously in section 5.1 the metadata also has to include information about the data properties available, such that we can utilize each entity as data source for argument configuration via data selection. In order to be able to update a data source, we also support the optional annotation of metadata for updating the value of a certain data property. That way the property metadata helps us to hide the differences between simple object properties and fields not only for read access, but also for write access.
Furthermore to allow any module to utilize the established uniform interface, we provide it as a separate module, called *Entity metadata*. That way the module serves as a simple abstraction hiding the differences between the APIs of Drupal's entity types, which could be leveraged by a lot of different modules. Thus any module providing an entity would have to provide metadata only once to be integrated with all modules building upon the uniform interface. In turn a module building upon it automatically benefits when modules provide metadata in order to integrate with

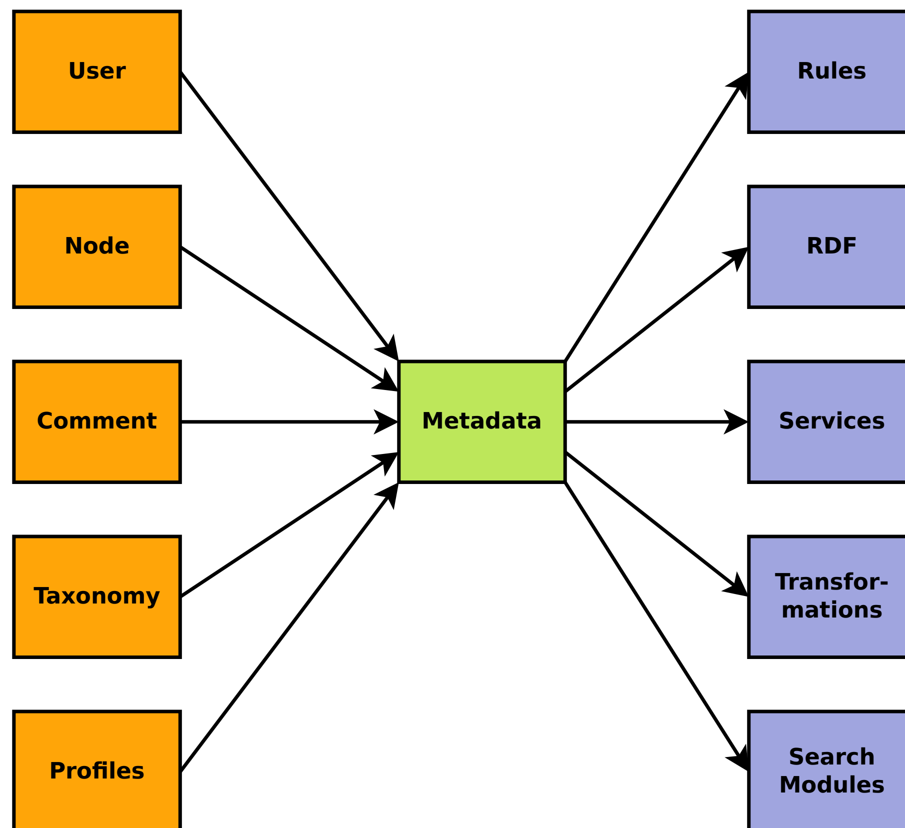

Figure 5.2: The design of the Entity metadata module - showing entities providing metadata on the left side, and possible modules utilizing the provided API based on the metadata on the right side.

another module - as seen in figure 5.2.

Note that the design of the Entity metadata module has been greatly inspired by the Fieldtool module[2] created by Jakob Petsovits, which already provides metadata about the properties of Drupal 6's data objects [DrFa, Pet09]. However the Entity metadata module takes the idea of Fieldtool to Drupal 7's entities and extends it with metadata for dealing with the entities itself.

Now the Rules module can utilize the uniform interface for entities provided by the Entity metadata module to generate Rules integration, such as actions to

[2]Fieldtool module: `http://drupal.org/project/fieldtool` (accessed 07-05-2010).

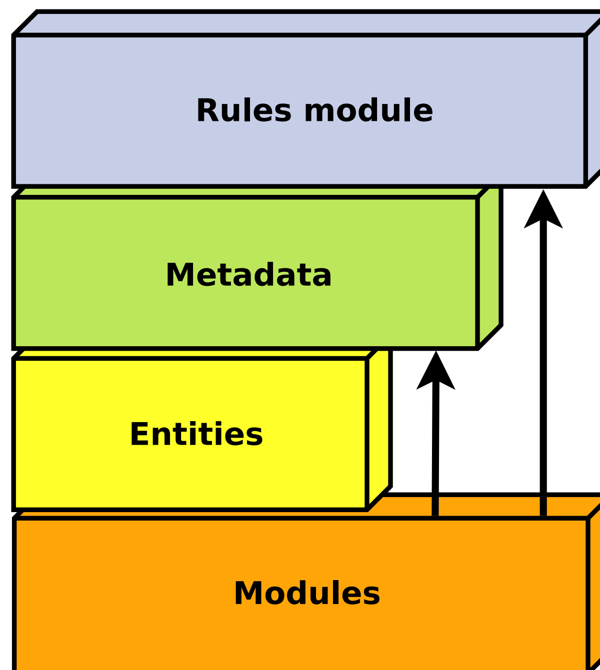

Figure 5.3: Architectural overview of the enhanced Rules module. For integrating with Rules any Drupal module may provide a new entity type, metadata for entities and non-entity data structures, or just directly new events, conditions and actions.

load, save, create and delete entities. Next it can utilize the metadata about the available entity properties, such that those are available as possible data sources for argument configuration. Also we can leverage this information by providing Rules integration that allows the comparison and modification of data properties based upon the property metadata. Furthermore to be able to make use of this functionality for non-entity data structures too, the Rules module would need a way to allow modules to provide metadata for their non-entity data structures. In addition to that modules should be still able to provide their own custom conditions and actions to Rules as it is already the case for the existing Rules module.
Figure 5.3 shows the resulting architecture of the module. In order to integrate with Rules any Drupal module may provide a new entity type, metadata for entities and non-entity data structures, or just directly new events, conditions and actions.

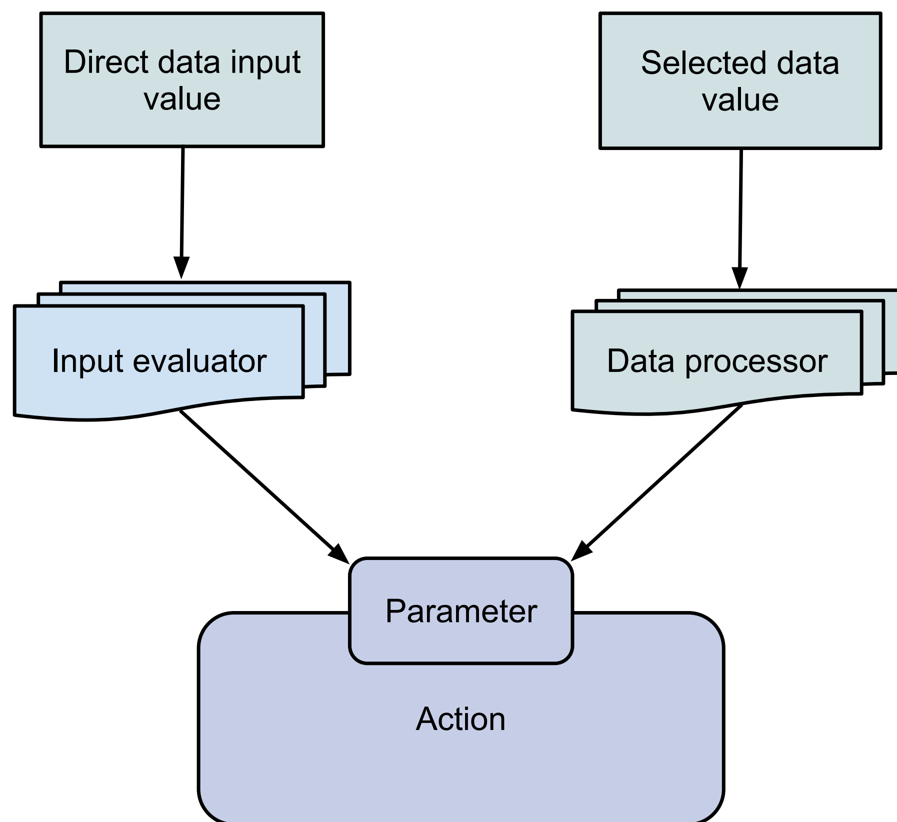

Figure 5.4: Overview over the possible argument configuration modes: Either data selection, together with an arbitrary number of data processors, or direct data input together with input evaluators can be utilized.

Rules data processor

As analyzed in the preceding section 5.1, argument values can be specified by using data selection or direct data input. For the latter we already have the input evaluation system in place, which enables users to easily integrate token replacements or even PHP code snippets. However in order to enable users to get along without PHP evaluation in common usage scenarios, we introduce the more general concept of Rules data processors, which process the argument value in a certain way on evaluation time. Therefore we can not only implement the input evaluators as a certain kind of data processors, but moreover we can leverage them to process any selected data source. This enables us to implement common needed processing tasks, like moving a date by a certain period of time or incrementing a number by one,

without having to rely on evaluating custom PHP code.
To achieve that, any module may provide further data processors as it is the case for input evaluators. Also the processor has to provide a configuration form that automatically appears during argument configuration, such that users are able to easily utilize any data processor.
Finally figure 5.4 provides an overview over the possible argument configuration modes. Either data selection, optionally together with an arbitrary number of data processors, is leveraged, or direct data input is utilized, which allows the use of input evaluators.

Web hooks

To enable other Drupal sites to react on the events occurring on Drupal site, we allow users to provide *web hooks*. A web hook can be described as followed: "A WebHook is an HTTP callback: an HTTP POST that occurs when something happens; a simple event-notification via HTTP POST." [Web]. Thus any site can subscribe to a published web hook and is so notified about the occurring event.
In order to provide web hooks with Drupal we build upon the REST server provided by the Services module, which we introduced in section 2.3. We use the module to implement the web hook, but also to provide RESTful resources for Drupal's data. That way the event-consuming client is able to fetch or even to update any additional data using the REST server. Additionally we create a service providing access to the available metadata, such that it can be utilized by Drupal based clients in order to provide a decent user interface for configuring rules that react on the event. For the actual invocation of the web hook, we provide an action, such that web hooks can be invoked rule-based.
Figure 5.5 shows how web hooks fit into the base architecture of the module. As seen in the figure the Services module also builds upon the metadata, such that the data exposed via the REST server is conform to the metadata describing it. Also, by leveraging the unified interface provided by the Entity metadata module, it is ensured that each entity supported via this interface is also available for use with Rules web hooks.

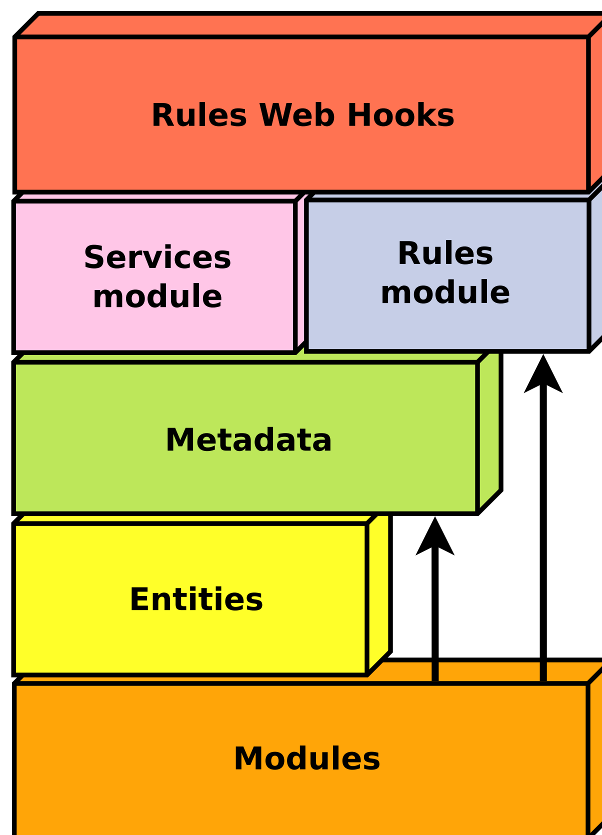

Figure 5.5: The architectural overview extended to include Rules web hooks, which build upon the Rules and Services modules.

Integrate remote systems

As required for Objective 1 and Objective 2 we need to be able to integrate with Rules web hooks, which are based on REST services as described in the preceding section, and to rule-based invoke WS* based web services, which have been introduced in section 2.2. Thus to achieve the integration with possible any kind of remote system, we create so called *remote proxies*, which help us to integrate any remote system represented as a remote proxy into Rules. As seen in figure 5.6 Rules can so incorporate support for remote systems just by building upon the remote proxies, while those may support various remote system endpoint types - in particular we need to support REST and SOAP based web services as well as Rules web hooks. As usual, we make it possible for modules to extend the solution by adding support for

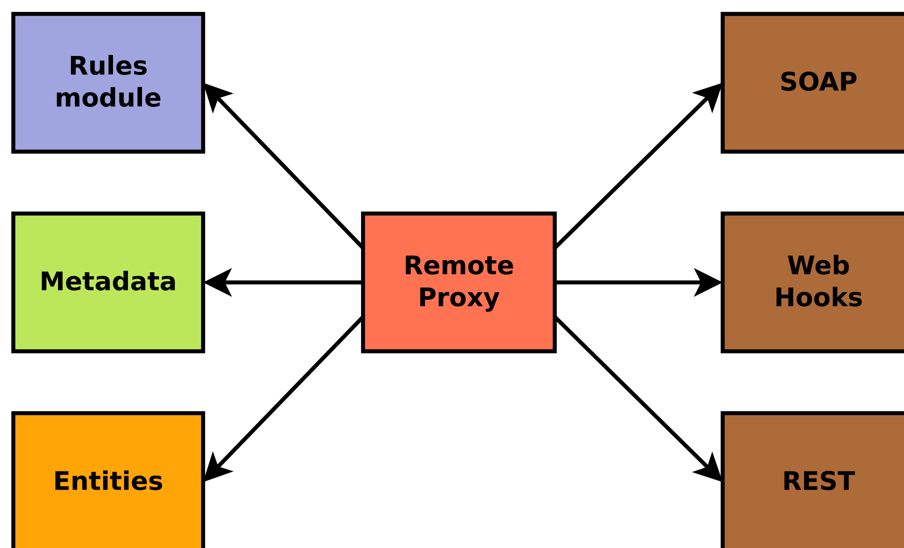

Figure 5.6: The role of remote proxies for integrating any kind of remote system into Rules by supporting various endpoint types - such as REST or SOAP based web services or Rules web hooks.

new endpoint types.

To allow users to easily utilize this system, we need to prepare for a decent user interface that allows users to define remote sites, which are built upon a certain endpoint type and include various settings, such as the service endpoint URL and any endpoint type specific settings. Hence this remote site definition acts together with the endpoint implementation as remote proxy and so allows us to generate the appropriate Rules integration, such as added remote events and actions for invoking specific services. In turn the user now can easily make use of this Rules integration just by configuring rules as usual.

As noted in the analysis in section 5.1 - we need the ability to deal with arbitrary data. In order to integrate the remote data seamlessly into the system, we need to have metadata available for it, such that the data can be utilized as possible data source for data selection. Apart from that we leverage the entity API for integrating remote data, so we are able to reuse the generic Rules integration for entities, e.g., to rule-based fetch a specific remote entity, but potentially also to be able to update the entities via the underlying services, such as the REST server provided by the Rules

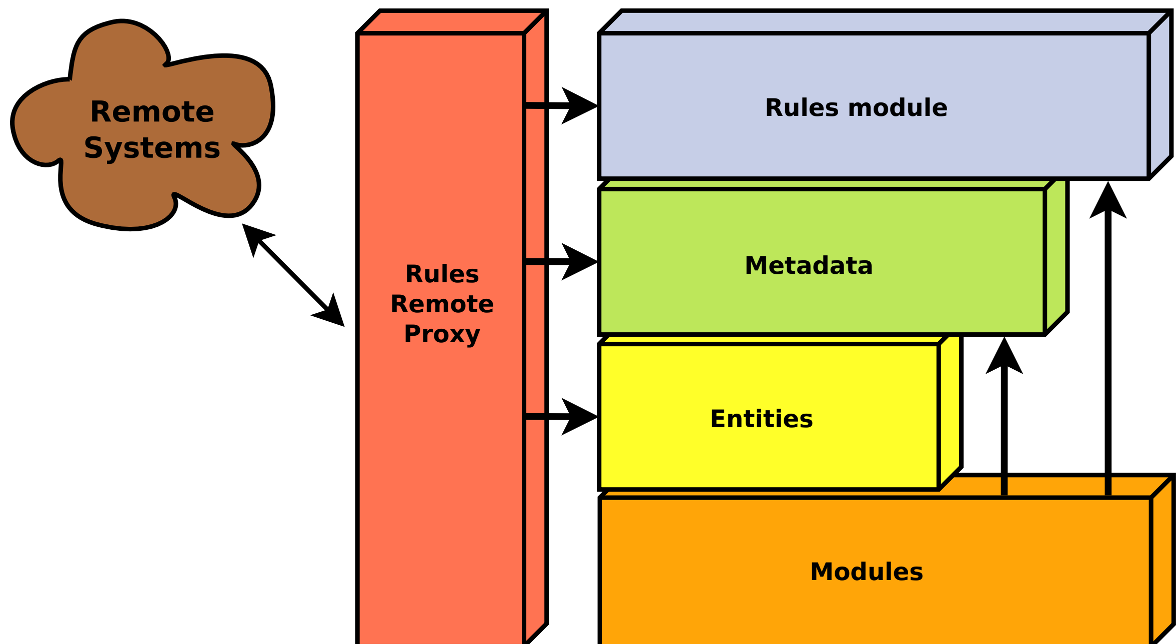

Figure 5.7: The extended base module architecture including remote proxies. A remote proxy may provide new entity types, non-entity data structures, metadata as well as events, conditions and actions to the system.

web hooks system. That way it is ensured that even updating remote data items would work exactly the same way as for local data and the user is able to leverage the usual tools built upon the entity API, respectively the unified interface provided by the Entity metadata module, also for remote data. In particular this is reasonable for integrating RESTful services, as each resource type can be provided as a new entity type to the system with the CRUD operations available as offered by the service. However, data items of remote services can be also integrated as non-entity data structures, such that metadata for the remote data can be provided but no CRUD operations need to be available.

Figure 5.7 shows how remote proxies are integrated with the base module architecture. Each remote endpoint type provides together with the remote site definition new entity types and metadata to the system, but also metadata about any non-entity data structures, remote events, as well as condition and actions for directly integrating any web service operation. That way we obtain a flexible framework that allows

site builders to easily integrate remote systems via a provided remote endpoint type, while modules may extend the solution by adding support for new endpoint types.

Rule configurations

A rule configuration represents an actual instance of a rule, which typically consists of an arbitrary number of conditions and actions. In case of a reaction rule, there is also an arbitrary number of events assigned to a rule configuration. In order to fulfill Objective 4 of being extensible and reusable any part of a rule configuration has to be a separate usable component, thus we introduce the concepts of plugins, elements and configurations for the Rules module:

Plugins represent different types of rule elements, which build rule configurations. Consequently there is a plugin for actions and conditions, but also for more sophisticated constructs like rule sets, rules, reaction rules, logical conjunctions and loops - while modules may introduce new plugins, hence extending the rule language.

Elements are configured and so usable plugins. However an element is often not usable on its own, but part of a bigger rule configuration, consisting of multiple rule elements.

Configurations are elements that are usable on their own, hence they are also based upon a certain plugin. The element representing the configuration forms the root element, which might contain further elements, e.g., a rule usually contains some conditions and actions, where each of those is a rule element being a configured action or condition plugin, while the root of the configuration is an element based upon the rule plugin.

This architecture ensures that the module is extensible by new plugins and makes each plugin not only reusable on its own, but moreover it allows any additional plugins provided by modules to be leveraged in rule configurations.

Configurations

As a configuration is an element that is usable on its own, we need to provide means for storing and loading configurations. To achieve that we implement a Rules configuration as a new entity type as that allows us to leverage the existing API for loading entities in Drupal core, but apart from that this enables us to make the configurations fieldable. Having fieldable configurations allows not only users to assign any field to a Rules configuration - e.g., to add a simple textfield for describing rule configurations - but moreover it enables us to programmatically create default fields and utilize those. For instance, currently the Rules module implements a simple tagging system in order to be able to categorize rule configurations. However with Drupal 7's field API and fieldable rule configurations we could build upon the existing taxonomy system instead, which makes it possible to tag any entity just by assigning a taxonomy term reference field to it. As another advantage having configurations as entities would allow us to smoothly provide Rules integration for the configurations itself - thus enabling the Rules module to deal with its own rules, what is usually referred to as *rule reification*.

Plugins

In order to fulfill compatibility between plugins, each provided plugin is represented by a PHP class, which has to inherit from the class `RulesPlugin` , the provided base class for plugins . Further the plugins can be distinguished as being reusable either as action or as condition, thus implementing either the `RulesConditionInterface` or the `RulesActionInterface` . Apart from that there is the class `RulesContainerPlugin` , a base class for container plugins, which in distinction to other plugins may contain further rule elements. However an element, which is an instance of a container plugin, may only contain elements that are based on plugins, which are marked to be embeddable and suit to the containers interface - being the action or the condition interface. Additionally an embeddable plugin may further restrict the plugins it may be embedded into, e.g., the `Rule` plugin may be only contained in rule sets. Table 5.1 provides an overview over the plugins provided by the Rules module and their properties.

The rule and the reaction rule plugins are sort of a special container plugin though.

Plugin	reusable as	Container	Embeddable
Action	Action	No	Yes
Condition	Condition	No	Yes
Logical OR	Condition	Yes	Yes
Logical AND	Condition	Yes	Yes
Loop	Action	Yes	Yes
Rule	Action	Yes	Yes
Rule set	Action	Yes	No
Reaction rule	Action	Yes	No

Table 5.1: The plugins provided by the Rules module and their properties.

They work as usual container embedding actions, but in addition to that we attach a single logical AND element to a rule, containing all conditions assigned to the rule. Then the rule is able to easily incorporate the result of the conditions, such that the rule fires only when the conditions are met. In contrast to the reaction rule plugin, the rule plugin is embeddable, such that rules may be embedded in a rule set element. On the other hand the reaction rule plugin extends the rule plugin to additionally allow the association of events in order to be be triggered, when one of the events occurs.
Apart from that the action and condition plugins are different to the other plugins, as each instance of the action or condition plugins is tied to a certain action or condition implementation provided by a module. Thus those plugins have to dynamically incorporate the actual condition or action implementation whenever necessary.

Variables

Each condition and action may specify multiple parameter, which the user configures at rule design time and are so passed to the action or condition implementation on evaluation time. As described previously in section 5.2 there are two configuration modes to specify the argument value of a parameter, direct data input and data selection, whereas the latter builds upon the variables available to the current evaluation state, such that the user is able to select the variable or a property of the variable as value to be passed as argument on evaluation time.

The variables available for configuration depend on the evaluation context, first off in case of a reaction rule that is triggered upon an occurring event, the event provides some variables associated to it. Other components that are built using a container plugin like rule sets, or condition sets - created either by using the logical OR or the logical AND plugin - may specify multiple variables they make use of. In turn those variables need to be passed to the component on evaluation time and are available to all contained rule elements. Apart from those variables any plugin implementing the action interface may provide new variables to the following rule elements, such that it is possible to fetch some new data with an action and make it available to the subsequent actions via a provided variable.
So far this variable system is nothing new, as it is already implemented that way in the current Rules module available for Drupal 6 [DrR09a]. However for the plugin system to be extensible, we need the plugins to completely control their behavior related to variables. In order to achieve that plugins have to implement the following methods:

`parameterInfo()`
: Returns information about the parameters needed by the element. Usually the argument values for those parameters are configured by the user via direct data input or data selection.

`argumentInfo()`
: In case the element represents a configuration, any parameter that has no configured argument value needs to be passed to the configuration once it is executed, thus this method returns information about the expected arguments upon execution.

`providesVariables()`
: Returns information about new variables that the element provides for any subsequent actions.

`stateVariables()`
: This method defines the variables that are available in the evaluation state for contained elements, thus it needs to be implemented only by container plugins. Mostly this method passes through variables that are available to the element

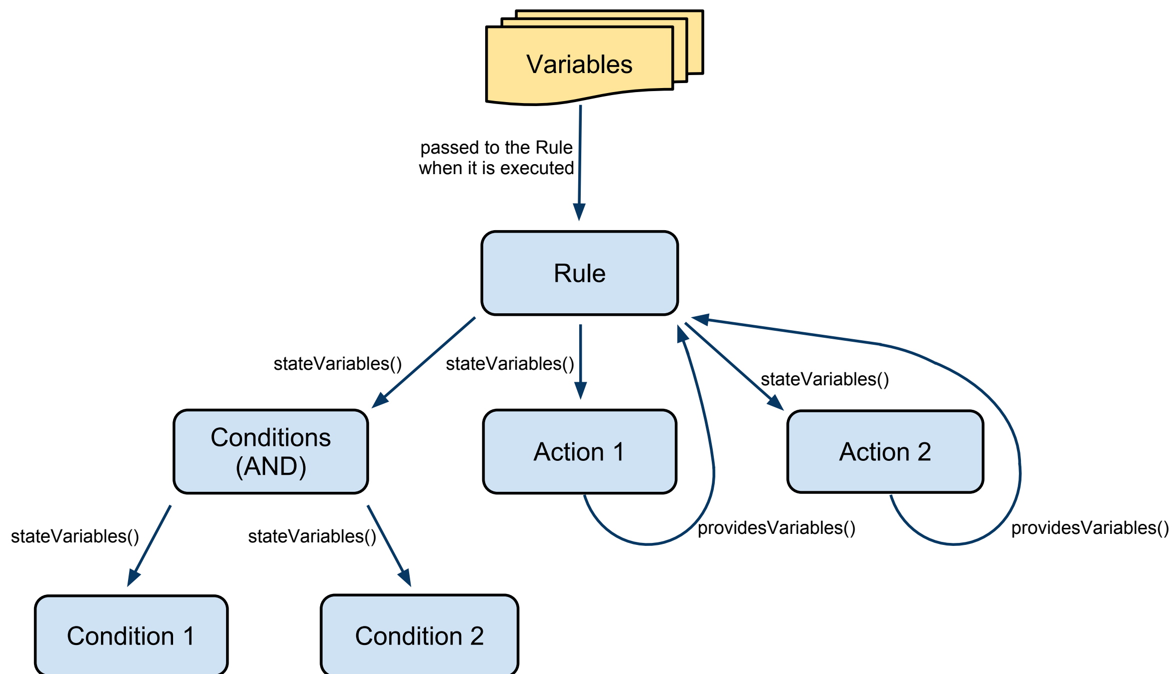

Figure 5.8: The propagation of available variables in a rule configuration.

itself, however in case a contained element is passed to the method, it has to return information about variables available in the state for the passed child element, which means it has to incorporate possible provided variables by actions evaluated before the passed child element.

`availableVariables()`
: Returns information about the variables that are available to be used by the element itself, e.g., to configure an argument value. Mostly this equals the `stateVariables()` provided by the parent element for the element - with the exception of the case that the element itself represents a configuration as then it would have to return information about the variables, which the configuration makes use of.

Note that all these methods have to work not only during evaluation, but moreover already on design time for each element of a configuration. This is necessary in order to provide a decent user interface that is aware of the available variables when users are designing the rules.

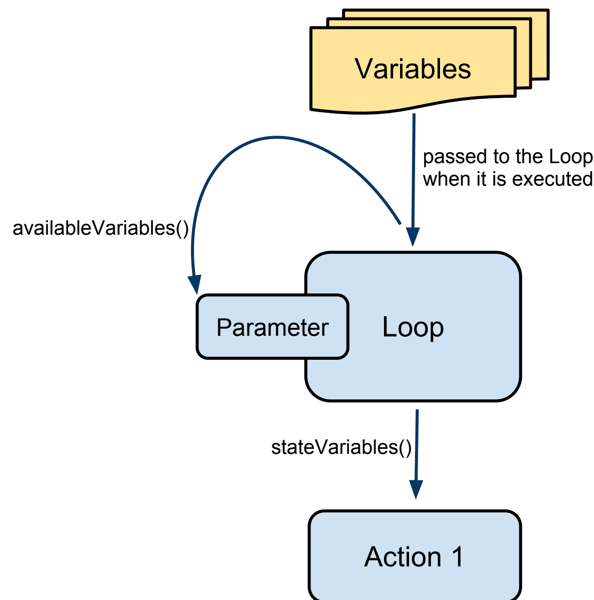

Figure 5.9: Available variables in a loop configuration making use of some specified variables. The state variables provided by the loop consist of the variables available to the loop and a new variable representing the current item of the list that is looped over.

Figure 5.8 shows how variables are propagated down the tree in a rule configuration. The rule configuration shown is making use of some specified variables, which have to be passed to the rule when it is executed. Then those variables are available in the evaluation state for all child elements of the rule. The *Conditions (AND)* element propagates the variables down the tree, so they are available for each of the conditions. Furthermore variables provided by an action are available to subsequent actions, thus the variables provided by *Action 1* are incorporated into the state variables for *Action 2*.

An interesting example in regard to the available state variables is the loop plugin. The loop plugin has a parameter, which is the list of items to be looped over and contains the actions that are to be executed for each list item. Consider a loop configuration making use of some specified variables, as seen in figure 5.9. As usual the parameter may be configured using the variables available to the element, what are the variables passed to the loop configuration. For instance the parameter could

be configured by selecting a list property of one of the variables as source for the argument value. Then the loop element loops over this list and provides the variables passed to the loop configuration together with a new variable, representing the current item of the list that is looped over, to the contained actions as available state variables. That way the contained actions are able to easily utilize the current list item just like any other variable that is available.

5.3 Foundational APIs

In order to implement the Rules module for Drupal 7, we created several foundational APIs, which form together with Drupal 7 the base for the actual implementation of the enhanced Rules module.

Extendable object faces

The Extendable object faces module[3], or shortly called faces, is an API module, which allows modules to extend objects. Thus it makes it possible to implement the Facade pattern in a modular way. Jan Bosch describes the Facade pattern as following: “The Facade design pattern is used to provide a single, integrated interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that simplifies the use of the subsystem.” [Bos95]. Hence with faces it is possible for any module to provide a subsystem that implements certain functionality, while the caller only needs to deal with the extended object, which is acting as Facade.
For that the faces API allows to:

- Extend an object by providing an extender class.
- Extend an object by some provided function callbacks, each implementing a method.
- Override any extended method with provided function callbacks or another extender class.

[3]Extendable object faces module: http://drupal.org/project/faces (accessed 12-05-2010).

- Lazy load needed include files upon method invocation.

Therefore the faces API provides not only a way to let modules extend an object, moreover it enables an object to easily lazy load parts of its implementation and allows modules to dynamically override implementations.

Entity CRUD API

As described in section 5.2 we make use of the entity API in Drupal core for Rules configurations. As noted before, the entity API of Drupal 7 does not yet provide ways to create, save or delete entities, thus in order to achieve this functionality for Rules configurations we have to implement the missing parts. For that, we have created the Entity CRUD API module[4], which builds upon the entity API in Drupal core and extends it such that it provides full CRUD functionality, standing for Create, Read, Update and Delete. By building this into a separate usable API we ensure that any module that wants to expose a new entity type is able to utilize the Entity CRUD API instead of having to implement the same functionality again. Also the module comes with test cases, which implement two sample entity types and use them to proof that the API is working correctly.

For a module to leverage the API, it just needs to define its database tables as usual by making use of `hook_schema()`[5] and to implement `hook_entity_info()`[6] to describe the provided entity types and to specify the usage of the controller class provided by the Entity CRUD API, whereas both hooks are already defined by Drupal core. Then for utilizing the CRUD functionality the provided functions `entity_save()`, `entity_create()`, `entity_delete` or `entity_delete_multiple()` may be used. Optionally, the controller also supports the use of custom classes for the created entity objects instead of using the default class of PHP objects - `stdClass` - as done by Drupal core, thus making it possible to leverage object oriented programming. The class to be used can be de-

[4]Entity CRUD API module: http://drupal.org/project/entity (accessed 12-05-2010).

[5] `hook_schema()`: http://api.drupal.org/api/function/hook_schema/7 (accessed 12-05-2010).

[6] `hook_entity_info()` : http://api.drupal.org/api/function/hook_entity_info/7 (accessed 12-05-2010).

clared by specifying the `entity class` key in `hook_entity_info()` . For that the entity API provides two base classes that can be utilized - `EntityDB` and `EntityDBExtendable`, whereas the latter basically equals the former but is extendable via the Extendable object faces API introduced in the preceding section.

Exportables

The API incorporates the concept of exportables introduced by the Chaos tool suite module [DrC] for Drupal 6 and takes it from the database level to work on the entity API level, such that any entity type can be easily made exportable if desired. In case an entity type is marked to be exportable, the controller automatically invokes a default hook when it is loading entities. That way modules may provide defaults in code, which are incorporated automatically when entities are loaded. Therefore it does not matter to the consuming code whether a loaded entity is stored in the database or is defined in code. Next the API provides some utility functions that help exporting entities to code, as well as a simple default implementation for exporting.

We leverage this functionality for the Rules configuration entity type by defining it to be exportable. Therefore we retain the capability of the existing Rules module to be exportable and to provide default configurations in code, while at the same time we leverage the existing API for loading entities provided by Drupal.

Entity metadata

As described in the architectural overview on page 40 the Entity metadata module[7] leverages metadata to provide a uniform interface hiding the differences between the APIs of Drupal's entity types and the various entity properties. For that the module provides a new hook `hook_entity_property_info()`, which modules may use to define metadata about entity properties. For establishing a uniform interface to perform operations like create, update and delete on entities the module introduces some new keys for the existing `hook_entity_info()`, such that modules may

[7] Entity metadata module: http://drupal.org/project/entity (accessed 12-05-2010).

specify function callbacks for performing a certain operation on an entity of a given type.

Data types

As noticed during the analysis in section 5.1, we need to know about the data type of properties on rule design time in order to be able to provide valuable operations one can perform on a property. For that the module defines a list of common data types modules should make use of for describing their data. That way modules working with properties can concentrate on these data types to provide meaningful support for the properties based upon their type, such that any data that can be mapped to a structure built of the known types is supported. As documented in the API documentation of the module, the Entity metadata module defines the following list of supported data types:

`text`
: A string value, which may contain any characters, represented as PHP string. If not specified else using the `sanitized` key, the text is assumed to be not sanitized for output. A function callback to sanitize the text value may be specified using the optional `sanitize` key.

`integer`
: A numeric value without a fractional component, represented using a PHP integer variable.

`decimal`
: A numeric value with an optional fractional component, represented as PHP float or integer.

`date`
: A date, specifying a date and a time using the UTC timezone, represented as an unix timestamp as PHP integer variable.

`duration`
: A value specifying a duration in seconds, represented as PHP integer.

`boolean`
: A boolean value, represented as a usual PHP boolean variable.

`uri`
: A value specifying a Uniform Resource Identifier (URI) [BL94] or a URL, represented as a PHP string variable.

`node, comment, user, taxonomy_term, file, ...`
: Any entity type known to Drupal is supported as a valid data type, while the entity is represented by its identifier. Optionally also the entity object may be used.

`struct`
: This as well as any else not known data type may be used to describe arbitrary data structures. For that additional metadata has to be specified using the `data info` key. The structure itself may be represented using an PHP array or object, such that it corresponds to the specified metadata.

`list, list<X>`
: A list of data values, optionally with a specified data type `X` for the list items, where `X` may be any other known data type. A list of values is represented as a numerically indexed PHP array.

Modules describe each entity property using a PHP array containing keys like `type` for describing the data type, `label` and `description`, as well as other keys as partially noted at the description of the data types. The full list of keys is available as part of the API documentation shipping with the module.

Data wrappers

In order to ease making use of the available metadata for entities and their properties, the module provides wrapper classes, which may be used to wrap entities or possible any data structure, so that the unified interface established with the help of the provided metadata can be easily utilized. For that the wrapper provides suiting methods to invoke supported operations on entities, but also there are means to retrieve wrappers for the properties of a wrapped entity as well as to get and set the property's value. As an entity property might reference to another entity, a retrieved property wrapper can wrap another entity, which itself has entity properties that in

turn can be accessed the same way. Therefore the API supports chained usage, as demonstrated by the following usage examples:

```
// Create the initial wrapper object for a node entity.
$wrapper = entity_metadata_wrapper('node', $node);

// Get the node author's mail address.
$wrapper->author->mail->value();

// Update the node author's mail address.
$wrapper->author->mail->set('name@example.com');

// Another way to update the mail address.
$wrapper->author->mail = 'name@example.com';

// Make the changes to the node author permanent.
$wrapper->author->save();

// Permanently delete the wrapped node entity.
$wrapper->delete();

// Article nodes have a taxonomy term reference
// field called 'field_tags'. Get the name of
// the first tag assigned to an article node.
$wrapper->field_tags[0]->name->value();
```

As seen in the last example, which is getting the name of the first tag assigned to a node, the metadata wrappers are also capable of dealing with lists of data in a practical way. In case a list item or an entity property that does not exist is accessed, the system throws an `EntityMetadataWrapperException`.

We leverage the demonstrated chained usage of the data wrappers in the Rules module in order to easily evaluate the data selectors used for selecting data sources for argument values. Thus a selector string like `node:author:mail` is easily applied to a wrapper of a node object, such that the node author's mail address is retrieved.

5.4 Implementation

In this section we provide an overview over the implementation of the enhanced Rules module, which is based upon the foundational APIs we have created and described in the preceding section.

Code overview

The Rules modules leverages inheritance in order to improve code re-usage and to ease the creation of new plugins, which can just extend any existing plugin. Figure 5.10 shows the class hierarchy for the provided plugins as previously listed in table 5.1. A plugin visible in the diagram, but not listed previously, is the `RulesEventSet`. This plugin is used only internally to prepare and cache a rule set for each supported event, such that when the event occurs this rule set is leveraged to efficiently evaluate all reaction rules assigned to the occurred event by using cached data only. To achieve that the system also caches the information about the available actions, conditions, data types and plugins.

A plugin must be registered using `hook_rules_plugin_info()`, for which the providing class has to be specified. Additionally one may specify interfaces with associated implementations to extend elements, which are instances of this plugin, with the help of the Extendable object faces API introduced in section 5.3. However different to the faces API rule elements are extended automatically - based on the information specified in the hook - once they are constructed. This functionality is implemented by the `RulesExtendable` class by building upon the faces API. In turn the base class for plugins `RulesPlugin` is based upon the `RulesExtendable` class as seen in figure 5.10.

This extension mechanism is utilized to incorporate UI related functionality for each Rules plugin. To achieve that, the `RulesPluginUIInterface` has been defined, which in turn may be later implemented in order to provide a decent user interface. That way each rule component has its associated UI components based upon a defined interface, hence the UI related methods are known, as well as easily accessible for modules that might want to make use of it. Next the faces extension mechanism decouples the UI related code from the plugins itself and thus

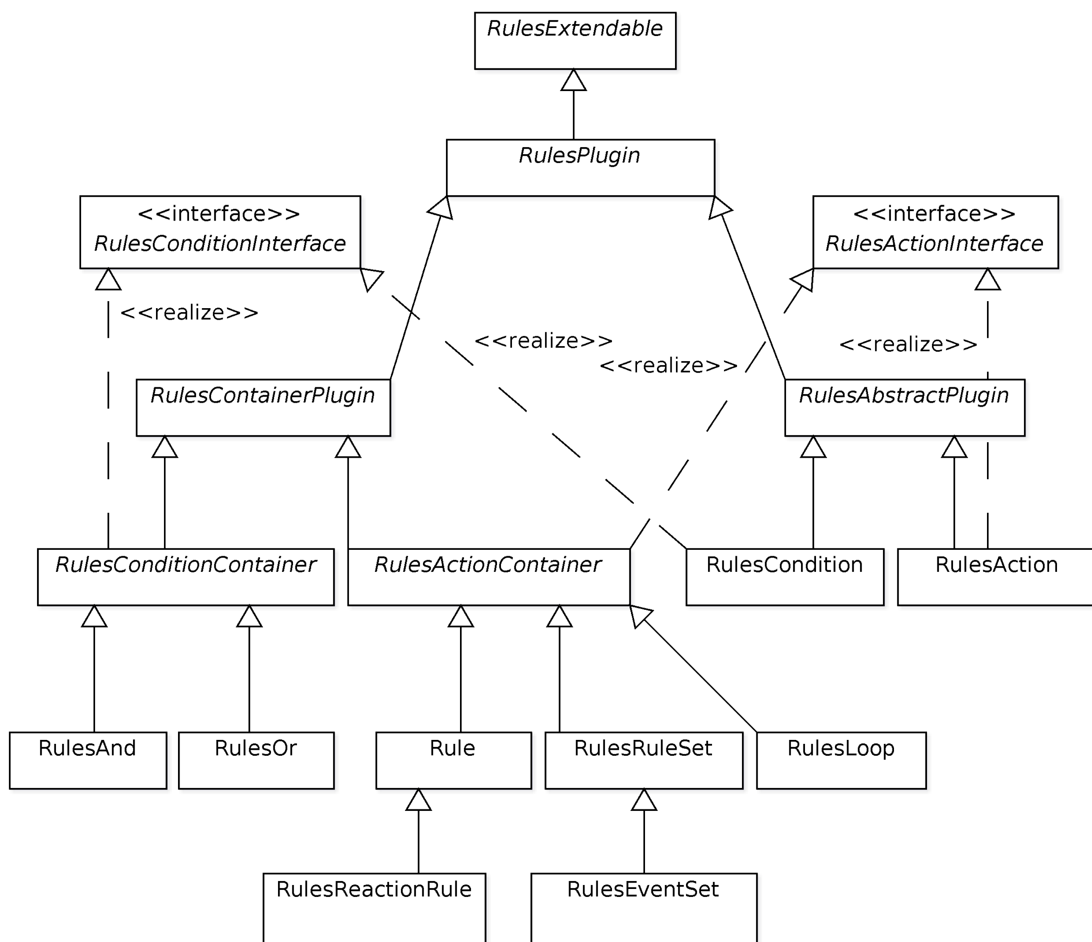

Figure 5.10: The class hierarchy of the provided plugin implementations. Each plugin inherits from the `RulesPlugin` class and implements either the `RulesConditionInterface` or the `RulesActionInterface`.

enables us to easily lazy-load it, such that the accompanied include files are only loaded whenever necessary. Next the mechanism could be used by modules to easily override parts of the UI or even to replace the UI with a completely new, improved version.

As the `RulesAction` and `RulesCondition` plugins are different to the other plugins - such that each instance of the action or condition plugin has to dynamically incorporate the actual condition or action implementation provided by a module - the base class `RulesAbstractPlugin` has been created in order to share the code necessary to implement this functionality between both classes, as seen in figure 5.10. To achieve that this base class makes use of the faces API to integrate the module provided function callbacks into the action or condition element. For extending the elements with the help of the faces API the `RulesPluginImplInterface` is defined, as well as the class `RulesAbstractPluginDefaults`, which implements the interface and acts as default implementation extending the action and condition plugin. However each function callback specified by the action or condition implementation overrides the provided default methods, such that the faces extension mechanism invokes the function callback instead. That way the provided module integration is seamlessly integrated and easily usable via the created rule element of an action or condition, while the lazy-loading capability of the faces API is utilized to lazy-load the include files containing the needed Rules integration.

Apart from the plugins the module allows for specfiying new data types by implementing `hook_rules_data_info()` e.g., for specifying metadata of data structures not being entities. Next there is `hook_rules_evaluator_info()` and `hook_rules_data_processor_info()`, which allow modules to declare provided input evaluators and data processors - whereby our work comes already with input evaluators for token replacements and PHP input evaluation as known from the existing Rules module. Apart from that we make use of input evaluators to transform relative specified URLs to absolute URLs and to support textual date inputs relative to the execution time as supported by the PHP `strtotime()` function like `+1 day`. Additionally we provide a data processor for applying custom offsets to dynamic date values as well as a general PHP data processor that allows the use of PHP code snippets for the custom processing of data values.

Utilizing rule configurations

The API of the rules plugin has been designed to work similarly to the new database layer of Drupal 7 [DrA10] in order to help developers to get used to the system. For the creation of new rule elements the module provides some useful functions, which act as factory for creating rule elements based upon the plugins as specified in `hook_rules_plugin_info()` . For instance, consider the following example reaction rule that is triggered upon an update of a Drupal node and just prints a message to the acting user, if the node is not published and of type *page*:

```
$rule = rules_reaction_rule();
$rule->event('node_update')
     ->condition(rules_condition('data_is', array('data:select' =>
         'node:status', 'value' => TRUE))->negate())
     ->condition('data_is', array('data:select' => 'node:type', '
         value' => 'page'))
     ->action('drupal_message', array('message' => 'A node has been
          updated.'))
     ->save();
```

Furthermore configurations created that way can easily be executed by using the `execute()` method, whereas the `argumentInfo()` method describes the arguments the configuration expects for execution. The following example creates a simple rule making use of a variable called *node*, for which an action modifies the node creation date to the day of yesterday:

```
$rule = rule(array('node' => array('type' => 'node')));
$rule->action('data_set', array('data:select' => 'node:created', '
    value' => '-1 day'));
$rule->execute($node);
```

As shown, the variables are passed to the `execute()` method as described, such that the rule is evaluated using the passed data. Also note that the changes to the node are automatically saved permanently by leveraging the Entity metadata for saving a node object.

Modules may make use of this API in order to build upon Rules configurations as suitable, so could a module not only build upon Rules by providing default reaction rules as already possible in Drupal 6, but moreover can the available rule components be reused separately. For instance it would be conceivable to reuse user configurable conditions acting upon predefined variables, such that the module only enables certain functionalities when the user-defined conditions are met.

Apart from that an easy usable API is of a great help for testing the module and provided Rules integration programmatically. The Rules module itself leverages the API to implement plenty of test cases in order to proof the provided Rules engine and the Rules integration for Drupal core is working correctly. Modules providing further Rules integration are encouraged to follow the lead and to write test cases ensuring their provided integration is working as excepted.

Exporting configurations

As described previously in section 5.3 we define the Rules configuration entity type to be exportable by making use of the Entity CRUD API, we have created. But as analyzed previously in section 5.1 we need to create an easy to read export format that does not require the use of PHP evaluation in order to import previously exported rules.

To achieve that we export Rules configurations by making use of the JSON format, a simple and language independent data interchange format, which is based on a subset of javascript. The format is lightweight, easy to read and Drupal has built in support to parse JSON based upon the JSON PHP extension [RR07, DrA10, Drua]. The following JSON code is an export of the reaction rule already used in the preceding section as example:

```
{ "rules_example" : {
    "PLUGIN" : "reaction rule",
    "REQUIRES" : [ "rules" ],
    "ON" : [ "node_update" ],
    "IF" : [
      { "NOT data_is" : { "data" : [ "node:status" ], "value" :
          true } },
```

```
        { "data_is" : { "data" : [ "node:type" ], "value" : "page"
            } }
      ],
      "DO" : [ { "drupal_message" : { "message" : "A node has been
          updated." } } ]
  }
}
```

The export generally consists of a JSON object with the names of the exported configurations as keys and the export of each configuration as values. The configuration export is described using another JSON object with some predefined keys describing the plugin, the modules required by the configuration, as well as an optional label. Further possible keys depend on the plugin, as seen in the example the reaction rule plugin makes use of the keys `ON`, `IF` and `DO` to describe the configured events, conditions and actions.

In order for the export of configurations to cope with the modular design of the Rules module each plugin needs to be able to define its own data structure to be exported, such that any plugin is supported. For that a plugin is identified by the key being the uppercase plugin name or the name of a action or condition implementation provided by a module, whereas the value assigned to the key describes the actual configuration of the element. As the exported data structure may vary by plugin, the proper interpretation of this value and the re-creation of the element configuration is up to the respective plugin class.

Web hooks

For the implementation of web hooks to follow the architecture as described in section 5.2 we build a module that allows the creation of web hooks, as well as the rule-based hook invocation with the help of an action. Thus for loading and saving web hook configurations, we build upon the Entity CRUD API and introduce a new, exportable entity type `rules_web_hook`. That way we do not have to reinvent the CRUD related functionality for web hooks, but obtain a hook for modules to provide default web hook configurations in code, and could even leverage the field API, such that the module could allow site builders to add their own fields to web hook configurations.

In order to specify a web hook a user or a module has to define at least a name and the variables, which are passed to the web hook upon invocation. The variables have to be described in the way as usual for the Rules module, such that the action provided for invoking web hooks is able to specify them as parameters and one can leverage the usual argument configuration of Rules for specifying the actual variable values. This approach makes it possible to react upon any event supported by Rules to provide a new web hook in situations, when the specified conditions are met. Furthermore the actual data provided to the notified clients upon web hook invocation can be easily controlled by specifying appropriate variables and argument values.

For providing the actual web services we build upon the Services module, as described in the architecture. However as of the time of the development there is no official Drupal 7 compatible version of the Services module available yet, but a development version created by Hugo Wetterberg[8]. Due to the lack of an official version, we have built upon the work of Hugo Wetterberg to build a module that exposes web hooks, whereby our module can be easily updated later on in order to create an official release, when an official version of the Services module is available. Analogously, there is no Drupal 7 compatible REST client module for Drupal 7 available yet, but again a development version of Hugo Wetterberg[9], which we are using in our work to communicate with REST servers. Fortunately, in the meanwhile the Services module developers have agreed upon building upon Hugo Wetterberg's work for a new, Drupal 7 compatible version 3.x of the Services module [Dru10b].

The Rules web hook module utilizes the Services module to provide the web hook configurations as resources, such that clients are able to retrieve the configuration details. Next it provides services for listing all available web hooks, to subscribe or unsubscribe from a specific web hook and to retrieve metadata about the used data structures. In order to subscribe to a web hook a client has to provide the URL of an HTTP callback and a subscription token. When a web hook is invoked the token is

[8]Services module Drupal 7 compatible development version of Hugo Wetterberg: `http://github.com/hugowetterberg/services` (accessed 21-05-2010).

[9]REST client module of Hugo Wetterberg, which has been renamed to 'http_client': `http://github.com/hugowetterberg/http_client` (accessed 21-05-2010).

used to generate a hash value that is sent together with the actual hook invocation arguments as notification to the subscribed client, so the client is able to proof the validity of the notification with the help of its token.

The service for retrieving metadata about the used data structures is necessary for the client to know about the structure of the used data at design time, such that the metadata can be utilized to provide a decent user interface for configuring rules reacting on invoked web hooks, what includes making use of remote data as data source for the data selection argument configuration mode. However any function callbacks - which are part of the metadata - to map the data to match the described data types cannot be applied by a client. As a consequence those function callbacks are applied in advance, such that the data structures returned by resources already match the described data structure. Therefore the metadata provided for clients can safely exclude this function callbacks.

As of now the provided RESTful services make use of the JSON format for data serialization, however as the module builds upon the Services module for providing web services, support for further data formats could be added via the API of the Services module. Similarly we rely on the Services module for handling authentication as the module supports pluggable authentication mechanisms [DrS]. As of the time of the development the used unofficial Drupal 7 version provides no authentication methods though, but that can be expected to change for any in future released version of the Services module. For the time being our solution can be used only by configuring Drupal's permissions such that anonymous users are able to subscribe to web hooks, which is respected by the corresponding service.

In the end building upon the Services module makes our solution future proof, such that any upcoming developments like further supported data formats or authentication mechanisms can be easily leveraged. Therefore we obtain a flexible solution for defining and invoking web hooks that is easy to control by the mean of rules. An example showing the configuration of a web hook and its usage can be found in chapter 6.

Remote systems

For communication with remote systems we have described the architecture of remote site definitions acting as remote proxies in section 5.2. To implement that we have created a module, which provides the possibility to create remote site definitions and allows modules to specify remote endpoint type implementations, whereas the module itself already provides an implementation supporting Rules web hooks, such that the module can be used to react on web hook invocations as described in the preceding section.

Just as for web hook configurations we utilize the entity CRUD API for implementing the remote site definitions by providing a new, exportable entity type `rules_web_remote` , such that we obtain a hook that allows modules to define remote site definitions in code and we are prepared for making remote site definitions fieldable. A remote site definition needs to have a unique name, an assigned remote endpoint type and a URL pointing to the remote service, whereas the usage of the URL finally depends on the remote endpoint type in use. To enable modules to provide further remote endpoint types the module introduces the hook `hook_rules_endpoint_types()` . Any module implementing this hook has to specify a class implementing the interface `RulesWebRemoteEndpointInterface` . This interface contains methods that enable the endpoint type to define Rules data types, entities, conditions, actions and events depending on the specified rules remote site definition. For that the endpoint type may take endpoint type specific configuration settings into account, for which the configuration form of remote site definitions can be altered in order to give users the possibility to specify those settings once a user interface is provided.

The created module providing integration with remote systems now takes the information provided by the endpoint type implementations of the available remote site definitions and registers the defined Rules data types, entities, conditions, actions and events in the system - hence making the definitions available for regular use with the Rules module.

Reacting on web hooks

The provided remote endpoint type implementation for reacting on Rules web hooks makes use of the REST client module of Hugo Wetterberg, as mentioned in the preceding section, in order to communicate with the RESTful services of the Rules web hook module. It connects to the remote Drupal instance to retrieve information about the available entities, their metadata and the available web hooks and provides those definitions for the corresponding remote site definition, whereas for each web hook an event is defined. However in order to receive any HTTP notification when a web hook is invoked, the site needs to subscribe to the hook. For that the endpoint type implements the corresponding method for subscribing to remote events. This method gets invoked whenever necessary, such that the system subscribes to the remote event when required.

As a result the web hooks are integrated as remote events in the system, whereas the passed data is fully available and the necessary subscription is managed automatically. This ensures that remote events can be easily utilized, just like any regular Rules event. A usage example making use of this functionality is presented in chapter 6.

Invoking WS* web services

One of the aims of our work is being able to invoke any SOAP service, as stated by Objective 2. To meet this requirement a remote endpoint type for invoking WS* web services has been implemented. The solution relies on the PHP SOAP extension[10] available for PHP 5 and requires the user to provide a WSDL file describing the service. Additionally the endpoint type requires further configuration that defines the available SOAP operations as well as metadata about the data structures used for the input and output parameter of the operations. With this definitions in place the WS* endpoint type implementation makes an action for each described operation available to the system.

For testing purposes and to give an example usage we leverage the system to

[10] PHP SOAP extension: `http://php.net/manual/en/book.soap.php` (accessed 22-05-2010).

integrate with a simple, freely available SOAP service called *geocoder.us*[11], which finds the latitude and longitude of any US address. It follows the remote site definition for the service, which is specified in code by making use of the provided default hook:

```
/**
 * Implements hook_default_rules_web_remote().
 */
function rules_soap_default_rules_web_remote() {
  $remote = new RulesWebRemote();
  $remote->name = 'geocoder';
  $remote->label = 'Geocoder.us';
  $remote->url = 'http://geocoder.us/dist/eg/clients/GeoCoderPHP.
      wsdl';
  $remote->type = 'soap';

  // Add info about the SOAP service.
  $operation['label'] = 'Geocode an address';
  $operation['parameter']['address'] = array('type' => 'text');
  $operation['provides']['address_results'] = array(
    'type' => 'list<struct>',
    'property info' => array(
      'number' => array('type' => 'integer'),
      'zip' => array('type' => 'integer'),
      'suffix' => array('type' => 'text'),
      'prefix' => array('type' => 'text'),
      'type' => array('type' => 'text'),
      'street' => array('type' => 'text'),
      'state' => array('type' => 'text'),
      'city' => array('type' => 'text'),
      'lat' => array('type' => 'decimal'),
      'long' => array('type' => 'decimal'),
    ),
  );
  $remote->settings['operations']['geocode_address'] = $operation;
  $remotes[$remote->name] = $remote;
  return $remotes;
}
```

[11]geocoder.us: http://geocoder.us/ (accessed 22-05-2010)

As seen the remote site definition defines a new operation, taking the address as input parameter and returning a list of results, whereas each result is represented by a data structure including multiple properties such as the latitude and longitude. As a result the operation is available as usual Rules action to the system, thus a simple rule that leverages the action to pass an address to *geocoder.us* and shows the returned longitude using a system message, could be defined and executed as shown in the following code excerpt:

```
$rule = rule(array('address' => array('type' => 'text')));
$rule->action('rules_web_geocoder_geocode_address', array('
    param_address:select' => 'address'))
     ->action('drupal_message', array('message:select' => 'var:0:
         long'))
     ->execute('1600 Pennsylvania Av, Washington, DC');
```

Integrating RESTful services

Similarly as we have shown how to invoke WS* web services, we provide a remote endpoint type implementation revealing the possibility to utilize our work for integrating RESTful web services. The implementation makes use of the REST client module we have already leveraged for reacting on web hook invocations and analogously to the SOAP implementation it requires further configuration that defines the available operations - thus being in particular useful for the common REST-RPC hybrid web services as described in section 2.2. However one could easily extend the implementation to better integrate with REST conform designs by providing the resources as entities to the system - such as it has been implemented for the Rules web hook endpoint type implementation.

To show an example usage we make use of Google's AJAX API for Translation and Detection[12]. For that the following remote site definition making use of the described endpoint type implementation for REST services is created:

[12]Google AJAX API for Translation and Detection http://code.google.com/apis/ajaxlanguage/documentation/#Translation (accessed 23.05.2010).

```
/**
 * Implements hook_default_rules_web_remote().
 */
function rules_rest_default_rules_web_remote() {
  $remote = new RulesWebRemote();
  $remote->name = 'google';
  $remote->label = 'Google Ajax APIs';
  $remote->url = 'http://ajax.googleapis.com/ajax/services/';
  $remote->type = 'REST';

  // Add info about the REST service.
  $operation['label'] = 'Translate text';
  $operation['url'] = 'language/translate';
  $operation['parameter']['q'] = array('type' => 'text', 'label' =>
       'Text');
  $operation['parameter']['langpair'] = array(
    'type' => 'text',
    'label' => 'Language pair',
    'description' => 'The language pair used for translating, such
        as de|en.',
  );
  // We have to specify the version 1.0
  $operation['parameter']['v'] = array('type' => 'hidden', 'default
       value' => '1.0');

  $operation['provides']['responseData'] = array(
    'type' => 'struct',
    'label' => 'Translation result',
    'property info' => array(
      'translatedText' => array(
        'type' => 'text',
        'label' => 'Translated text',
      ),
    ),
  );
  $remote->settings['operations']['translate'] = $operation;
  $remotes[$remote->name] = $remote;
  return $remotes;
}
```

The remote site definition describes the *translate* operation, such that the system

provides an action for it. Thus the following example usage of the action translates the German text *Hallo Welt* to English and shows the result *Hello World* to the currently logged in user using a regular Drupal system message:

```
$rule = rule(array('text' => array('type' => 'text')));
$rule->action('rules_web_google_translate', array(
       'param_q:select' => 'text',
       'param_langpair' => 'de|en',
     ))
     ->action('drupal_message', array('message:select' => '
         responsedata:translatedText'))
     ->execute('Hallo Welt');
```

CHAPTER 6

Use cases

In this chapter we show how our work can be utilized to implement some use cases. First off we present how a simple editorial workflow can be accomplished, then in the next step we improve this solution to work in a distributed way, such that there is a separate editorial back-end supplying the content for one or more front-ends. Furthermore we illustrate how the significant use cases stated in Objective 3 could be implemented.

6.1 Editorial workflow

For implementing a simple editorial workflow with the means of Drupal and the enhanced Rules module, we build upon Drupal's field system as introduced in section 2.3. Thus we create a field of type *List (text)* for tracking the workflow state of a node and configure it to have the following allowed values:

- Draft (needs work)
- Draft (needs review)
- Published

We attach this field to a content type, e.g., to the content type *article*, which Drupal configures by default during installation. This field allows users to specify the state of an article, so an editor can start a new article, which gets the default state *Draft (needs work)* assigned. When the editor decides that the article is ready, he assigns

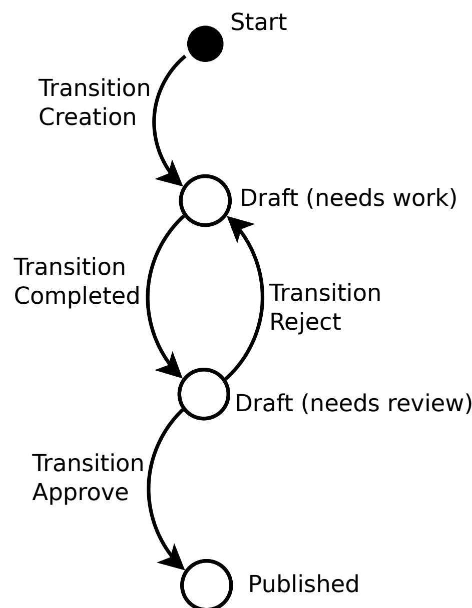

Figure 6.1: The states of the editorial workflow with the state transitions of a usual instance.

the state *Draft (needs review)* to the article. Thus when a reviewer has a look at the draft, he either approves the draft and sets its state to *Published*, or if the article needs work, back to *Draft (needs work)*. An overview over the usual state transitions of this workflow can be seen in figure 6.1.

Now our work can be used to detect state transitions of the created field and to react accordingly. First off we configure rules that promote any published article to Drupal's front page. The rule as shown in the following export reacts on the event *Before saving content* to execute the action *Promote content to front page* if the content has the workflow field attached and the workflow state is *Published*.

```
{ "rules_workflow_promote" : {
    "LABEL" : "Promote published content to the front page",
    "PLUGIN" : "reaction rule",
    "REQUIRES" : [ "rules" ],
    "ON" : [ "node_presave" ],
    "IF" : [
      { "entity_has_field" : { "entity" : [ "node" ], "field" : "
          field_workflow" } },
```

```
      { "data_is" : { "data" : [ "node:field-workflow" ], "value"
           : "published" } }
    ],
    "DO" : [ { "node_promote" : { "node" : [ "node" ] } } ]
  }
}
```

Thus this rule ensures that any content being in the *Published* state appears on the front page. Additionally we have to make sure any content that is not in the *Published* state is removed from the front page. For that we create an analogously working rule, which executes the action *Remove content from front page* in case the content is not in the *Published* state.

Next we use rules to log occurring state transitions. To achieve that we automatically create a new revision of the content and log the state change to the node revision log with the help of rules reacting on the state transitions. As a result any state change is logged by the system and the changes between the state transitions can be easily tracked by looking at the captured node revisions. The following rule detects workflow state transitions and ensures a new revision is created:

```
{ "rules_workflow_transition" : {
    "LABEL" : "Log workflow state changes",
    "PLUGIN" : "reaction rule",
    "REQUIRES" : [ "rules" ],
    "ON" : [ "node_presave" ],
    "IF" : [
      { "entity_has_field" : { "entity" : [ "node" ], "field" : "
          field_workflow" } },
      { "NOT data_is" : {
          "data" : [ "node:field-workflow" ],
          "value" : [ "node_unchanged:field-workflow" ]
        }
      }
    ],
    "DO" : [
      { "data_set" : {
          "data" : [ "node:log" ],
          "value" : "The workflow state has been set from \"[
              node_unchanged:field-workflow]\" to \"[node:field-
```

```
              workflow]\"."
        }
      },
      { "data_set" : { "data" : [ "node:revision" ], "value" :
          true } },
      { "drupal_message" : { "message" : "Your changes to the
          workflow state have been logged and a new revision has
          been created." } }
    ]
  }
}
```

Thus when content is updated, the rule compares the workflow state of the unchanged content node to the updated one. If the states are not equal, the rule fires. As seen in the export the rule ensures a new revision is created by enabling the *Create new revision* (`node:revision`) option of the node, adds an informative revision log message and displays a message informing the acting user that the changes have been logged. Apart from that the configured revision log message utilizes the token input evaluator, which provides dynamic replacement tokens, for including the actual values of the workflow state before and after the transition in the message. It is noted that we have the ability to use these token replacements only as the Entity metadata module automatically provides token replacements based on the provided metadata for all textual fields having no token replacement support yet.

The previous shown rule logs any state transitions when content is updated. In order to also log a message for the initial revision, we configure the following rule:

```
{ "rules_workflow_init" : {
    "LABEL" : "Log initial workflow state",
    "PLUGIN" : "reaction rule",
    "REQUIRES" : [ "rules" ],
    "ON" : [ "node_presave" ],
    "IF" : [
      { "entity_has_field" : { "entity" : [ "node" ], "field" : "
          field_workflow" } },
      { "entity_is_new" : { "entity" : [ "node" ] } }
    ],
    "DO" : [
```

```
      { "data_set" : {
          "data" : [ "node:log" ],
          "value" : "Initial workflow state: \"[node:field-
              workflow]\"."
        }
      }
    ]
  }
}
```

Thus as seen in the export the rule makes use of the *Entity is new* condition to let the rule log the initial workflow state of the newly created content node. As a result finally all state transitions are logged to the node revision log, which therefore provides the complete workflow history for a content node. As example figure 6.2 shows the revision log created by the editorial workflow for a node, which has gone through all usual state transitions as visualized in figure 6.1.

Lastly it should be noted that for all rules we have utilized the *Entity has field* condition instead of using the *Content is of type* condition to check for the content type which is making use of the workflow. This ensures that the rules automatically apply to any content type to which the field is attached. Hence adapting the site configuration to leverage the created editorial workflow also for another content type is as easy as attaching the workflow field to it.

Useful enhancements

In addition to the described basic editorial workflow the following useful enhancements would be feasible:

- In order to match the description of the use case *A workflow based review process* as described in Objective 3 as significant use case to consider, the Views module as introduced in section 2.3 could be utilized to create two listings - one of content that needs work and one of content that needs review. Furthermore the Views module is able to create pages that can be configured to be visible to a

Revisions for *Example article* View Edit **Revisions** Dev load Dev render

Revision	Operations
06/06/2010 - 17:40 by fago The workflow state has been set from "Draft (needs review)" to "Published".	*current revision*
06/06/2010 - 17:39 by Bob The workflow state has been set from "Draft (needs work)" to "Draft (needs review)".	revert delete
06/06/2010 - 17:38 by fago The workflow state has been set from "Draft (needs review)" to "Draft (needs work)".	revert delete
06/06/2010 - 17:37 by Bob The workflow state has been set from "Draft (needs work)" to "Draft (needs review)".	revert delete
06/06/2010 - 17:37 by Bob Initial workflow state: "Draft (needs work)".	revert delete

Figure 6.2: The node revision log listing all previous workflow state transitions of the node.

certain user role only [DrV09], such that each listing could be shown on a page visible for reviewers or editors only.

- To enable reviewers to leave a comment, one could build upon the commenting functionality provided by the comment module of Drupal core. Furthermore with Drupal 7 comments are fieldable, thus one could attach the workflow field to the comments and set up rules that apply the selected workflow state of a posted comment to the node. Doing so would enable reviewers to leave a comment that explains their intentions but also changes the workflow state of the node at the same time.

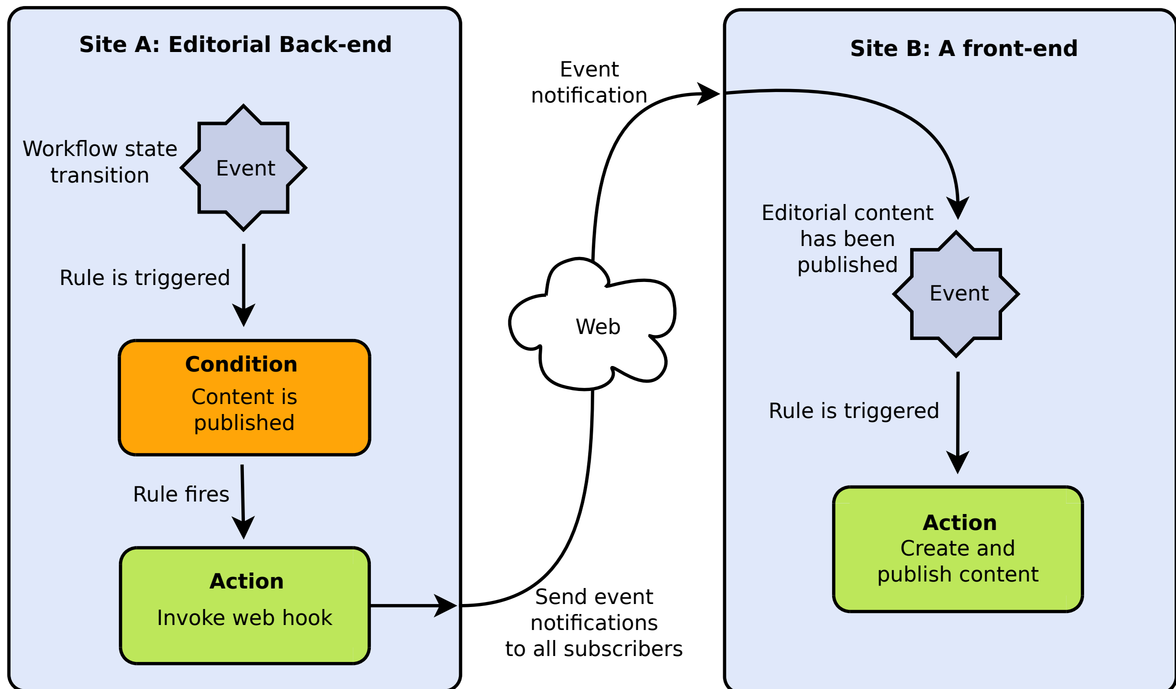

Figure 6.3: The editorial workflow extended with a separate front-end site to which the content is published with the help of Rules web hooks.

6.2 A distributed workflow

To show a distributed workflow, we improve the publishing workflow as introduced in the preceding section to work in a distributed way. That means instead of having editors working directly on the site where content is going to be published, there is a separate editorial back-end site and content is published to another, or even multiple, sites that act as front-ends serving the content to the public. To achieve that we provide Rules web hooks on the editorial back-end, such that the front-ends are able to react on these hooks in order to locally create and publish the content. Figure 6.3 gives an overview over the needed rules to publish the content on a separate front-end site.

On the back-end we build upon the editorial workflow as described in the preceding section. For that we add a definition for a Rules web hook that is invoked through a rule whenever content is published. It follows the definition of the web hook, whereby we have specified the definition in code by making use of the provided

default hook:

```
/**
 * Implements hook_default_rules_web_hook().
 */
function rules_workflow_default_rules_web_hook() {
  $hook = new EntityDB(array(), 'rules_web_hook');
  $hook->name = 'content_published';
  $hook->label = 'Content has been published';
  $hook->active = TRUE;
  $hook->variables = array(
    'node' => array(
      'type' => 'node',
      'label' => 'Content',
    ),
  );
  $hooks[$hook->name] = $hook;
  return $hooks;
}
```

Then to actually invoke the web hook we have created the following rule, which invokes the web hook with the help of the provided action whenever some content is set to the *published* workflow state:

```
{ "rules_workflow_content_published" : {
    "LABEL" : "Workflow state changed to published",
    "PLUGIN" : "reaction rule",
    "REQUIRES" : [ "rules", "rules_web_hook" ],
    "ON" : [ "node_update" ],
    "IF" : [
      { "entity_has_field" : { "entity" : [ "node" ], "field" : "
          field_workflow" } },
      { "data_is" : { "data" : [ "node:field-workflow" ], "value"
           : "published" } },
      { "NOT data_is" : {
          "data" : [ "node:field-workflow" ],
          "value" : [ "node_unchanged:field-workflow" ]
        }
      }
    ],
```

```
    "DO" : [ { "web_hook_invoke_content_published" : { "node" : [
          "node" ] } } ]
  }
}
```

In order to configure a front-end site we create the following remote site definition to react on web hook invocations of the editorial web site, whereby the definition is again specified in code by making use of the provided default hook:

```
/**
 * Implements hook_default_rules_web_remote().
 */
function rules_workflow_default_rules_web_remote() {
  $remote = new RulesWebRemote();
  $remote->name = 'editorial_site';
  $remote->label = 'Editorial back-end site';
  $remote->url = 'http://backend.example.com';
  $remote->type = 'rules_web_hook';

  // Optional HTTP authentication settings for the back-end.
  $remote->settings['curl options'][CURLOPT_USERPWD] = 'username:
      password';
  $remote->settings['curl options'][CURLOPT_HTTPAUTH] =
      CURLAUTH_BASIC;
  $remotes[$remote->name] = $remote;
  return $remotes;
}
```

As seen in the remote site definition the URL of the remote Drupal site has to be specified. Additionally HTTP authentication details may be configured if the editorial site requires that. With the remote site definition in place the specified web hook is available to our front-end site as remote event, upon which can be reacted with the help of rules. As soon as a rule is configured to be triggered upon the remote event, the system automatically subscribes to the editorial site in order to receive the event notifications. It is noted that the data values of the variables associated with the web hook are sent together with the event notification to all subscribers, such that this data can be utilized by the reacting rule without having to fetch it separately

from the editorial back-end.

It follows the export of a rule, which makes use of the remote data of the node sent with the event notification to locally create a new content node each time the remote event occurs:

```
{ "rules_workflow_create_published_content" : {
    "LABEL" : "Create published content",
    "PLUGIN" : "reaction rule",
    "REQUIRES" : [ "rules", "rules_web_remote" ],
    "ON" : [ "rules_web_editorial_site_content_published" ],
    "DO" : [
      { "entity_fetch" : {
          "USING" : { "type" : "user", "id" : "1" },
          "PROVIDE" : { "entity_fetched" : { "entity_fetched" : "
              User 1" } }
        }
      },
      { "entity_create" : {
          "USING" : {
            "type" : "node",
            "param_type" : "page",
            "param_title" : [ "node:title" ],
            "param_author" : [ "entity_fetched" ]
          },
          "PROVIDE" : { "entity_created" : { "entity_created" : "
              Created content" } }
        }
      },
      { "data_set" : { "data" : [ "entity_created:body" ], "value
          " : [ "node:body" ] } },
      { "data_set" : { "data" : [ "entity_created:field-uri" ], "
          value" : [ "node:url" ] } }
    ]
  }
}
```

As seen in the export the rule creates a new node of type *page* for each published content node, whereas it takes over the title and the body, sets the node author to the user with the id 1 and finally it saves the URL of the original node in the for that

created field `field-uri`. Having this field ensures that each created node has a reference on the originating content on the editorial site. Thus with that rule in place any published content is locally created on the front-end site, whereas the Rules web remote site integration automatically logs each occurred remote event and the so triggered rule evaluation to Drupal's system log.

We have now shown the configuration needed on the back-end site in order to provide a web hook as well as the configuration needed to setup a simple front-end for publishing the content as visualized in figure 6.3, whereas multiple front-ends - perhaps serving different parts of the content - could be configured the same way.

Possible improvements

The presented solution has been kept rather simple, thus it creates a new content node on the front-end whenever content is published. Thus it could be improved, such that each content node is published on the front-end only one time. Also associated front-end nodes could be hidden when the originating content on the editorial site is changed to be not published any more with the help of another web hook.

6.3 Other significant use cases

We have already stated some significant use cases as part of Objective 3, whereas we have already described in section 6.1 how the presented editorial workflow could be extended to fulfill the use case *A workflow based review process*. Subsequently we shortly present how the remaining significant use cases as stated as part of Objective 3 could be implemented:

For implementing the *Taxonomy sharing* use case the Rules web hooks could be utilized analogously as for the presented distributed workflow: One site has to be defined to be the master site, for which the terms are replicated to all configured client sites. For that on the master site web hooks for the creation, update and deletion of terms have to be configured, as well as rules for invoking the hooks accordingly for all taxonomy terms that should be replicated. Then on each client site on which the taxonomy should be available a remote site definition needs to be created, such that

the web hooks are available as remote events. Next rules can be utilized, such that on term creation on the master site the term is also created locally and the update and deletion of a term is also applied to the local version of the term, whereby for that to work one needs to ensure a lookup of the local version of a remote term is possible. This can be achieved by saving the URL or the identifier of the originating term in a for that added field of the local term, which then can be looked up by using the *Fetch entity by property* action. Then for the creation of a local term the *Create a new entity* action may be used. An update can be propagated to the local version of the term by overwriting the data values of the local term with the latest data values of the remote term. Finally for propagating the deletion of a term, the *Delete entity* action can be utilized.

The *Distributed content deployment* use case could be theoretically implemented as presented for the distributed workflow, where content published on the editorial site is replicated to one or more front-end sites. However as an alternative one might want to follow the pattern as described for the *Taxonomy sharing* use case, which can be analogously applied to nodes, as this pattern has the advantage that content updates and deletions would be properly propagated.

For the *Automated publishing and expiration of content* the scheduling capabilities of the Rules module could be utilized. In order to do so one needs to create two rule sets, one for publishing content and one for letting the content expire, e.g., with the help of the so called *Unpublish content* action. Then if the provided Rules scheduler extension module is activated, actions for scheduling these rule sets are available.

To enable content editors to specify the publishing and optionally an expiration date, fields could be created - whereas for that we would need date fields as provided by the Date module[1], which unfortunately has no Drupal 7 compatible release as of the time of the writing. Additionally also the existence of appropriate metadata for date fields would be required, such that the created fields are recognized by the system to provide date values. With that in place, one could easily create two date fields for a content type - one for the publishing date and one optional field for

[1]Date module: `http://drupal.org/project/date` (accessed 11-06-2010)

the expiration date. Then these fields can be used by rules that execute the actions for scheduling the created rule sets whenever content is created or edited and a publishing or expiration date is provided. Thus when a configured publishing or expiration date is reached, the Rules scheduler module evaluates the configured rule sets and thus the content gets published or expires as desired.

Chapter 7

Evaluation

For evaluating the results of the thesis we revisit the objectives stated in chapter 4 in order to show that we have succeeded in attaining all objectives.

O1: Bring Rules to the web
In order to cope with the distributed nature of the web we have built our work on top of the ability to deal with any kind of data structures for which suiting metadata can be provided, what allows for a seamless integration of remote data. For the communication across system boundaries we have built an extensible solution integrating remote systems, whereas we have demonstrated its usage for the rule-based invocation of common REST-RPC hybrid web services in section 5.4.

Furthermore we have enabled the rule-based communication between several Drupal based web sites by introducing web hooks, which allow our work to be leveraged for reacting on events occurring on remote Drupal installations. Also as both - the web hook invocations and the reactions - are configured with the help of rules, the solution provides great flexibility to site administrators.

O2: Integrate WS* web services
Just as for invoking RESTful services, we have leveraged the solution for integrating remote systems to demonstrate the rule-based invocation of WS* web services as shown in section 5.4. Therefore a SOAP based web service can be simply invoked by making use of the generated actions.

O3: Consider significant use cases
All use cases as described in Objective 3 have been considered, whereas we have already shown how our work can be utilized to achieve the desired functionality of all described use cases in section 6.3.

O4: Be extensible and reusable
The enhanced Rules module has been implemented in a modular way, such that modules are not only able to provide further events, conditions, actions, default rule configurations or data types, moreover they can even introduce new rule language constructs by providing new Rules plugins to the system. Furthermore our work supports configurations built upon any plugin enabling the reuse of any plugin on its own, e.g., this enables modules to utilize the provided API for making use of conditions only.

O5: Prepare for decent user interfaces
In order to prepare for a decent user interface to be created later on, we have defined an appropriate interface for the implementation of the user interface per plugin. This ensures the user interface of any configuration can be utilized the same way regardless of the actually configured Rules plugin. Apart from that we have ensured that the metadata of all utilized variables is available on configuration time, which enables the UI to leverage it, e.g., for providing a useful data selection widget that only lists the data properties available for a given data type.

In addition to these objectives the existing module as described in section 2.3 has been analyzed in section 5.1 and all identified flaws have been eliminated in the enhanced module. Thus we have improved the specification of argument values by supporting the two argument configuration modes *direct data input* and *data selection*, improved the data selection to work independent of token replacements and ensured that common usage scenarios are supported without the use of any PHP input evaluation using the `eval()` function by introducing the concept of data processors (see section 5.2).

CHAPTER 8

Conclusions and outlook

In this thesis we have described the enhancement of the existing Rules module for Drupal, whereby we have developed a superior module for Drupal's upcoming major version 7. For that the existing Rules module has been analyzed and all identified flaws have been eliminated. The module has been totally revised in order to leverage the object oriented improvements of PHP 5 for the creation of an extensible and reusable module, such that the API is easy to use but supports sophisticated features like looping and generic data lists. Furthermore the solution has received a vast number of significant new features like the ability to work with arbitrary data structures based on specified metadata, the optional data source selection for argument values or a better readable export format. Next, to achieve that several foundational and generally useful APIs for Drupal have been created (see section 5.3), such as the Entity CRUD API - which simplifies the creation of new entity types - and the Entity Metadata module, which provides us with a unified way to deal with entities and their properties. These foundational APIs, as well as the enhanced Rules module itself, have been contributed to the Drupal community and are available as the *Entity API*[1] and *Rules*[2] projects on drupal.org.

Apart from that we have succeeded in attaining all objectives as shown in chapter 7, in particular we have taken the first step to bring Rules to the web. There-

[1]Entity API: `http://drupal.org/project/entity` (accessed 11-06-2010).
[2]Rules module: `http://drupal.org/project/rules` (accessed 11-06-2010).

fore the data of remote systems has been seamlessly integrated and the module has been enabled to work across system boundaries, such that it could be utilized for the rule-based invocation of web services as well as for reacting on remotely occurring events. More specifically we have demonstrated the rule-based invocation of WS* and RESTful web services and introduced Rules web hooks - a novel approach for the interaction of Drupal based web applications (see section 5.2) enabling custom near-instant reactions on remotely occurring events. Based on that we have demonstrated how our work can be utilized to implement significant use cases in chapter 6 in order to proof the practical relevance of our work.

Therefore our work enables the Rules module to cope with the emergence of web applications and web services, but moreover it lays the foundation for leveraging rules for a flexible, user controlled utilization of the emerging "Web of Data" with Drupal.

8.1 Future work

An overview over issues that could be addressed in the context of possible future enhancements is following.

Provide a decent user interface

While we have prepared for decent user interfaces as stated in Objective 5, the creation of such an interface was not part of this work. However the existence of a decent user interface is not only critical for the adoption of the module in the Drupal community, moreover it is substantial for the practical viability of our work. Therefore we have already started working on a native Drupal user interface and as of the time of the writing the first development version is already available from the modules project page[3] on drupal.org. As shown in figure 8.1 the user interface makes use of Drupal's javascript solution for sorting items to provide drag-and-drop functionality for ordering rule elements.

[3]Rules module project: http://drupal.org/project/rules (accessed 11-06-2010).

Home
Editing reaction rule "Log initial workflow state"

EVENT	OPERATIONS
Before saving content	delete

+ Add event

CONDITIONS	OPERATIONS
Entity is new Parameter: *Entity:* [node]	edit delete
Entity has field Parameter: *Entity:* [node], *Field:* field_workflow	edit delete

+ Add condition + Add or + Add and

ACTIONS	OPERATIONS
Modify data Parameter: *Data:* [node:log], *Value:* Initial workflow state: ...	edit delete

+ Add action + Add loop

SETTINGS

Save changes

Figure 8.1: Screenshot of the development version of the user interface for the enhanced Rules module showing the configuration overview of a reaction rule.

Advance the Rules web modules

We have presented solutions for the integration of remote systems and for providing Rules web hooks, however as described in section 5.4 the shown implementation is based upon unofficial development versions of Drupal modules. Therefore the solution needs to be migrated to build upon official modules once they are available for Drupal 7, so that the solution can be provided as a regular Drupal module on drupal.org. To achieve that also a decent user interface for the management of remote site definitions needs to be created. Furthermore the presented solution for rule-based service invocations does not manage authentication, security, failures or versioning of web service interfaces, thus these points remain open and could be addressed in future work.

Improved integration with the Service Oriented and Event Driven Architecture

The integration with WS* web services as demonstrated in section 5.4 allows for a simple integration with the Services Oriented Architecture of enterprises. However first off the usability of this solution could be improved by parsing the WSDL description of the web services, such that the needed configuration for the definition of the available SOAP operations as well as the metadata about the data structures used for the input and output parameter of the operations is automatically determined.

Furthermore events exchanged as messages between web systems are a natural communication paradigm, as already exploited in the Service Oriented and Event Driven Architecture [BE06]. Ideally the integration of remote systems (see section 5.2) allows for the reaction on remote events, thus our solution could be easily extended to leverage existing event messages. To gain the ability to react on received event messages, an appropriate remote endpoint implementation could be implemented in order to tighten the integration with the Service Oriented and Event Driven Architecture.

Reactivity on the web

The rule-based invocation of web services enables Drupal site builders to utilize the variety of available services on the web. However with the evolvement of the web from a distributed repository of documents to web applications, web services and the Web 2.0, the web has become more dynamic. Therefore the ability to react on dynamic changes, such as the update of a data source, is gaining importance [BBB+07]. Bry, F. and Eckert, M. [BE06] even postulate that high-level reactive languages are needed on the web and Event-Condition-Action rules are well-suited to specify reactivity on the web.

While Rules web hooks provide a novel solution for reacting on events occurring on remote Drupal sites, our work does not provide means to react on changes occurring on any remote sites that are not based upon Drupal - as for enabling instant reactions those remote sites would need to send event messages to our Drupal installation upon which we can react. A way to achieve that would be adding support for the *PubSubHubbub* protocol - a protocol that extends Atom or RSS feeds for providing near-instant notifications of changes via a Publish-Subscribe

mechanism [GTW10, pub10]. Thus with the help of the introduced Rules remote site integration it would be feasible to extend the Rules module, such that events for reacting on *PubSubHubbub* provided near-instant change notifications are available.

Semantic Web

According to May, W. and Alferes, J. and Amador, R. [MAA05] Event-Condition-Action rules provide a generic uniform framework for specifying and implementing communication, local evolution, policies and strategies, and altogether global evolution in the Semantic Web. Also there is already foundational support for Semantic Web technologies such as RDFa (see 2.3) in Drupal core and it is likely that further enhancements for Drupal 7 will be available as extension modules, such as it is already the case for Drupal 6 [CCPD08, CDC+09]. Thus it appears to be reasonable to extend the Rules module accordingly by building upon the Semantic Web related modules for Drupal, e.g., such that the system is able to utilize RDF data and RDFa attributes embedded in XHTML documents can be read.

Mashups

Web mashups are web applications that are composed of existing content and services and have recently received rapidly increasing attention [YBCD08]. With the gained ability to rule-based invoke RESTful and WS* web services and the possibility to leverage this functionality for creating re-usable components, the Rules module could even serve as Drupal based mashup tool. Thus this application of the module could be further explored and any for that use case missing parts could be added as extensions to the module, e.g., it would be feasible to create a user interface that is designed to ease the creation of mashups based upon the module.

Appendix A

Acronyms

AMF Action Message Format

API Application Programming Interface

BRMS Business Rules Management System

CEP Complex Event Processing

CRUD Create, Read, Update and Delete

ECA Event-Condition-Action rules

JSON JavaScript Object Notation

OWL The Web Ontology Language

RDF Resource Description Framework

REST Representational State Transfer

RIF Rule Interchange Format

RPC Remote Procedure Call

RSS Really Simple Syndication

RuleML Rule Markup Language

SOAP Simple Object Access Protocol

SWRL Semantic Web Rule Language

UDDI Universal Description, Discovery and Integration

UI User Interface

URI Uniform Resource Identifier

URL Uniform Resource Locator

WSDL Web Services Description Language

XHTML Extensible HyperText Markup Language

XML Extensible Markup Language

APPENDIX **B**

API documentation

B.1 Rules module

```
/**
 * @defgroup rules_hooks Rules' hooks
 * @{
 * Hooks that can be implemented by other modules in order to extend rules.
 */

/**
 * Define rules compatible actions.
 *
 * This hook is required in order to add a new rules action. It should be
 * placed into the file MODULENAME.rules.inc, which gets automatically included
 * when the hook is invoked.
 *
 * @return
 *   An array of information about the module's provided rules actions.
 *   The array contains a sub-array for each action, with the action name as
 *   the key.
 *   Possible attributes for each sub-array are:
 *   - label: The label of the action. Start capitalized. Required.
 *   - group: A group for this element, used for grouping the actions in the
 *     interface. Should start with a capital letter and be translated.
 *     Required.
 *   - parameter: An array describing all parameter of the action with
 *     the parameter's name as key.  Optional. Each parameter has to be
 *     described by a sub-array with possible attributes as described
 *     afterwards, whereas the name of a parameter needs to be a lowercase,
 *     valid PHP variable name.
 *   - provides: An array describing the variables the action provides to the
 *     evaluation state with the variable name as key. Optional. Each variable
 *     has to be described by a sub-array with possible attributes as described
 *     afterwards, whereas the name of a parameter needs to be a lowercase,
```

```
 *     valid PHP variable name.
 *   - 'named parameter': If set to TRUE, the arguments will be passed as a
 *     single array with the parameter names as keys. This emulates named
 *     parameters in PHP and is in particular useful if the number of parameters
 *     can vary. Optionally, defaults to FALSE.
 *   - base: The base for action implementation callbacks to use instead of the
 *     action's name. Optional (defaults to the name).
 *   - callbacks: An array which allows to set specific function callbacks for
 *     the action. The default for each callback is the actions base appended
 *     by '_' and the callback name.
 *   - 'access callback': An optional callback, which has to return whether the
 *     currently logged in user is allowed to configure this action. See
 *     rules_node_integration_access() for an example callback.
 *  Each 'parameter' array may contain the following properties:
 *   - label: The label of the parameter. Start capitalized. Required.
 *   - type: The rules data type of the parameter, which is to be passed to the
 *     action. All types declared in hook_rules_data_info() may be specified, as
 *     well as an array of possible types. Also lists and lists of a given type
 *     can be specified by using the notating list<integer> as introduced by
 *     the entity metadata module. The special keyword '*' can be used when all
 *     types should be allowed. Required.
 *   - bundles: Optionally, an array of bundle names. When the specified type is
 *     set to a single entity type, this may be used to restrict the allowed
 *     bundles.
 *   - description: If necessary, a further description of the parameter.
 *     Optional.
 *   - options list: Optionally, a callback that returns an array of possible
 *     values for this parameter. The callback has to return an array as used
 *     by hook_options_list(). For an example implementation see
 *     rules_data_action_type_options().
 *   - save: If this is set to TRUE, the parameter will be saved by rules when
 *     the rules evaluation ends. This is only supported for savable data
 *     types. If the action returns FALSE, saving is skipped.
 *   - optional: May be set to TRUE, when the parameter isn't required.
 *   - 'default value': The value to pass to the action, in case the parameter
 *     is optional and there is no specified value. Optional.
 *   - restriction: Restrict how the argument for this parameter may be
 *     provided. Supported values are 'selector' and 'input'. Optional.
 *   - sanitize: Optionally. Allows parameters of type 'text' to demand an
 *     already sanitized argument. If enabled, any user specified value won't be
 *     sanitized itself, but replacements applied by input evaluators are.
 *  Each 'provides' array may contain the following properties:
 *   - label: The label of the variable. Start capitalized. Required.
 *   - type: The rules data type of the variable. All types declared in
 *     hook_rules_data_info() may be specified. Types may be parametrized e.g.
 *     the types node<page> or list<integer> are valid.
 *   - save: If this is set to TRUE, the provided variable is saved by rules
 *     when the rules evaluation ends. Only possible for savable data types.
 *     Optional (defaults to FALSE).
 *   - 'label callback': A callback to improve the variables label using the
 *     action's configuration settings. Optional.
 *
 *  The module has to provide an implementation for each action, being a
 *  function named as specified in the 'base' key or for the execution callback.
 *  All other possible callbacks are optional.
 *  Supported action callbacks by rules are defined and documented in the
 *  RulesPluginImplInterface. However any module may extend the action plugin
 *  based upon a defined interface using hook_rules_plugin_info(). All methods
 *  defined in those interfaces can be overridden by the action implementation.
```

```
 *  The callback implementations for those interfaces may reside in any file
 *  specified in hook_rules_file_info().
 *
 *  @see hook_rules_file_info()
 *  @see rules_action_execution_callback()
 *  @see hook_rules_plugin_info()
 *  @see RulesPluginImplInterface
 */
function hook_rules_action_info() {
  return array(
    'mail_user' => array(
      'label' => t('Send a mail to a user'),
      'parameter' => array(
        'user' => array('type' => 'user', 'label' => t('Recipient')),
      ),
      'group' => t('System'),
      'base' => 'rules_action_mail_user',
      'callbacks' => array(
        'validate' => 'rules_action_custom_validation',
        'help' => 'rules_mail_help',
      ),
    ),
  );
}

/**
 * Specify files containing rules integration code.
 *
 * All files specified in that hook will be included when rules looks for
 * existing callbacks for any plugin. Rules remembers which callback is found in
 * which file and automatically includes the right file before it is executing
 * a plugin method callback. The file yourmodule.rules.inc is added by default
 * and need not be specified here.
 * This allows you to add new include files only containing functions serving as
 * plugin method callbacks in any file without having to care about file
 * inclusion.
 *
 * @return
 *   An array of file names without the file ending which defaults to '.inc'.
 */
function hook_rules_file_info() {
  return array('yourmodule.rules-eval');
}

/**
 * The execution callback for an action.
 *
 * It should be placed in any file included by your module or in a file
 * specified using hook_rules_file_info().
 *
 * @param
 *   The callback gets arguments passed as described as parameter in
 *   hook_rules_action_info() as well as an array containing the action's
 *   configuration settings.
 * @return
 *   The action may return an array containg parameter or provided variables
 *   with their names as key. This is used update the value of a parameter or to
 *   provdide the value for a provided variable.
 *   Apart from that any parameters which have the key 'save' set to TRUE will
```

```
 *   be remembered to be saved by rules unless the action returns FALSE.
 *   Conditions have to return a boolean value in any case.
 *
 * @see hook_rules_action_info()
 * @see hook_rules_file_info()
 */
function rules_action_execution_callback($node, $title, $settings) {
  $node->title = $title;
  return array('node' => $node);
}

/**
 * Define rules conditions.
 *
 * This hook is required in order to add a new rules condition. It should be
 * placed into the file MODULENAME.rules.inc, which gets automatically included
 * when the hook is invoked.
 *
 * Adding conditions works exactly the same way as adding actions, with the
 * exception that conditions can't provide variables and cannot save parameters.
 * Thus the 'provides' attribute is not supported. Furthermore the condition
 * implementation callback has to return a boolean value.
 *
 * @see hook_rules_action_info()
 */
function hook_rules_condition_info() {
  return array(
    'rules_condition_text_compare' => array(
      'label' => t('Textual comparison'),
      'parameter' => array(
        'text1' => array('label' => t('Text 1'), 'type' => 'text'),
        'text2' => array('label' => t('Text 2'), 'type' => 'text'),
      ),
      'group' => t('Rules'),
    ),
  );
}

/**
 * Define rules events.
 *
 * This hook is required in order to add a new rules event. It should be
 * placed into the file MODULENAME.rules.inc, which gets automatically included
 * when the hook is invoked.
 * The module has to invoke the event when it occurs using rules_invoke_event().
 * This function call has to happen outside of MODULENAME.rules.inc,
 * usually it's invoked directly from the providing module but wrapped by a
 * module_exists('rules') check.
 *
 * @return
 *   An array of information about the module's provided rules events. The array
 *   contains a sub-array for each event, with the event name as the key.
 *   Possible attributes for each sub-array are:
 *   - label: The label of the event. Start capitalized. Required.
 *   - group: A group for this element, used for grouping the events in the
 *     interface. Should start with a capital letter and be translated.
 *     Required.
 *   - 'access callback': An callback, which has to return whether the
 *     currently logged in user is allowed to configure rules for this event.
```

```
 *      Access should be only granted, if the user at least may access all the
 *      variables provided by the event. Optional.
 *    - variables: An array describing all variables that are available for
 *      elements reaction on this event. Optional. Each variable has to be
 *      described by a sub-array with the possible attributes:
 *      - label: The label of the variable. Start capitalized. Required.
 *      - type: The rules data type of the variable. All types declared in
 *        hook_rules_data_info() may be specified. Types may be parametrized e.g.
 *        the types node<page> or list<integer> are valid.
 *      - 'skip save': If the variable is saved after the event has occured
 *        anyway, set this to TRUE. So rules won't save the variable a second
 *        time. Optional, defaults to FALSE.
 *      - handler: A handler to load the actual variable value. This is useful
 *        for lazy loading variables. The handler gets all so far available
 *        variables passed in the order as defined. Optional. Also see
 *        http://drupal.org/node/298554.
 *
 *  @see rules_invoke_event()
 */
function hook_rules_event_info() {
  $items = array(
    'node_insert' => array(
      'label' => t('After saving new content'),
      'group' => t('Node'),
      'variables' => rules_events_node_variables(t('created content')),
    ),
    'node_update' => array(
      'label' => t('After updating existing content'),
      'group' => t('Node'),
      'variables' => rules_events_node_variables(t('updated content'), TRUE),
    ),
    'node_presave' => array(
      'label' => t('Content is going to be saved'),
      'group' => t('Node'),
      'variables' => rules_events_node_variables(t('saved content'), TRUE),
    ),
    'node_view' => array(
      'label' => t('Content is going to be viewed'),
      'group' => t('Node'),
      'variables' => rules_events_node_variables(t('viewed content')) + array(
        'build_mode' => array('type' => 'string', 'label' => t('view mode')),
      ),
    ),
    'node_delete' => array(
      'label' => t('After deleting content'),
      'group' => t('Node'),
      'variables' => rules_events_node_variables(t('deleted content')),
    ),
  );
  // Specify that on presave the node is saved anyway.
  $items['node_presave']['variables']['node']['skip save'] = TRUE;
  return $items;
}

/**
 * Define rules data types.
 *
 * This hook is required in order to add a new rules data type. It should be
 * placed into the file MODULENAME.rules.inc, which gets automatically included
```

```
 * when the hook is invoked.
 * Rules builds upon the entity metadata module, thus to improve the support of
 * your data in rules, make it an entity if possible and provide metadata about
 * its properties and CRUD functions by integrating with the entity metadata
 * module.
 * For a list of data types defined by rules see rules_data_data_info().
 *
 *
 * @return
 *   An array of information about the module's provided data types. The array
 *   contains a sub-array for each data type, with the data type name as the key.
 *   Possible attributes for each sub-array are:
 *   - label: The label of the data type. Start uncapitalized. Required.
 *   - wrap: If set to TRUE, the data is wrapped internally using wrappers
 *     provided by the entity metadata module. This is required for entities and
 *     data structures to support the application of data selectors or
 *     intelligent saving.
 *   - property info: May be used for data structures being no entities to
 *     provide info about the data properties, such that data selectors via an
 *     entity metadata wrapper are supported. Specify an array as expected by
 *     entity_metadata_wrapper(). Optionally.
 *   - parent: Optionally a parent type may be set to specify a sub-type
 *     relationship, which will be only used for checking compatible types. E.g.
 *     the 'entity' data type is parent of the 'node' data type, thus a node may
 *     be also used for any action needing an 'entity' parameter. Can be set to
 *     any known rules data type.
 *   - group: A group for this element, used for grouping the data types in the
 *     interface. Should start with a capital letter and be translated.
 *     Required.
 *   - 'token type': The type name as used by the token module. Defaults to the
 *     type name as used by rules. Use FALSE to let token ignore this type.
 *     Optional.
 *   - hidden: Whether the data type should be hidden from the UI. Optional
 *    (defaults to FALSE).
 *
 *  @see entity_metadata_wrapper()
 *  @see hook_rules_data_info_alter()
 *  @see rules_data_data_info()
 */
function hook_rules_data_info() {
  return array(
    'node' => array(
      'label' => t('content'),
      'parent' => 'entity',
      'group' => t('Node'),
    ),
  );
}

/**
 * Defines rules plugins.
 *
 * A rules configuration may consist of elements being instances of any rules
 * plugin. This hook can be used to define new or to extend rules plugins.
 *
 * @return
 *   An array of information about the module's provided rules plugins. The
 *   array contains a sub-array for each plugin, with the plugin name as the
 *   key. Possible attributes for each sub-array are:
```

```
 *   - label: A label for the plugin. Start capitalized. Required.
 *   - class: The implementation class. Has to extend the RulesPlugin class.
 *   - embeddable: A container class in which elements of those plugin may be
 *     embedded or FALSE to disallow embedding. Common classes that are used
 *     here are RulesConditionContainer and RulesActionContainer.
 *   - component: If set to TRUE, the rules admin UI will list elements of those
 *     plugin in the components UI and allows the creation of new components
 *     based upon this plugin. Optional.
 *   - extenders: This allows one to specify faces extenders, which may be used
 *     to dynamically implement interfaces. Optional. All extenders specified
 *     here are setup automatically by rules once the object is created. To
 *     specify set this to an array, where the keys are the implemented
 *     interfaces pointing to another array with the keys:
 *     - class: The class of the extender, implementing the FacesExtender
 *       and the specified interface. Either 'class' or 'methods' has to exist.
 *     - methods: An array of callbacks that implement the methods of the
 *       interface where the method names are the keys and the callback names
 *       the values. There has to be a callback for each defined method.
 *     - file: An optional array describing the file to include when a method
 *       of the interface is invoked. The array entries known are 'type',
 *       'module', and 'name' matching the parameters of module_load_include().
 *       Only 'module' is required as 'type' defaults to 'inc' and 'name' to
 *       NULL.
 *   - overrides: An optional array, which may be used to specify callbacks to
 *     override specific methods. For that the following keys are supported:
 *     - methods: As in the extenders array, but you may specify as many methods
 *       here as you like.
 *     - file: Optionally an array specifying a file to include for a method.
 *       For each method appearing in methods a file may be specified by using
 *       the method name as key and another array as value, which describes the
 *       file to include - looking like the file array supported by 'extenders'.
 *
 *  @see class RulesPlugin
 *  @see hook_rules_plugin_info_alter()
 */
function hook_rules_plugin_info() {
  return array(
    'or' => array(
      'class' => 'RulesOr',
      'embeddable' => 'RulesConditionContainer',
      'component' => TRUE,
    ),
    'and' => array(
      'class' => 'RulesAnd',
      'embeddable' => 'RulesConditionContainer',
      'component' => TRUE,
    ),
    'rule' => array(
      'class' => 'Rule',
      'embeddable' => 'RulesRuleSet',
      'extenders' => array (
      // Interfaces => array( class => className / methods => array of Methods).
      ),
      'overrides' => array(
      // Array of overrides each being an array ('methods' => .., 'file' => ..).
      ),
    ),
  );
}
```

```
/**
 * Declare provided rules input evaluators.
 *
 * The hook implementation should be placed into the file MODULENAME.rules.inc,
 * which gets automatically included when the hook is invoked.
 * For implementing an input evaluator a class has to be provided which
 * extends the abstract RulesDataInputEvaluator class. Therefore the abstract
 * methods prepare() and evaluate() have to be implemented, as well as access()
 * and help() could be overridden in order to control access permissions or to
 * provide some usage help.
 *
 * @return
 *   An array of information about the module's provided input evaluators. The
 *   array contains a sub-array for each evaluator, with the evaluator name as
 *   the key. Possible attributes for each sub-array are:
 *   - class: The implementation class, which has to extend the
 *     RulesDataInputEvaluator class. Required.
 *   - weight: A weight for controlling the evaluation order of multiple
 *     evaluators. Required.
 *   - type: Optionally, the data types for which the input evaluator should be
 *     used. Defaults to 'text'. Multiple data types may be specified using an
 *     array.
 *
 *  @see class RulesDataInputEvaluator
 *  @see hook_rules_evaluator_info_alter()
 */
function hook_rules_evaluator_info() {
  return array(
    'token' => array(
      'class' => 'RulesTokenEvaluator',
      'type' => array('text', 'uri'),
      'weight' => 0,
     ),
  );
}

/**
 * Declare provided rules data processors.
 *
 * The hook implementation should be placed into the file MODULENAME.rules.inc,
 * which gets automatically included when the hook is invoked.
 * For implementing a data processors a class has to be provided which
 * extends the abstract RulesDataProcessor class. Therefore the abstract
 * method process() has to be implemented, but also the methods form() and
 * access() could be overridden in order to provide a configuration form or
 * to control access permissions.
 *
 * @return
 *   An array of information about the module's provided data processors. The
 *   array contains a sub-array for each processor, with the processor name as
 *   the key. Possible attributes for each sub-array are:
 *   - class: The implementation class, which has to extend the
 *     RulesDataProcessor class. Required.
 *   - weight: A weight for controlling the processing order of multiple data
 *     processors. Required.
 *   - type: Optionally, the data types for which the data processor should be
 *     used. Defaults to 'text'. Multiple data types may be specified using an
 *     array.
```

```
 *
 *  @see class RulesDataProcessor
 *  @see hook_rules_data_processor_info_alter()
 */
function hook_rules_data_processor_info() {
  return array(
    'date_offset' => array(
      'class' => 'RulesDateOffsetProcessor',
      'type' => 'date',
      'weight' => -2,
     ),
  );
}

/**
 * Alter rules compatible actions.
 *
 * The implementation should be placed into the file MODULENAME.rules.inc, which
 * gets automatically included when the hook is invoked.
 *
 * @param $actions
 *   The items of all modules as returned from hook_rules_action_info().
 *
 * @see hook_rules_action_info().
 */
function hook_rules_action_info_alter(&$actions) {
  // The rules action is more powerful, so hide the core action
  unset($actions['rules_core_node_assign_owner_action']);
  // We prefer handling saving by rules - not by the user.
  unset($actions['rules_core_node_save_action']);
}

/**
 * Alter rules conditions.
 *
 * The implementation should be placed into the file MODULENAME.rules.inc, which
 * gets automatically included when the hook is invoked.
 *
 * @param $conditions
 *   The items of all modules as returned from hook_rules_condition_info().
 *
 * @see hook_rules_condition_info()
 */
function hook_rules_condition_info_alter(&$conditions) {
  // Change conditions.
}

/**
 * Alter rules events.
 *
 * The implementation should be placed into the file MODULENAME.rules.inc, which
 * gets automatically included when the hook is invoked.
 *
 * @param $events
 *   The items of all modules as returned from hook_rules_event_info().
 *
 * @see hook_rules_event_info().
 */
function hook_rules_event_info_alter(&$events) {
```

```
  // Change events.
}

/**
 * Alter rules data types.
 *
 * The implementation should be placed into the file MODULENAME.rules.inc, which
 * gets automatically included when the hook is invoked.
 *
 * @param $data_info
 *   The items of all modules as returned from hook_rules_data_info().
 *
 * @see hook_rules_data_info()
 */
function hook_rules_data_info_alter(&$data_info) {
  // Change data types.
}

/**
 * Alter rules plugin info.
 *
 * The implementation should be placed into the file MODULENAME.rules.inc, which
 * gets automatically included when the hook is invoked.
 *
 * @param $plugin_info
 *   The items of all modules as returned from hook_rules_plugin_info().
 *
 * @see hook_rules_plugin_info()
 */
function hook_rules_plugin_info_alter(&$plugin_info) {
  // Change plugin info.
}

/**
 * Alter rules input evaluator info.
 *
 * The implementation should be placed into the file MODULENAME.rules.inc, which
 * gets automatically included when the hook is invoked.
 *
 * @param $evaluator_info
 *   The items of all modules as returned from hook_rules_evaluator_info().
 *
 * @see hook_rules_evaluator_info()
 */
function hook_rules_evaluator_info_alter(&$evaluator_info) {
  // Change evaluator info.
}

/**
 * Alter rules data_processor info.
 *
 * The implementation should be placed into the file MODULENAME.rules.inc, which
 * gets automatically included when the hook is invoked.
 *
 * @param $processor_info
 *   The items of all modules as returned from hook_rules_data_processor_info().
 *
 * @see hook_rules_data_processor_info()
 */
```

```
function hook_rules_data_processor_info_alter(&$processor_info) {
  // Change processor info.
}

/**
 * Act on rules configuration being loaded from the database.
 *
 * This hook is invoked during rules configuration loading, which is handled
 * by entity_load(), via classes RulesEntityController and EntityCRUDController.
 *
 * @param $configs
 *   An array of rules configurations being loaded, keyed by id.
 */
function hook_rules_config_load($configs) {
  $result = db_query('SELECT id, foo FROM {mytable} WHERE id IN(:ids)', array(':
       ids' => array_keys($configs)));
  foreach ($result as $record) {
    $configs[$record->id]->foo = $record->foo;
  }
}

/**
 * Respond to creation of a new rules configuration.
 *
 * This hook is invoked after the rules configuration is inserted into the
 * the database.
 *
 * @param RulesPlugin $config
 *   The rules configuration that is being created.
 */
function hook_rules_config_insert($config) {
  db_insert('mytable')
    ->fields(array(
      'nid' => $config->id,
      'plugin' => $config->plugin,
    ))
    ->execute();
}

/**
 * Act on a rules configuration being inserted or updated.
 *
 * This hook is invoked before the rules configuration is saved to the
 * database.
 *
 * @param RulesPlugin $config
 *   The rules configuration that is being inserted or updated.
 */
function hook_rules_config_presave($config) {
  if ($config->id && $config->module == 'yours') {
    // Add custom condition.
    $config->conditon(/* Your condition */);
  }
}

/**
 * Respond to updates to a rules configuration.
 *
 * This hook is invoked after the configuration has been updated in the
```

```
 * database.
 *
 * @param RulesPlugin $config
 *   The rules configuration that is being updated.
 */
function hook_rules_config_update($config) {
  db_update('mytable')
    ->fields(array('plugin' => $config->plugin))
    ->condition('id', $config->id)
    ->execute();
}

/**
 * Respond to rules configuration deletion.
 *
 * This hook is invoked after the configuration has been removed from the
 * database.
 *
 * @param RulesPlugin $config
 *   The rules configuration that is being deleted.
 */
function hook_rules_config_delete($config) {
  db_delete('mytable')
    ->condition('id', $config->id)
    ->execute();
}

/**
 * Respond to rules configuration execution.
 *
 * This hook is invoked right before the rules configuration is executed.
 *
 * @param RulesPlugin $config
 *   The rules configuration that is being executed.
 */
function hook_rules_config_execute($config) {

}

/**
 * Define default rules configurations.
 *
 * This hook is invoked when rules configurations are loaded. The implementation
 * should be placed into the file MODULENAME.rules_defaults.inc, which gets
 * automatically included when the hook is invoked.
 *
 * @return
 *   An array of rules configurations with the configuration names as keys.
 *
 * @see hook_default_rules_configuration_alter()
 */
function hook_default_rules_configuration() {
  $rule = rules_reaction_rule();
  $rule->label = 'example default rule';
  $rule->active = FALSE;
  $rule->event('node_update')
       ->condition(rules_condition('data_is', array('data:select' => 'node:status
            ', 'value' => TRUE))->negate())
       ->condition('data_is', array('data:select' => 'node:type', 'value' => '
```

```
            page'))
       ->action('drupal_message', array('message' => 'A node has been updated.'))
            ;

  $configs['rules_test_default_1'] = $rule;
  return $config;
}

/**
 * Alter default rules configurations.
 *
 * The implementation should be placed into the file
 * MODULENAME.rules_defaults.inc, which gets automatically included when the
 * hook is invoked.
 *
 * @param $configs
 *   The default configurations of all modules as returned from
 *   hook_default_rules_configuration().
 *
 * @see hook_default_rules_configuration()
 */
function hook_default_rules_configuration_alter(&$configs) {
  // Add custom condition.
  $configs['foo']->condition('bar');
}

/**
 * Alter rules components before execution.
 *
 * This hooks allows altering rules components before they are cached for later
 * re-use. Use this hook only for altering the component in order to prepare
 * re-use through rules_invoke_component() or the provided condition/action.
 * Note that this hook is only invoked for any components cached for execution,
 * but not for components that are programmatically created and executed on the
 * fly (without saving them).
 *
 * @param $plugin
 *   The name of the component plugin.
 * @param $component
 *   The component that is to be cached.
 *
 * @see rules_invoke_component()
 */
function hook_rules_component_alter($plugin, RulesPlugin $component) {

}

/**
 * Alters event sets.
 *
 * This hooks allows altering rules event sets, which contain all rules that are
 * triggered upon a specific event. Rules internally caches all rules associated
 * to an event in an event set, which is cached for fast evaluation. This hook
 * is invoked just before any event set is cached, thus it allows altering of
 * the to be executed rules without the changes to appear in the UI, e.g. to add
 * a further condition to some rules.
 *
 * @param $event_name
 *   The name of the event.
```

```
 * @param $event_set
 *   The event set that is to be cached.
 *
 * @see rules_invoke_event()
 */
function hook_rules_event_set_alter($event_name, RulesEventSet $event_set) {

}

/**
 * @}
 */

/**
 * Provides the interface used for implementing an abstract plugin by using
 * the Faces extension mechanism.
 */
interface RulesPluginImplInterface {

  /**
   * Execute the action or condition making use of the parameters as specified.
   */
  public function execute();

  /**
   * Validates $settings independent from a form submission.
   *
   * @throws RulesException
   *   In case of validation errors, RulesExceptions are thrown.
   */
  public function validate();

  /**
   * Processes $settings independent from a form submission. Only successfully
   * validated settings are processed, so it can be also used to prepare
   * execution dependent on the settings.
   */
  public function process();

  /**
   * Checks whether the user has access to configure this element or rule
   * configuration.
   */
  public function access();

  /**
   * Returns an array of required modules.
   */
  public function dependencies();

  /**
   * Alter the generated configuration form of the element.
   *
   * Validation and processing of the settings should be untied from the form
   * and implemented in validate() and process() wherever it makes sense.
   * For the remaining cases where form tied validation and processing is needed
   * make use of the form API #element_validate and #value_callback properties.
   */
  public function form_alter(&$form, $form_state, $options);
```

```
  /**
   * Optionally returns an array of info assertions for the specified
   * parameters. This allows conditions to assert additional metadata, such as
   * info about the fields of a bundle.
   *
   * @see RulesPlugin::variableInfoAssertions()
   */
  public function assertions();

}

/**
 * Plugin UI Interface.
 */
interface RulesPluginUIInterface {

  /**
   * Adds the whole configuration form of this rules configuration. For rule
   * elements that are part of a configuration this method just adds the
   * elements configuration form.
   *
   * @param $form
   *   The form array where to add the form.
   * @param $form_state
   *   The current form state.
   * @param $options
   *   An optional array of options with the known keys:
   *    - 'show settings': Whether to include the 'settings' fieldset for
   *      editing configuration settings like the label or categories. Defaults
   *      to FALSE.
   *    - 'button': Whether a submit button should be added. Defaults to FALSE.
   *    - 'init': Whether the element is about to be configured the first time
   *      and the configuration is about to be initialized. Defaults to FALSE.
   *    - 'restrict plugins: May be used to restrict the list of rules plugins
   *      that may be added to this configuration. For that set an array of
   *      valid plugins. Note that conditions and actions are always valid, so
   *      just set an empty array for just allowing those.
   *    - 'restrict conditions': Optionally set an array of condition names to
   *      restrict the conditions that are available for adding.
   *    - 'restrict actions': Optionally set an array of action names to
   *      restrict the actions that are available to for adding.
   *    - 'restrict events': Optionally set an array of event names to restrict
   *      the events that are available for adding.
   */
  public function form(&$form, &$form_state, $options = array());

  /**
   * Validate the configuration form of this rule element.
   *
   * @param $form
   * @param $form_state
   */
  public function form_validate($form, &$form_state);

  /**
   * Submit the configuration form of this rule element. This makes sure to
   * put the updated configuration in the form state. For saving changes
```

```
   * permanently, just call $config->save() afterwards.
   *
   * @param $form
   * @param $form_state
   */
  public function form_submit($form, &$form_state);

  /**
   * Returns a structured array for rendering this element in overviews.
   */
  public function buildContent();

  /**
   * Returns the help text for editing this plugin.
   */
  public function help();

  /**
   * Returns ui operations for this element.
   */
  public function operations();

}
```

B.2 Rules web extension modules

Rules web hooks

```
/**
 * @addtogroup rules_hooks
 * @{
 */

/**
 * Act on rules web hooks being loaded from the database.
 *
 * This hook is invoked during rules web hooks loading, which is handled by
 * entity_load(), via the EntityCRUDController.
 *
 * @param $hooks
 *   An array of rules web hooks being loaded, keyed by id.
 */
function hook_rules_web_hook_load($hooks) {
  $result = db_query('SELECT id, foo FROM {mytable} WHERE id IN(:ids)', array(':
       ids' => array_keys($hooks)));
  foreach ($result as $record) {
    $hooks[$record->id]->foo = $record->foo;
  }
}
```

```
/**
 * Respond to creation of a new rules web hook.
 *
 * This hook is invoked after the rules web hook is inserted into the database.
 *
 * @param EntityDB $hook
 *   The rules web hook that is being created.
 */
function hook_rules_web_hook_insert($hook) {
  db_insert('mytable')
    ->fields(array(
      'id' => $hook->id,
      'my_field' => $hook->myField,
    ))
    ->execute();
}

/**
 * Act on a rules web hooks being inserted or updated.
 *
 * This hook is invoked before the rules web hook is saved to the database.
 *
 * @param EntityDB $hook
 *   The rules web hook that is being inserted or updated.
 */
function hook_rules_web_hook_presave($hook) {
  $hook->myField = 'foo';
}

/**
 * Respond to updates to a rules web hook.
 *
 * This hook is invoked after the web hook has been updated in the database.
 *
 * @param EntityDB $hook
 *   The rules web hook that is being updated.
 */
function hook_rules_web_hook_update($hook) {
  db_update('mytable')
    ->fields(array('my_field' => $hook->myField))
    ->condition('id', $hook->id)
    ->execute();
}

/**
 * Respond to a rules web hook deletion.
 *
 * This hook is invoked after the web hook has been removed from the database.
 *
 * @param EntityDB $hook
 *   The rules web hook that is being deleted.
 */
function hook_rules_web_hook_delete($hook) {
  db_delete('mytable')
    ->condition('id', $hook->id)
    ->execute();
}

/**
```

```
 * Define default rules web hooks.
 *
 * This hook is invoked when rules web hooks are loaded.
 *
 * @return
 *   An array of rules web hooks with the hook names as keys.
 *
 * @see hook_default_rules_web_hook_alter()
 */
function hook_default_rules_web_hook() {
  $hook = new EntityDB(array(), 'rules_web_hook');
  $hook->name = 'test';
  $hook->label = 'A test hook.';
  $hook->active = TRUE;
  $hook->variables = array(
    'node' => array(
      'type' => 'node',
      'label' => 'Content',
    ),
  );
  $hooks[$hook->name] = $hook;
  return $hooks;
}

/**
 * Alter default web hooks.
 *
 * @param $hooks
 *   The default hooks of all modules as returned from
 *   hook_default_rules_web_hook().
 *
 * @see hook_default_rules_web_hook()
 */
function hook_default_rules_web_hook_alter(&$hooks) {

}

/**
 * @}
 */
```

Rules remote sites

```
/**
 * @addtogroup rules_hooks
 * @{
 */

/**
 * Define a remote endpoint type.
 *
 * This hook may be used to define a remote endpoint type, which users may
```

```
 * use for configuring remote sites.
 *
 * @return
 *   An array of endpoint type definitions with the endpoint type names as keys.
 *   Each definition is represented by another array with the following keys:
 *   - label: The label of the endpoint type. Start capitalized. Required.
 *   - class: The actual implementation class for the endpoint type. This class
 *     has to implement the RulesWebRemoteEndpointInterface. Required.
 *
 * @see hook_rules_endpoint_types_alter()
 * @see RulesWebRemoteEndpointInterface
 */
function hook_rules_endpoint_types() {
  return array(
    'rules_web_hook' => array(
      'label' => t('Rules Web Hooks'),
      'class' => 'RulesWebRemoteEndpointWebHooks',
    ),
  );
}

/**
 * Alter remote endpoint type definitions.
 *
 * @param $types
 *   The remote endpoint type definitions as returned from
 *   hook_rules_endpoint_types().
 *
 * @see hook_rules_endpoint_types()
 */
function hook_rules_endpoint_types_alter(&$types) {

}

/**
 * Act on rules web remote sites being loaded from the database.
 *
 * This hook is invoked during rules web remotes loading, which is handled by
 * entity_load(), via the EntityCRUDController.
 *
 * @param $remotes
 *   An array of rules web remote sites being loaded, keyed by id.
 */
function hook_rules_web_remote_load($remotes) {
  $result = db_query('SELECT id, foo FROM {mytable} WHERE id IN(:ids)', array(':
      ids' => array_keys($remotes)));
  foreach ($result as $record) {
    $remotes[$record->id]->foo = $record->foo;
  }
}

/**
 * Respond to creation of a new rules web remote site.
 *
 * This hook is invoked after the rules web remote is inserted into the
 * database.
 *
 * @param RulesWebRemote $remote
 *   The rules web remote site that is being created.
```

```
 */
function hook_rules_web_remote_insert($remote) {
  db_insert('mytable')
    ->fields(array(
      'id' => $remote->id,
      'my_field' => $remote->myField,
    ))
    ->execute();
}

/**
 * Act on a rules web remotes being inserted or updated.
 *
 * This hook is invoked before the rules web remote site is saved to the
 * database.
 *
 * @param RulesWebRemote $remote
 *   The rules web remote site that is being inserted or updated.
 */
function hook_rules_web_remote_presave($remote) {
  $remote->myField = 'foo';
}

/**
 * Respond to updates to a rules web remote.
 *
 * This hook is invoked after the remote site has been updated in the database.
 *
 * @param RulesWebRemote $remote
 *   The rules web remote site that is being updated.
 */
function hook_rules_web_remote_update($remote) {
  db_update('mytable')
    ->fields(array('my_field' => $remote->myField))
    ->condition('id', $remote->id)
    ->execute();
}

/**
 * Respond to a remote site deletion.
 *
 * This hook is invoked after the remote site has been removed from the
 * database.
 *
 * @param RulesWebRemote $remote
 *   The rules web remote site that is being deleted.
 */
function hook_rules_web_remote_delete($remote) {
  db_delete('mytable')
    ->condition('id', $remote->id)
    ->execute();
}

/**
 * Define default rules web remote sites.
 *
 * This hook is invoked when remote sites are loaded.
 *
 * @return
```

```
 *   An array of rules web remote sites with the remote site names as keys.
 *
 * @see hook_default_rules_web_remote_alter()
 */
function hook_default_rules_web_remote() {
  $remote = new RulesWebRemote();
  $remote->name = 'master';
  $remote->label = 'The master site.';
  $remote->url = 'http://master.example.com';
  $remote->type = 'rules_web_hook';
  $remotes[$remote->name] = $remote;
  return $remotes;
}

/**
 * Alter default remote sites.
 *
 * @param $remotes
 *   The default remote sites of all modules as returned from
 *   hook_default_rules_web_remote().
 *
 * @see hook_default_rules_web_remote()
 */
function hook_default_rules_web_remote_alter(&$remotes) {

}

/**
 * @}
 */

/**
 * Interface for remote endpoints. In case of any errors the implementing
 * classes should throw exceptions.
 */
interface RulesWebRemoteEndpointInterface {

  public function __construct(RulesWebRemote $remote);

  /**
   * Load remote data.
   */
  public function load($type, $id);

  /**
   * An array of definitions for the provided events.
   */
  public function events();

  /**
   * Subscribe to a remote event.
   */
  public function subscribe($event);

  /**
   * Unsubscribe from a remote event.
   */
  public function unsubscribe($event);
```

```
  /**
   * An array of info about entity types used by the provided
   * events/conditions/actions.
   */
  public function entities();

  /**
   * An array of info about data types used by the provided events/conditions/
   * actions being not entities.
   */
  public function dataTypes();

  /**
   * An array of definitions for the provided actions.
   */
  public function actions();

  /**
   * An array of definitions for the provided conditions.
   */
  public function conditions();

  /**
   * Allows altering the configuration form of remote site definitions, such
   * that the form can include endpoint type specific configuration settings.
   */
  public function formAlter(&$form, &$form_state);

}
```

APPENDIX **C**

References

[A.09] Bruckner A. Tool supported workflow integration of restful web services. Master's thesis, Vienna Univserity of Technology, 2009.

[AC08] B. Adida and C. Commons. Rdfa in xhtml: Syntax and processing. http://www.w3.org/TR/2008/REC-rdfa-syntax-20081014/, October 2008. Accessed 09-02-2010.

[BB08] R. Battle and E. Benson. Bridging the semantic web and web 2.0 with representational state transfer (rest). *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(1):61–69, 2008.

[BBB+07] B. Berstel, P. Bonnard, F. Bry, M. Eckert, and P. Patrânjan. Reactive rules on the web. *Lecture Notes in Computer Science*, 4636:183, 2007.

[BE06] F. Bry and M. Eckert. Twelve theses on reactive rules for the web. *Current Trends in Database Technology–EDBT 2006*, pages 842–854, 2006.

[BEGP06] F. Bry, M. Eckert, H. Grallert, and P. L. Patrânjan. Reactive web rules: A demonstration of xchange. 2006.

[BKPP07] H. Boley, M. Kifer, P. Patrânjan, and A. Polleres. Rule interchange on the web. *Lecture Notes in Computer Science*, 4636:269, 2007.

[BL94] T. Berners-Lee. Rfc 1630, universal resource identifiers in www. *Request for Comments, The Internet Society, June*, 1994.

[BM05] F. Bry and M. Marchiori. Ten theses on logic languages for the semantic web. 27:28, 2005.

[Bos95] J. Bosch. Language support for design patterns. 1995.

[Bra07] S. Bratt. Semantic web, and other technologies to watch. http://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/, 01 2007. Accessed 26-01-2010.

[Bro09] P. Browne. *JBoss Drools Business Rules*. Packt Publishing, 2009.

[Buy09] D. Buytaert. Whitehouse.gov using drupal. http://buytaert.net/whitehouse-gov-using-drupal, 10 2009. Accessed 09-02-2010.

[CCPD08] S. Corlosquet, R. Cyganiak, A. Polleres, and S. Decker. Rdfa in drupal: Bringing cheese to the web of data. *5th Workshop on Scripting and Development for the Semantic Web*, 2008.

[CDC+09] S. Corlosquet, R. Delbru, T. Clark, A. Polleres, and S. Decker. Produce and consume linked data with drupal! *Proceedings of the 8th International Semantic Web Conference*, 2009.

[cor10a] Core developer summit, san francisco. http://sf2010.drupal.org/conference/core-developer-summit, 04 2010. Accessed 12-05-2010.

[Cor10b] S. Corlosquet. Rdfa in drupal 7. http://groups.drupal.org/node/44094, 2010. Accessed 09-02-2010.

[DrA10] Drupal 7 alpha 1. http://drupal.org/drupal-7.0-alpha1, 2010. Accessed 09-02-2010.

[DrC] Drupal chaos tool suite module. http://drupal.org/project/ctools. Accessed 17-05-2010.

[DRD] Drupal rules module documentation. http://drupal.org/node/298476. Accessed 11-02-2010.

[DrE09] Drupal issue introducing the concept of entities. http://drupal.org/node/460320, 2009. Accessed 09-02-2010.

[DrFa] Drupal fieldtool module. http://drupal.org/project/fieldtool. Accessed 12-05-2010.

[DrFb] Issue introducing rules integration for flags. http://drupal.org/node/298109. Accessed 10-05-2010.

[DRM] List of modules extending the rules module. http://groups.drupal.org/rules/rules-modules. Accessed 11-02-2010.

[dro09] Drools documentation 5.0.1.final. http://downloads.jboss.com/drools/docs/5.0.1.26597.FINAL/, May 2009. Accessed 24-05-2010.

[Dro10] M. Droste. Automatisierte workflows mit drupal. *PHP User, Webtech Magazin*, 5:56, 2010.

[DrR] Issue for the support of looping and data lists. http://drupal.org/node/329500. Accessed 10-05-2010.

[DrR09a] Drupal rules module. http://drupal.org/project/rules, 2009. Inspected version: 6.x-1.1.

[DrR09b] Issue for providing rules integration for the taxonomy. http://drupal.org/node/256748, 2009. Accessed 10-05-2010.

[DrS] Drupal services module. http://drupal.org/project/services. Inspected versions 6.x-2.0-beta1 and 6.x-0.15.

[DrT] Drupal issue for including the token module in core. http://drupal.org/node/113614. Accessed 12-02-2010.

[DrT09] Drupal token module. http://drupal.org/project/token, 2009. Inspected version: 6.x-1.12.

[Drua] Drupal api reference. http://api.drupal.org/api/7. Accessed 19-05-2010.

[Drub] Drupal system requirements. http://drupal.org/requirements. Accessed 10-05-2010.

[DrU10a] Drupal usage statistics. http://drupal.org/project/usage, January 31st 2010. Accessed 09-02-2010.

[Dru10b] Services module development roadmap. http://groups.drupal.org/node/45636, 01 2010. Accessed 21-05-2010.

[DrV09] Drupal views module. http://drupal.org/project/views, 2009. Inspected version 6.x-2.8.

[DrW] Drupal workflow-ng module. http://drupal.org/project/workflow_ng. Accessed 11-02-2010.

[Fie00] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, Citeseer, 2000.

[Gro00] The Business Rules Group. Defining business rules - what are they really? http://www.businessrulesgroup.org/first_paper/br01c0.htm, July 2000. Accessed 25-01-2010.

[GTW10] D. Guinard, V. Trifa, and E. Wilde. Architecting a mashable open world wide web of things. *month*, 2010.

[Hoj07] G. Hojtsy. Multilingual web applications with open source systems. Master's thesis, Budapest University of Technology and Economics, 2007.

[jbo10] Jboss enterprise soa platform 5.0 documentation. http://www.redhat.com/docs/en-US/JBoss_SOA_Platform/, April 2010. Accessed 01-06-2010.

[Jos08] N. Josuttis. Soa in der praxis. *System-Design für verteilte Geschäftsprozesse, Heidelberg*, 2008.

[KDBBF05] M. Kifer, J. De Bruijn, H. Boley, and D. Fensel. A realistic architecture for the semantic web. *Lecture Notes in Computer Science*, 3791:17–29, 2005.

[KSA+09] S. Knox, J. Stahl, M. Aspeli, D. Convent, D. Hanning, R. Newbery, J. DeStefano, C. Parker, A. Clark, and V. Williams. *Practical Plone 3: A Beginner's Guide to Building Powerful Websites*. Packt Publishing, 2009.

[MAA05] W. May, J. Alferes, and R. Amador. Active rules in the semantic web: Dealing with language heterogeneity. ruleml, 2005.

[Mic06] B. Michelson. Event-driven architecture overview. *Patricia Seybold Group*, 2006.

[NL04] E. Newcomer and G. Lomow. *Understanding SOA with Web Services (Independent Technology Guides)*. Addison-Wesley Professional, 2004.

[NRD06] C. Nagl, F. Rosenberg, and S. Dustdar. Vidre–a distributed service-oriented business rule engine based on ruleml. pages 35–44, 2006.

[Né10] K. Négyesi. Field api tutorial. http://drupal.org/node/707832, 2010. Accessed 09-02-2010.

[Pap08] M. P. Papazoglou. *Web services: principles and technology*. Pearson Prentice Hall, 2008.

[Pat05] P. L. Patrânjan. The language xchange: A declarative approach to reactivity on the web. *Unpublished doctoral dissertation, Ludwig-Maximilians-Universität München, Germany*, 2005.

[Pet09] J. Petsovits. Nondestructive generic data transformation pipelines. Master's thesis, Vienna University of Technology, 2009.

[PK08] A. Paschke and A. Kozlenkov. A rule-based middleware for business process execution. 2008.

[plo] Plone community developer documentation, events and rules. http://plone.org/documentation/manual/plone-community-developer-documentation/events-and-rules/view. Accessed 02-06-2010.

[pub10] Pubsubhubbub core 0.3 – working draft. http://pubsubhubbub.googlecode.com/svn/trunk/pubsubhubbub-core-0.3.html, February 2010. Accessed 11-06-2010.

[RD05] F. Rosenberg and S. Dustdar. Business rules integration in bpel-a service-oriented approach. 2005.

[RPB+06] I. Romanenko, A. Prof, F. Bry, B. Prof, P. Patrânjan, and A. Januar. Use cases for reactivity on the web: Using eca rules for business process modeling. *Master's thesis, Inst. for Informatics, Univ. of Munich, Germany*, 2006.

[RR07] L. Richardson and S. Ruby. *Web Services mit REST*. O'Reilly Germany, 2007.

[Sch04] S. Schaffert. Xcerpt: A rule-based query and transformation language for the web. *Unpublished doctoral dissertation, Ludwig-Maximilians-Universität München, Germany*, 2004.

[SG04] S. Spreeuwenberg and R. Gerrits. Business rules in the semantic web, are there any or are they different. *Reasoning Web. LNCS*, 4126:152–163, 2004.

[Sha09] M. Shariff. *Alfresco Enterprise Content Management Implementation*. Packt Publishing Ltd., 2009.

[sit09] Sitecore cms 6.1 rules engine cookbook. http://sdn.sitecore.net/upload/sitecore6/61/rules_engine_cookbook_sc61_a4.pdf, Aug 2009. Accessed 01-06-2010.

[TW01] K. Taveter and G. Wagner. Agent-oriented enterprise modeling based on business rules. *Lecture notes in computer science*, pages 527–540, 2001.

[vD09] J. K. van Dyk. *Das Drupal-Entwicklerhandbuch: Der Praxisleitfaden für Drupal-basierte Webprojekte*. Addison Wesley Verlag, 2009.

[Wag02] G. Wagner. How to design a general rule markup language. 2002.

[WATB04] G. Wagner, G. Antoniou, S. Tabet, and H. Boley. The abstract syntax of ruleml-towards a general web rule language framework. page 631, 2004.

[Web] Webhooks.org wiki. http://wiki.webhooks.org/FrontPage.2010-04-26-18-10-49. Accessed 13-05-2010.

[YBCD08] J. Yu, B. Benatallah, F. Casati, and F. Daniel. Understanding mashup development. *IEEE Internet Computing*, pages 44–52, 2008.

[Zie08] W. Ziegler. Rules presentation at the drupalcon szeged. http://szeged2008.drupalcon.org/program/sessions/rules-new-opportunities-site-builders, August 2008. Accessed 11-02-2010.