

---

# Towards Reactive Information Systems and their Services

---

MASTER THESIS

*Author:*  
Dominic BOSCH

*Supervisors:*  
Prof. Dr. Helmar BURKHART  
Dr. Martin GUGGISBERG

June 19, 2014



---

# Abstract

The Web is a rapidly growing information universe, consisting of Information Systems that provide access over heterogeneous services. The majority of those Information Systems are dynamic and changes in their Information Space can be modeled as events. Apart from modeling such changes as events to be detected, they can also be modeled as actions when imposed onto the Information Space. If appropriate services exist to access such an Information System for read and write operations, we are able to orchestrate it. By modeling Information Space changes as events and actions, we are able to introduce an event-based conceptual model, which enables the detection of events and the dispatching of actions according to predefined rules, thus imposing reactivity on top of the involved Information Systems. Since our model bases on Information Systems and their services, it is not limited to the Web, but can include any in a way accessible Information System. In our work we introduce a prototype system which uses the Web's programmability to impose reactivity onto the Web. Through our prototype it is possible to orchestrate Web resources based on an event-driven architecture, a method which pushes towards the vision of real-time reactive Information Systems.

---

# Acknowledgment

I would like to express my deep gratitude to my master thesis advisor, Prof. Dr. Helmar Burkhart, for his patience and constant encouragement to keep going further. I also want to thank Dr. Martin Guggisberg for his ever helpful advices, hints to existing technologies and guidance through the whole process of this thesis. Besides my advisors, I would like to thank the members of the research group: Danilo Guerrero, Antonio Maffia, Alexander Gröflin and Robert Frank for their help, many interesting discussions and inputs. My thanks also go to Manus Bonner for reading through this and coming up with well formed sentences. Finally I would like to thank my family; my parents Silvia and Peter Bosch, my sister Svenja, her husband Andreas Buser, their seven months old son Nico and my girlfriend Kathrina for their inspiring ideas, the joyfulness, irresistible smiles and financial support they gave me on my journey.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Related Work</b>	<b>3</b>
2.1	Data and Functionality Providers on the Web . . . . .	3
2.2	Reactivity through Event-Condition-Action Rules . . . . .	7
<b>3</b>	<b>Conceptual Model for Reactive Information Systems and their Services</b>	<b>12</b>
3.1	From Physical Events to Virtual Events . . . . .	13
3.2	Capturing Events from Information Systems . . . . .	13
3.3	Event Pattern Detection . . . . .	14
3.4	Imposing Reactivity to Information Spaces . . . . .	14
<b>4</b>	<b>Use Cases for Reactive Information Systems</b>	<b>16</b>
4.1	Reacting on changes in the World Wide Web . . . . .	16
4.2	Enhance existing Web Applications . . . . .	16
4.3	Service Functionality and Availability Checking . . . . .	18
4.4	Exploiting the Web of Things . . . . .	19
4.5	Multi-Sourced Bad Weather Prediction . . . . .	19
<b>5</b>	<b>Prototype System</b>	<b>21</b>
5.1	Architecture . . . . .	21
5.2	Rule Language for Prototype System . . . . .	25
5.3	Example Use Cases . . . . .	25
5.4	Web Programming . . . . .	25
<b>6</b>	<b>Conclusions &amp; Future Work</b>	<b>29</b>
	<b>Bibliography</b>	<b>29</b>
	<b>Appendices</b>	<b>36</b>

# Chapter 1

## Introduction

The Web is an ever growing institution, in all aspects that it covers. The number of data and functionality providing Web Resources is growing in the whole spectrum from bigger computing centers down to smaller devices. Computing centers are growing in size and quantity and they allow for massive amounts of data to be stored and accessed, moreover they also enable the construction and offering of more complex functionality. At the same time, a quickly increasing number of ever smaller devices also provides more resources to the Web. Many of them are providing access to the Web itself, granting even more devices access and thus leverage the effect of the growing Web, e.g. mobile phones can act as a hotspot to grant Web access to other devices over WiFi. A recent trend that can be observed is also all the smart things, which are gaining access to the Web and start to form the Web of Things. Today, these smart things can be everything from a temperature sensor to all the electronic devices within a house. They do not only provide sensor data but they can also be controlled over the Web. All these different types of services available on the Web make it a heterogeneous collection of data and functionality. Great efforts are made to turn them into uniformly accessible resources, e.g. the Semantic Web is a widely supported initiative towards a machine-readable, structured and semantically described Web.

Confronted with this rapid growth of the Web, an increasing number of human beings is exposed to it in their daily life, and they get literally flooded with informations and means to retrieve them. Even though users have access to so much data and functionality in the Web, they often lack the knowledge, necessary time or right approach to weave them together. It would be of great value for them to automatically get appropriate informations, in the right moment and in a condensed matter that supports them best. They should be able to automate tedious tasks, e.g. detecting relevant changes in their preferred data resources and react on behalf of such changes. This requires the identification of and filtering for user-relevant changes, appropriate timing, assembly and processing of additional data and finally the placement of the outcome in the user's preferred context on the Web.

With the many existing services on the Web, users don't want to be bound to specific ones for certain tasks, as it is often the case nowadays. They want to use the functionality or data of their preferred service, which helps them best to fulfill their needs. Hence, users should be able to create their own specific but still flexible Web Resources compositions. Since the need of users to compose different services has gotten a lot of attention, some of the services on the Web offer ways to spread their data to others, but in a limited way. For example it is common for social network applications to push user-specific notifications to other social networks, e.g. singing in at a place in Foursquare can also be posted directly to the Facebook timeline. Because of existing limitations, such as customizability or free choice of receiving service, users still end up

mixing data and functionality from different resources on the Web by hand, which often means to execute similar tasks repeatedly by hand. Moreover the manual reaction on changes is deferred because the changes are not detected in real-time by the user or because the user is not able to react in a timely fashion. Also if a certain workflow could be split up into several automated sub tasks, it is likely that some parts could be reused for other workflows or even by other people to get similar work done.

Since data and functionality already exist in the Web, the users are theoretically enabled to automate their work to some extent by orchestrating those Web Resources. Even though the access to resources gets simpler, the average user is still not capable to fully exploit the Web's full potential. Another challenge is, that often a lot of effort has to be made, in understanding how the specific resource works, before it can be fully exploited. There is a lot of research that goes towards an easy to orchestrate the Web, but they are either often complicated to wield themselves, mere data copy tasks or static resource compositions. Our goal is to enable user-defined resource composition, and still exploiting their full potential by not limiting the set of their functionality.

A big part of the data, that becomes available to the users, is short-lived data that corresponds to state changes, and which can be modeled as events. In this thesis we introduce an event-driven conceptual model that uses the programmability of the Web and imposes reactivity to it. We claim the whole Web as our information space, in which we listen for triggered events and in which we execute actions as a result wherever possible. Such a user-specific reactivity allows a personalization of the Web and a tool to govern its data flood by automating tedious tasks. It allows users to orchestrate the Web and to tailor reactivity to their needs. Current Web Resource orchestrations concentrate on data flow rather than on event flow, which are mere copy/paste tasks of data than smart reactivity. This makes us believe that our event-based conceptual model can overcome certain shortcomings of the existing approaches and provides a step towards the reactive Web.

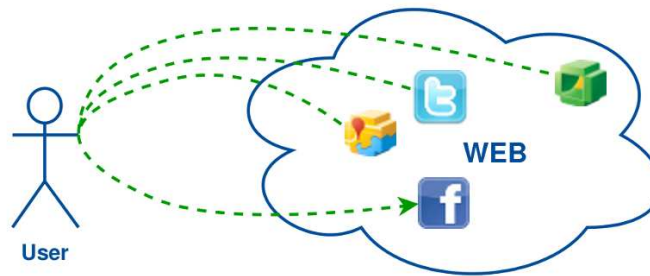


Figure 1.1: Users orchestrate the Web's Data and Functionality

## Chapter 2

# Related Work

In this chapter we will give a brief overview over a certain kind of Web Resource collections, the Web services, which are of particular interest to our vision. We then introduce the Event-Condition-Action(ECA) paradigm, an event-driven approach to impose reactivity. And finally we point out existing rule languages and engines that exploit the ECA paradigm.

### 2.1 Data and Functionality Providers on the Web

Data and Functionality Providers on the Web are in general referred to as Web service. But the term service in the context of the Web is somewhat ambiguous and there have been a lot of completely different approaches to offer services within the Web, some of the latest used in cloud computing are Platform as a service (PaaS), Software as a service (SaaS) and Infrastructure as a service (IaaS). The term Web Service (capitalized word Service) commonly stands for Web service based on Simple Object Access Protocol (SOAP) communication[5], which has been adopted and developed extensively by the industry. With the advent of the Representational State Transfer (REST) architectural paradigm, the understanding of the term service in the Web has undergone a slight generalization so that Web service (lower-case word service) is not anymore bound to a certain protocol, but describes services as interfaces for communication between applications over a network[36]. We will point out some main research areas on service-orientation within the Web and point out how they make the Web programmable.

Remote execution of programs on other computers has always been a strong research area, ever since computer started to exist. After the coinage of the term World Wide Web[6], the Web has become a synonym for Berners-Lee's vision of a global information universe. The trend towards remote program execution offered through the Web followed immediately; computers waiting in the Web for a request in order to execute some application logic and return an answer. Familiar to this concept is the encapsulation of functionality into services[31] in order to offer them to other applications, which is called Service-Oriented Architecture (SOA)[32]. Applying SOA to an existing application means splitting it into smaller loosely-coupled pieces (services), which then communicate with each other. This does not only provide robustness, it also allows the reuse of functionality. Moreover these services can be offered to other applications and also to the Web, thus allowing others to access certain functionality or even the whole application behind the services. All nodes in the Web are stand-alone entities, which offer services of some sort, be it a webpage, pure data, instant measurements or functionality of some sort. This makes the Web itself a Service-Oriented Architecture and all these services are naturally Web services. It is because of its advantages that SOA has received a great deal of attention and has been widely

adopted throughout the Web. This led to an increasing number of Web accessible services and their compositions, the so called Mashups. An empirical study[24] on a directory, which the researchers of the paper call the "[...] *most active Web APIs and mashups collection*", and statistical data taken from this directory (depicted in Figure 2.1) seem to underline a growing popularity, at least in terms of published services within this directory. A server-side Web API is a service interface to an application, meant for application to application communication.

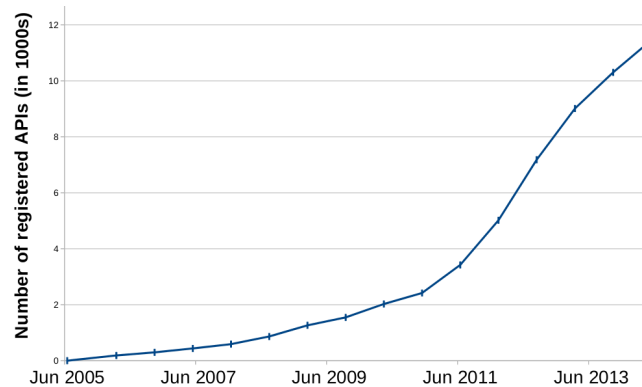


Figure 2.1: Number of registered APIs in the ProgrammableWeb directory by date

### 2.1.1 Accessing services on the Web

An early adoption of the service concept to computers were the Remote Procedure Calls (RPC)[8]. Through RPC a piece of code can be executed on a different machine, other than the one which is calling the procedure. It is basically an inter-process communication and doesn't necessarily require the Web nor distributed computers. Also because of when RPC was invented, the term World Wide Web didn't even exist. RPC also found its use in grid computing[39] and through this, opened doors into the field of distributed computation. The RPC paradigm isn't bound to certain technologies and thus, has been implemented in a lot of different programming languages. These implementations were tightly bound to the respective language that was used, which resulted in incompatibility among them. It became necessary to enhance RPC's in order to get cross platform compatibility. The abstraction of RPC through the Extensible Markup Language (XML)[11], which is called XML-RPC, made it easier to achieve compatibility between services that used different technologies.

Since XML-RPC was held relatively simple but received a lot of attention, it was further enhanced. Together with additional proposed functionality, XML-RPC was the base for Simple Object Access Protocol (SOAP)[10]. SOAP is accompanied by the Web Service Description Language (WSDL)[15] which is used to describe the interfaces to SOAP services, the Web Services. Through SOAP and WSDL a client for the service can issue a request for the WSDL information of the service and retrieves all interface specifications he requires in order to issue a call to the actual service. The service specifications are then incorporated into the existing application as if it is a local function call. SOAP has found its applicability in business applications[5] and was enhanced with a lot of industrial standards, also called the "WS-\*" specifications, e.g. WS-Addressing, WS-Policy or WS-Security.

Another initiative that aimed for eased communication between different platform is the Common Object Request Broker Architecture (CORBA)[22]. As the name already suggest it is an object-oriented approach and it allows the exchange of whole objects. CORBA relies on its communication layer, the Object Request Broker (ORB), which forms the basis of



its architecture. The platform-specific ORBs provide the communication abstraction, which free the application from platform dependencies. Similar to SOAP's WSDL, CORBA has its Interface Definition Language (IDL) to provide information about the objects to be offered and accessed. An object is instantiated by an application and the interface to this instance is offered through the ORB. Another application attached to the ORB can then access all public variables, data structures and functions of this object. This means not only remote access to variables and data structures, but also remote function invocation as seen in RPCs. CORBA requires the implementation of object-oriented mechanisms in programming languages which aren't object-oriented. This can be technically difficult and become an eventually tedious task. CORBA allows communication between applications written in different programming languages and which are running on the same physical computer, as well as the communication between different computers in the same network. With the Internet Inter-ORB Protocol (IIOP) it is also possible to connect ORB's over the Web. Through this, the offered objects can become services in the Web, but they are shielded by the ORB.

### **2.1.2 Services on the Web become Web Resources**

All the afore mentioned approaches require a specific protocol and are therefore incompatible with each other. For this reason and its simplicity, an architectural style has gained popularity which frees application from this constraint: Representational State Transfer (REST)[19]. REST concentrates on the roles of components and on constraints upon interactions between them. An important architectural constraint is that all communication is stateless, which means for a client-server communication, no state is stored on the server. Therefore all informations required for a single interaction need to be provided within one request. This allows for the definition of simple and well-defined interfaces, since responses are not bound to a certain session state. Services within the Web that adhere to the REST architecture are called RESTful Web services. RESTful Web services provide access to their data and functionality through grouped Web Resources, which can be identified via Uniform Resource Identifiers (URI)[26]. In the Semantic Web[7], a Web Resource is anything in the Web that can be identified, addressed and handled. This started with documents and went over objects to abstract concepts, such as operators of equations. Simple access to Web services without communication overhead and negotiation before using it, increased REST's popularity and spread it into more application fields. There is for example the upcoming concept of the Web of Things[23], which aims to incorporate smart things (e.g. tagged things, sensor measurements, device controllers, etc.) into the Web through REST interfaces. REST brings advantages into the context of smart things connected to the Web, because incompatible standards and protocols were used by different manufacturers of such things and often their nature is already compatible with the REST architectural constraints.

### **2.1.3 Composing Services in the Web**

Webpages emerged into dynamic sites on the web through the upcoming of scripting languages to control the browser and the webpage itself. With all their server-sided infrastructure in the background they became literally applications, with more or less functionality and persistence on a server. These Web Applications (Web Apps) became even more responsive with the advent of asynchronous calls from the browser to the server, which allows to load data into the current webpage while the user is interacting with it. Those asynchronous calls are requests to services, which act as the application programming interface to the Web App (Web API) which sits on the server. As a side-note, the term Web API not only comprises server-side interfaces but also client-sided ones (e.g. the browser), after all they are also interfaces to the Web. For server-side

Web APIs this means that these services can be accessed from other entities in the Web than just browsers, which eases application to application communication. Often the model behind a Web App can be controlled without the Web App itself, depending on how fat the server-side and how thin the client-side is. Imagine not going to the Google webpage anymore to make a search and manually crawling through the results, but you have your own application doing it for you and processing the results instantly. There is a trend of Web App providers to publish their Web API in order to grant easy access to it. This lead to an increasing number of Web App Mashups in the past few years.

Mashups combine data and functionality of more than one service in the Web in a new site. Simple services from different sources can be combined into more powerful ones, which can in turn again be composed, and so on. These service compositions assemble data and services in a novel way which provides a new perspective. Ever since services were accessible in a more or less convenient way, Mashups have been developed as well. One of the first Web service Mashups[34], was invented in the same year after Google Maps came up in 2005. It was a webpage that displayed Craigslist's rental houses on a Google Map. At that time no Web API was available that provided easy access to these two services. But there was an added value to be observed from anybody being able to create a Mashup through publicly available services, because this leads to new ideas and an increase of popularity in all incorporated Web services. Such Mashups are often a read-only fixed wiring of different services that provide a new view on specific data. Some recent Mashup examples are:

- Wifi and Plugs: MapBox, Google Docs and Import.io API's used to display where Wi-Fi and plugs are available in London.
- MapLight: GovTrack.us and OpenSecrets API's used to combine political results with financial contributions, in order to show how capital contributions to certain campaigns influence voting.
- Shared Count: Facebook, LinkedIn, Pinterest and Twitter API's used to display informations about how well spread a URL is on social media sites.

But also a number of studies[14][25][37][40] made efforts towards personalized Mashups, where users are capable of choosing what and how to link in order to enhance Web Resources according to their needs. These flexible Mashup applications often provide methods to access user-specific functionality within external Web Apps, which makes them even more user-centered and customizable.

#### 2.1.4 Subscribing to Web Resources

There is another type of service in the Web which is about the opposite of the afore mentioned approaches in terms of the data flow. It is the concept of push notifications on state changes and it is a recent and active research area. There are some manifestations of this model for browser-server communication, such as Comet[16] or Server-Sent Events<sup>1</sup>. Webhooks are a method that introduces the asynchronous delivery of data whenever it gets available, compared to the need of actively requesting a service to deliver it. Webhooks are URIs, which point to a service in the Web, that accepts the data delivered to it. Within the publish/subscribe paradigm[18], such asynchronous delivery of data is referred to as events, since that's what the appearance of new data is. Webhooks are callbacks that can be placed by a Web service provider at a remote Web API, informing this distinct event delivering service about the interest in the

<sup>1</sup><http://dev.w3.org/html5/eventsource/>

promised events. Both parties are services in the Web, since the Webhook providers accept the data delivered to their URI and the Webhook receiver offer to send the data. PubSubHubbub<sup>2</sup> is an open server-to-server publish/subscribe protocol that uses Webhooks for servers to address their interest in updates from other servers. Only through such push notifications a reactive system can be instantly reactive through real-time event detection.

### 2.1.5 Towards Simple Access and Communication

With JavaScript's success as browser scripting language and recently also as server-side programming language, JavaScript Object Notification (JSON) as an alternative to XML has become popular for data representation throughout the Web. It is also because of its human-readable format and often simple parsing into data structures of existing programming languages. There is a notable trend towards RESTful services in the Web that offer JSON communication. They benefit from simple but powerful interfaces and easy to debug human-readable communication, which eases integration into other applications, along with the reduced communication volume. Together with client- and server-side Web APIs the Web becomes ever more programmable.

## 2.2 Reactivity through Event-Condition-Action Rules

In this chapter we have so far shown research in different areas that lead towards a programmable Web. As a result of this research, it is getting easier to compose and orchestrate services in the Web, but reactivity needs to be programmed specifically by experts and general approaches are only available in specific domains. Several studies[3][12][13][28][29] have been made on reactivity. They point out Event-Condition-Action (ECA) rules as the most adequate way to impose reactivity on a system. As the name already suggests it bases on an Event-Driven Architecture (EDA) and ECA rules consist of three parts:

- Event: An event identifier, that enables detection of a triggered event
- Condition: Expressions to be evaluated to determine whether an action is triggered
- Action: A set of instructions that complete the reactive behaviour

Several different rule languages have been developed for different domains. We will give a brief overview over the research done that relates to our goal, reactivity in the Web. During our research, apart from identifying the key properties of different rule languages, we analyzed them with respect to a certain use case, in order to determine their applicability for our research goal. The use case's ECA rule is:

- Event: Receipt of an Email
- Condition: Check for a certain sender
- Action: Store it remotely via a Web API

We also tried to get access to existing rule engines for each rule language, since we aim to build our model as well as our own reference implementation on top of existing work.

---

<sup>2</sup><http://code.google.com/p/pubsubhubbub/>

### 2.2.1 Rule Languages & Rule Engines

The Resource Description Framework (RDF) is a collection of specifications to model informations in the Semantic Web. Papamarkos et al. (2004) published an ECA language for RDF: RDF Triggering Language (RDFTL). It was designed to react on insert and delete events within RDF repositories and for an action notify users and propagate the changes through related resources. RDFTL bases on RDF resources which need to run engines in order to react on the rules. These engines retrieve events, detect changes and communicate them as events to other engines and actions are executed on local repositories. Through distributed engines, RDF resources can be made reactive. We envision an engine that orchestrates the Web, rather than relying on other Web sites to incorporate our model. But still their research provides important insights on reactivity through ECA rules.

The rule language XChange[30] emerged from the Reasoning on the Web with Rules and Semantics (REWERSE) project[35], which took place from 2004 to 2008. It was designed to track changes in dynamic Web Resources and add reactive behaviour in a way that such changes influence other dynamic resources. XChange incorporates the vision of distributed, event exchanging rule engines. Those rule engines execute actions on local data or issue new events. The local-only actions oppose our vision to orchestrate heterogeneous Web Resources through reactive behaviour, so does the RDFTL. Eventually a dedicated XChange rule engine could be realized, which is enhanced to translate XChange actions into communication with remote Web Resources. The use case applicability study was promising but access to a reference implementation of an engine, in order to enhance it with our vision, could not be gained. Still the thorough research done with the language XChange holds valuable concepts, especially in terms of temporal event composition.

JSON Rules[21] was introduced 2008 as a language to react on specific Document Object Model (DOM) tree states of a webpage and as reactive behaviour control the browser and also DOM again. The incorporation of script function calls into the action part of the language allows the abstraction of eventually complex action behaviour. This feature influenced our concept as it allows for different levels of complexity to be offered as the reactive system is growing. JSON Rules is bound to DOM tree events and actions, where we aim to react on any events happening in Web Resources and also execute actions on them.

The Rule Markup Language (RuleML)[9] is a language written in XML and aims to standardize many different types of rules. Reaction RuleML[29] is an enhancement of the existing standard by reactive rules. Reaction RuleML subsumes:

- Complex Event Processing (CEP)
- Knowledge Representation (KR) calculi
- Event-Condition-Action (ECA) rules
- Production (CA) rules
- Trigger (EA) rules

Reaction RuleML represents thorough research for a language to describe virtually any type of reactivity. Together with the expression in XML it doesn't score with readability, but provides a way to define a multitude of rule types and the interchangeability of them between different sites in the Web. Since our vision does not require interchangeable rules, we chose an internal JSON representation, inspired by JSON Rules, for our rules to have certain properties, such as human-readability, simple parsing and efficient storage. A notable system that relies on RuleML is Rule Responder[?]. Rule Responder connects different types of heterogeneous

rule engines together over the Mule open-source Enterprise Service Bus (ESB) which acts as a communication middleware to exchange rules expressed in RuleML.

A recent research outcome (Windley, 2011) is the Kinetics Rule Language (KRL)[41] together with the Kinetic Rules Engine (KRE). It was invented to impose reactivity to the Web and incorporates many different event origins and action resources. The language is based on declarative syntax, enriched with imperative elements. A handy feature are the ways to activate webpages, which bridges the gap between the user's browser and the centralized KRE. Either a user can install a browser plugin which will communicate with the KRE, or a webpage provider can include a library in order to get events from accesses to the webpage. Through this, events can be raised from the browser and actions can also execute in it. The KRL fits very well into our concept and only a few reasons kept us from realizing our reference implementation on top of the KRE, such as:

- complexity required to maintain states with a declarative syntax
- system footprint of the KRE
- Perl, a procedural programming language, as base of the KRE

The concept of the KRL is promising in terms of orchestrating the Web through reactivity. But we decided to implement a light weighted reference system, using an event-driven programming language that is laid out for an event-driven architecture. Another important key property of our envisioned conceptual model, which diverges from the KRE architecture, is the abstraction of state maintenance into action dispatching modules.

Table 2.1 gives an overview of the key properties of existing rule languages and their key properties for our research:

- Event Origin: Resources type from where the events originate.
- Distributed: Whether the language is laid out to run on a centralized or distributed architecture. All examined rule languages that support distributed architectures can as well run on a centralized architecture.
- Action Resource: Resource type on which actions are executed.
- Accessible Engine: Determines whether a reference implementation of an engine was accessible.
- Applicability to our concept: Names the main difference to our envisioned concept.

Rule languages that support a distributed architecture require engines to be distributed on sites that are thought to be reactive. This is not service-oriented in terms of external services and does not attempt to orchestrate the Web's heterogeneous services as action part of rules. It seems common that ECA rules only invoke actions on local systems, even though the KRL goes into the direction of accessing remote systems too.

Other examined Rule Engines were the Object-Oriented Java Deductive Reasoning Engine for the Web (OO jDrew), Prova, and Drools Fusion. They are all implemented in Java and have their own rules syntax which is more or less closely related to Java, with inline Java code. Some of them have a heavy system footprint because they base on communication middlewares or existing frameworks, such as Drools Fusion which requires a Java graphical user interface. Our decision not to use any of these existing approaches for a reference implementation was, because all of the examined research only provides limited support for our concept or has a heavy system footprint, caused by the communication layer such as the JBoss ESB. We envision a scalable system, which is event-driven from the application layer, and allows the orchestration of

Language	Event Origin	Distributed	Action Resource	Accessible Engine	Applicability to our concept
<b>RDFTL</b>	RDF Repository Changes	Yes	(Local) RDF Repository	-	Only Web sites with engines are reactive
<b>XChange</b>	Web Resources	Yes	Local Resources	-	Actions in remote Web Resources missing
<b>JSON Rules</b>	DOM Events	No	Browser / DOM	-	Only Browser / DOM Events
<b>RuleML</b>	Web Resources	Yes	Local Resource	(OO) jDrew, Prova, Rule Responder	Complex Syntax
<b>KRL</b>	Web Resources	No	Local & Remote Web Resources	KRE	User-specific Web App functionality missing

Table 2.1: Key Properties of existing Rule Languages with respect to our Conceptual Model

heterogeneous Web Resources. These resources are already available and accessible and we do not need to alter them in order to impose reactivity to the Web. Such a system does not require a messaging middleware because the Web itself acts as the communication channel to receive events and dispatch actions.

### **2.2.2 Complex Event Processing**

An important research area in Event-Driven Architectures is in the field of Complex Event Processing (CEP)[4] and deals with event composition, also called Complex Event Detection (CED)[2][33]. It is the research for methods to detect predefined event relations and also temporal patterns, over different streams of data or events. This topic has already received a lot of attention in the context of active databases[1][20][42] and was picked up again in the context of Event-Driven Architectures. With CED atomic and composite events are successively aggregated into higher-order events. The event-driven architecture allows the processing of large amounts of data and trigger detections in nearly real-time. In our conceptual model we envision a CED engine that detects complex event patterns and assembles them into higher-order events. These complex events can then be detected by the ECA rules engine to dispatch appropriate actions.

## Chapter 3

# Conceptual Model for Reactive Information Systems and their Services

The challenges and opportunities arising with the growth of the Web in terms of volume and complexity inspired our research towards the reactive Web. Therefore our starting point were the studies of related work in the context of reactivity on the Web, event composition and programmability of the Web. In the last chapter, we pointed out how they received a lot of attention and provide powerful tools to orchestrate the rapidly growing Web. By combining existing research in these fields we developed a conceptual model, which allows to impose smart reactivity to any Information Space. Even though our initial set of Information Spaces was thought to consist of Web Resources, our model is applicable to any Information System whose Information Space can be accessed and altered over interfaces, i.e. services. Thus we introduce our conceptual model for reactive Information Systems and their services in this chapter.

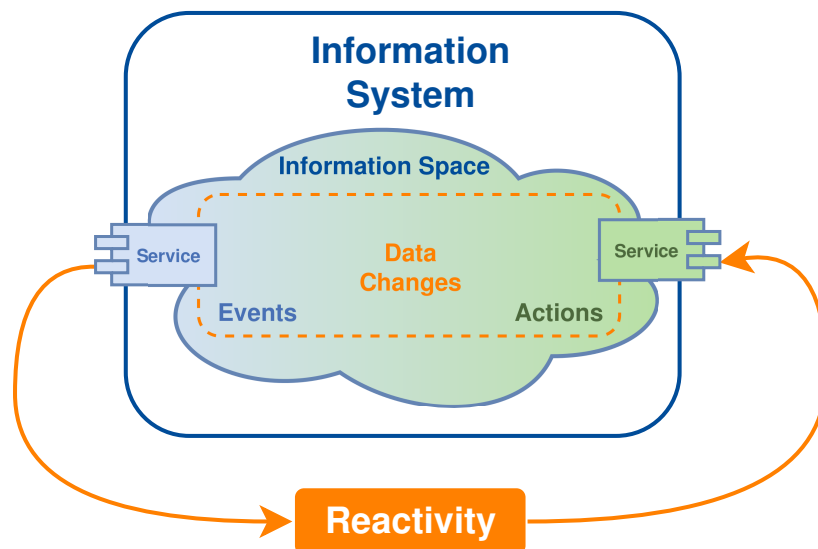


Figure 3.1: Reactivity imposed on Information Systems and their Information Space over Services

Data changes within an Information System can be detected and imposed from the outside, if proper interfaces exist, the services. We model the detection of data changes as events, and the inflicting of such changes as actions, as shown in Figure 3.1. Through this we are able to



introduce an event-based model that is capable to detect events and react on behalf of them by imposing actions on the same or another Information Space. A more precise distinction of the important modules for such a reactivity imposing entity is displayed in Figure 3.2, and we will introduce each module in this chapter.

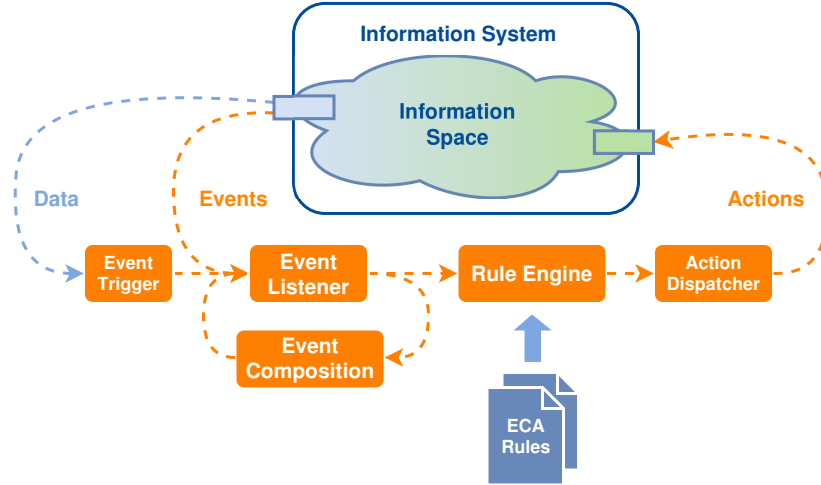


Figure 3.2: Conceptual Model for Reactive Information Systems and their Services

### 3.1 From Physical Events to Virtual Events

We introduced the Web of Things in the last chapter, where smart devices gain access to the Web. It is based on the Internet of Things which mostly dealt with the incorporation of sensor networks into the Internet on the network level. Such sensors bring the physical world directly into the virtual world. This transfer pictures an important difference between physical and virtual events. In physics, and in particular relativity, an event indicates a physical situation or occurrence, located at a specific point in space and time. While physical events correspond to a physical situation which is located at a specific point in space and time, virtual events primarily consist of implicit parameters, i.e. a name and occurrence time. These virtual events can be anywhere within Information Systems at any point in time and thus their actual location differs most certainly from its occurrence location. But virtual events also have explicit parameters which correspond to all available information about the event. As soon as events are transferred into the virtual world the afore mentioned location information turns into explicit parameters. Therefore wherever a virtual event is, it has a name, an occurrence time and most likely some explicit parameters attached to it. If the virtual event is of a physical nature, it has a physical location, if it is of a virtual nature it is likely associated with a virtual origin. Since in our model events are changes in data, they can be virtually anything, e.g. measurements, changes on a static webpage, changes of the object behind a Web service or also a login attempt.

### 3.2 Capturing Events from Information Systems

The optimal case for an event-driven system which requires events from an Information System is, that events are triggered within the Information Systems and then immediately communicated to interested external systems such as our envisioned reactivity imposing system. But our research has shown that such Information Systems are often passive and rarely provide ways for external

systems to announce interest in changes of their data. For example in the context of Web Resources, many of them provide service access to data but do not actively communicate changes to interested parties. This is where the upcoming concept of Webhooks comes into play. Only through them we are able to provide nearly real-time reactivity without risking huge costs of continuously polling for changes over all Information Spaces. We envision a future where the whole Web is event driven and events are directed to any system, which is interested in them. This would be the optimal case for effective real-time notifications and thus reactivity on the Web. But still we need to incorporate polling for changes into our model, in order to cope for the widely spread passiveness of Information Systems which is likely to never thoroughly vanish. Wherever an Information System is not capable to provide events over services, we can use all the accessible data and detect changes in it, and as a result model them as events in order to feed them into our model. In our model the polling for changes is incorporated in the Event Trigger modules. Those are flexible modules that have the proper tools to access any Information System service and therefore its Information Space and are capable of identifying changes in the data. For example the World Wide Web, as envisioned by Tim Berners-Lee[6], is an information universe of interlinked documents, that a user can browse through. Through our model, we can pull changes in the data on the World Wide Web, i.e. document changes, and turn them into events. These events which are derived from changes in the data of Information System are then fed into the Event Listener. The Event Listener also receives events directly from the Information Systems which are capable to communicate events to external systems. It queues all events and forwards them to the Event Composition module and the Rule Engine whenever they are ready.

### 3.3 Event Pattern Detection

Traditional ECA systems only react on single events, but this might often not be enough to detect meaningful situations. Events are initially primitive that occur at a point in time (e.g. a press down mouse button event). When they are composed (e.g. the latter event with a release mouse button event), they turn into a composite event which is more complex and has a duration. This is why there is a trend towards the detection of complex event patterns, as we have pointed out in the last chapter. CEP could be incorporated into the rules of the Rule Engine, which then reacts on event patterns, but this opposes our vision of a successively growing complexity of composite events that are defined on top of each other and fed back into the Event Listener. Thus in our model an Event Composition module composes events into more complex events according to CED definitions. It is beyond the scope of this thesis to dive into this complex matter, but since it is a very active research field, it has seen interesting studies[2][33] and outcomes<sup>1</sup> that could be incorporated into our model. Such an event composing service systems works loosely coupled and could be realized by any suitable system, as described in [38].

### 3.4 Imposing Reactivity to Information Spaces

In the last chapter we gave an introduction into reactivity and the ECA paradigm as an approach to achieve it. So far we have introduced the first part of our model which provides the foundation of an Event-Driven Architecture. What we now need is a module that translates events into actions on Information Systems. Almost all existing ECA system's actions write on the local Information Space which opposes our vision of the orchestration of different Information

---

<sup>1</sup>such as <http://drools.jboss.org/drools-fusion.html>

Systems in order to impose reactivity on top or between them. For that reason we introduce the Action Dispatcher modules which are located right behind the Rule Engine in terms of the data flow and complete the reactivity flow between heterogeneous Information Systems. Action Dispatcher modules are an important part of our model because they allow flexible coupling with Information System services, much like the Event Trigger modules do. Event Trigger and Action Dispatcher modules are communication abstractions to services of Information Systems, that allow us to deal with their heterogeneity in terms of communication. The Information Space of an Information System is not limited to internal data, but can also refer to a coupling with other devices and the sensing and controlling of it. Thinking of the Web of Things this can also include an Action Dispatcher that has access to an Information System which controls devices and thus is capable of turning down the heating in a house, as shown in Figure 3.3.

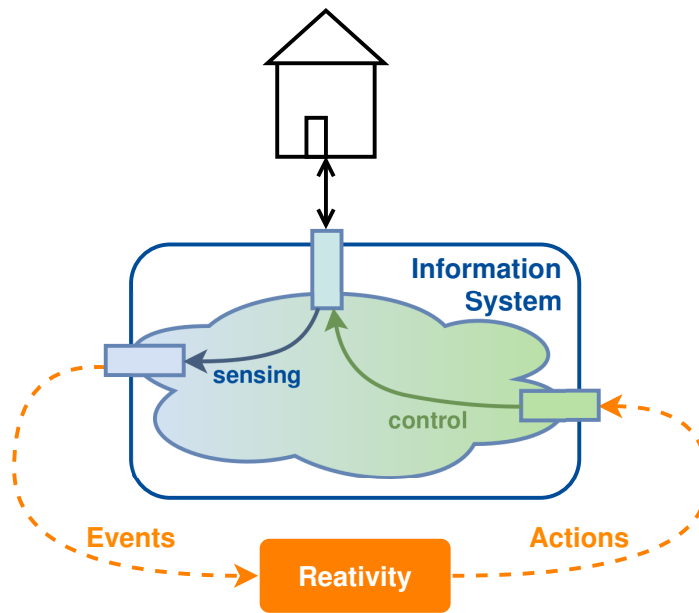


Figure 3.3: Conceptual Model for Reactive Information Systems and their Services

So far we defined all the modules to access Information Systems over their services, now we are able to define a Rule Engine that orchestrates them in a reactive way. We have shown in the last chapter that ECA rules consist of three parts; an event to be recognized, conditions to be evaluated on the event and actions to be executed if an event triggers the rule through valid conditions.

## Chapter 4

# Use Cases for Reactive Information Systems

We have so far introduced a conceptual model for reactive information systems and their services. Through our model we are able to react on events happening in Information Systems and dispatch changes to it or any other accessible Information Systems. In this chapter we will give examples of how use cases can be realized through our model.

### 4.1 Reacting on changes in the World Wide Web

Many documents of the World Wide Web are dynamic in the way that they change over time. Some might change in an interval of a few minutes (e.g. news), while others change much slower, such as knowledge sites. To detect such changes a site in the Web needs to keep track of the document history and trigger events if there is a change. In our model this would be realized by an Event Trigger which monitors a certain set of Information Spaces and triggers events as soon as differences are detected. A user will then set up a rule that checks if the changes are related to a certain category of interest, such as sports or a certain observed site and how big those changes are. If for example a Wikipedia article changed in more than 20% of its content this will be a reason for a moderator to look at it and should be entered as a task in the next free slot of his calendar.

### 4.2 Enhance existing Web Applications

Web applications, such as webmails, social networks or Content Management System (CMS) are widely spread and used by a large number of internet users. But often users or developers miss some features or interoperability with other web applications, which would result in enhanced functionality and also in less work. Features of that kind could also include data and functionality from other sites on the web. This would require web applications to mashup together and to grab data or invoke functionality on each other. A lot of enhancements will not be implemented by the web applications themselves because they are very specific to a small number of their users. With a reactive information system, users and developers could realize such features on their own.

### 4.2.1 Enrich CMS Posts with Additional Information

Every new post to a content management system can be modeled as an event. In the case that a user would like to enrich such a system with founded knowledge from a remote resource, reactivity in the Web can be used to enrich it. Enhancing an existing CMS can be realized by a rule that evaluates new posts and checks whether knowledge tags are included in the post. Whenever there are tags included within the post, the reactive system will enrich the post with additional knowledge to these tags from a remote service of the user's choice.

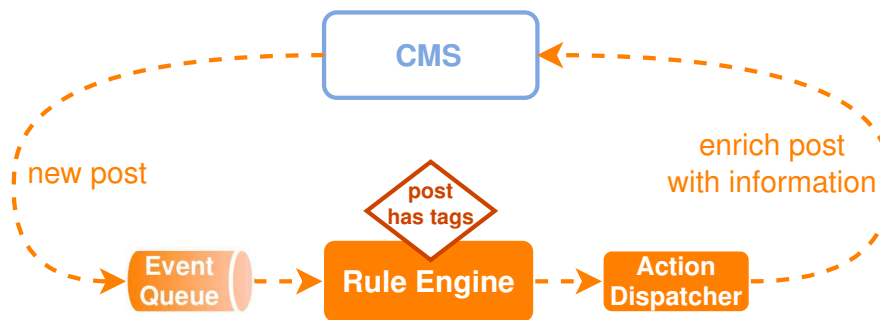


Figure 4.1: Enrich CMS Post with Remote Knowledge Data

### 4.2.2 Workflow Automation

Within such web applications, users often have very specific workflows. And because workflows always start with an event, they are predestined to be automated by a reactive information system. As an example for workflow automation, course and student exercise submission administration can be taken care of by a reactive information system. Whenever a new semester starts, the reactive system will detect this through one of its rules and command an action dispatcher to set up infrastructures for courses. This can also include grabbing course data from an official web page and including it into the infrastructure, thus eliminating the need for manual data copy tasks.

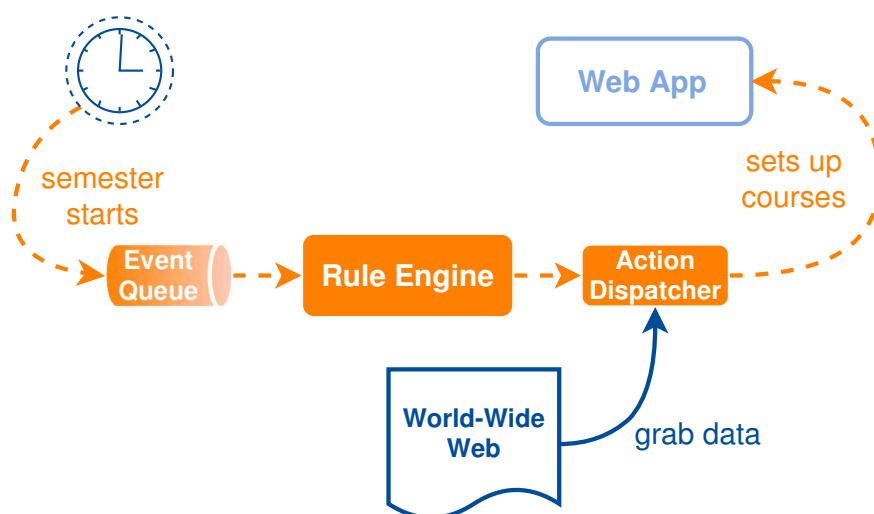


Figure 4.2: Create course resources at semester start

After setting up the semester courses, the reactive system is ready to process new student

registrations for these courses. It automatically associates students into the afore mentioned infrastructure and sets up additional infrastructure such as an exercise submission container. Whenever a course tutor submits a new exercise to the course resource, the system will detect this and spread this information to the students, together with a deadline. The students are expected to submit their exercise solutions before the deadline to the exercise submission container that was created reactively for them.

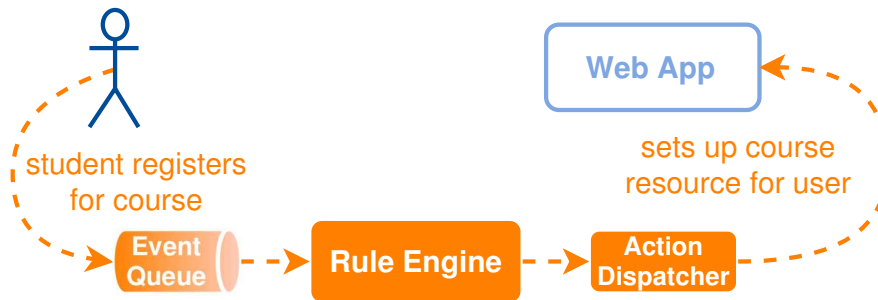


Figure 4.3: Create course resource for registered student

A certain amount of time before the deadline, e.g. one day, the reactive system will detect the deadline and process events that depict the current exercise submission status per student. If the system detects a student who hasn't uploaded her exercises yet, it will notify her about the deadline. This is an additional service that gives students the chance to react on a missed exercise submission deadline. As soon as the deadline passed, the system will revoke write-rights to the exercise submission container and therefore disallow submissions which are too late.

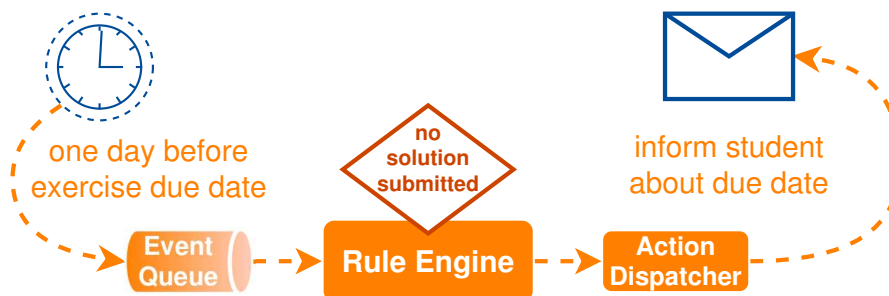


Figure 4.4: Notify student before exercise due date

### 4.3 Service Functionality and Availability Checking

Services offered through the Web aren't monitored or tested by users or developers from other sites. If they rely on correct functionality or availability they need a way to assert this. It is also possible that an owner of such a service doesn't have the tools to monitor his own services. Whenever such a service isn't working correctly anymore or stops responding, these users or developers need to be able to react on this before it is too late. With a reactive rule in place that evaluates Service Testing results, measurements can be taken early. One action to such a failing service test could be an automatic switching of the utilized service within an action dispatcher, so that from then on it uses one which still works correctly.

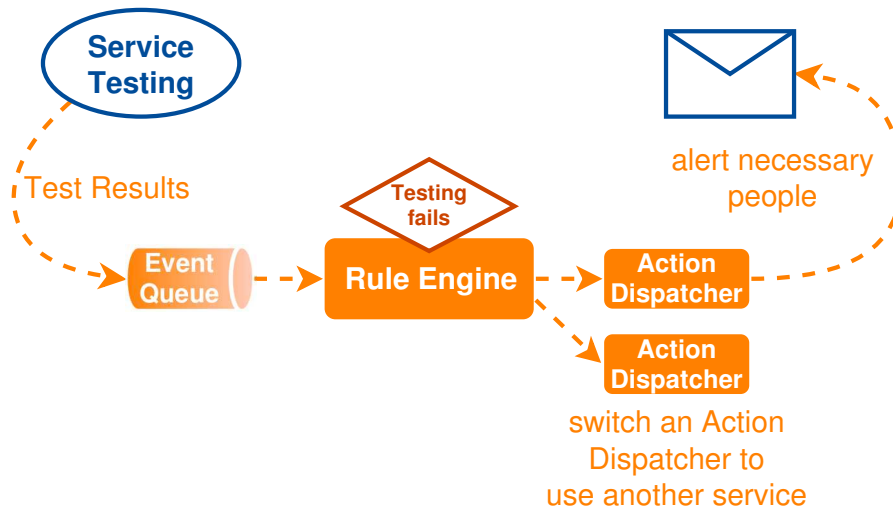


Figure 4.5: Test proper Service Functionality and Availability

## 4.4 Exploiting the Web of Things

A model for reactive information systems becomes also very interesting in the context of the Web of Things. Through the small connected devices, a lot of sensor data become accessible via the Web and can be used as events to trigger actions. These actions could also be part of the Web of things, if there are Things that offer services. One example of a reactive rule, that has parts in the Web of Things, is that of a server room which has a defective cooling. The increasing temperature eventually causes the servers to shutdown or even fail. Servers in this room could push current state information into a reactive system. The reactive system could take measurements if it detects a certain pattern that will lead to an overheating of all systems. It could inform certain (not so important) servers to gracefully shutdown and additionally inform administrators, who otherwise might miss the shutdown. Even better would be if it would have the power to kick in an additional emergency cooling system to prevent the shutdown of any of the servers.

Another scenario gets more realistic with the increasing number of homes that are connected to the Web. A home or apartment owner has his light controls attached to the Web. The first thing a reactive system could do, is that it detects holidays in the owner's agenda and automatically sets the light control to somewhat reasonable random during his absence. This would make suspicious characters, which are eventually interested in his wealth, think that he's still at home. In combination with another Thing that is connected to the Web and always accompanies people, the cell phone, an even more interesting application scenario can be thought of. The phone would push location information about the owner into the system. Whenever the owner gets close to his house, the reactive system could turn on the light in the entry area, pull up some nice music. On a Wednesday evening it could also inform the delivery service that they can deliver the owner's preferred dish now, because the owner is doing this always on a Wednesday evening.

## 4.5 Multi-Sourced Bad Weather Prediction

There are a lot of different weather services existing in the Web. One has to check several of them in order to get an idea on how likely it is that it will be raining on the journey to work and back. By composing a higher-order event from several weather update events, a user could store

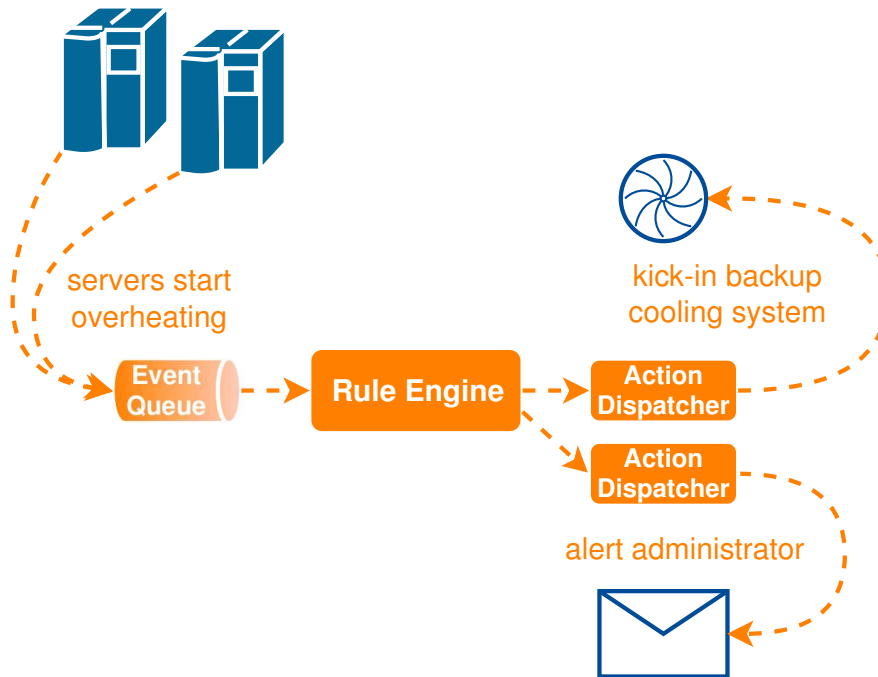


Figure 4.6: Measurements on Server Failure

a rule which alarms him early in the morning if more than 50% of the weather forecasts expect rain on the way to or back from work.



## Chapter 5

# Prototype System

We have so far introduced our conceptual model for reactive Information Systems and their Services and some example use cases to point out what would be possible with our model. In this chapter we present our proof of concept prototype system, which has a focus on the Web as its Information Spaces. We will then introduce our ECA rule language, which gives all the necessary power over our prototype system and which can be directly translated into the internal rules representation.

### 5.1 Architecture

The prototype system is the adoption of our conceptual model for reactive Information Systems and their services to the Web. The Web consists of many Information Systems and because of its Service-Oriented Architecture it can be seen as one large Information System, therefore we can impose reactivity on the Web. Since communication over services in the Web is often latency driven, we came to the conclusion that asynchronous communication and therefore scalability should be attributes our prototype system has to support natively. Another aspect to be regarded for the architectural decision was how the rules are going to be represented internally. We introduced XML and JSON as common ways to communicate data between services on the Web. Both formats represent data in a tree structure, and this is also what we decided to assume for the explicit parameters in the events that will enter our prototype. Together with the requirement of native support for an Event-Driven Architecture (EDA) our decision was to build upon the recent adoption of JavaScript to application development through Node.js<sup>1</sup> and its human-readable JSON communication format.

The prototype system consists of several modules, shown in Figure 5.1, which we are going to introduce within this section:

- **Poller:** Loads Event Trigger modules and forwards events coming from them to the Event Queue. Event Trigger modules poll for changes in the Web and transform them into events.
- **Webhook Listener:** Listens on active Webhooks for events and forwards them to the Event Queue.
- **Event Queue:** Buffers events for the case of an overly busy Rule Engine.

---

<sup>1</sup><http://nodejs.org/>

- **Rules Engine:** Picks an event from the Event Queue whenever there is one and it is idle.
- **User Request Handler:** The user interface modules to administrate Event Trigger, Webhooks, Rules and Action Dispatchers.

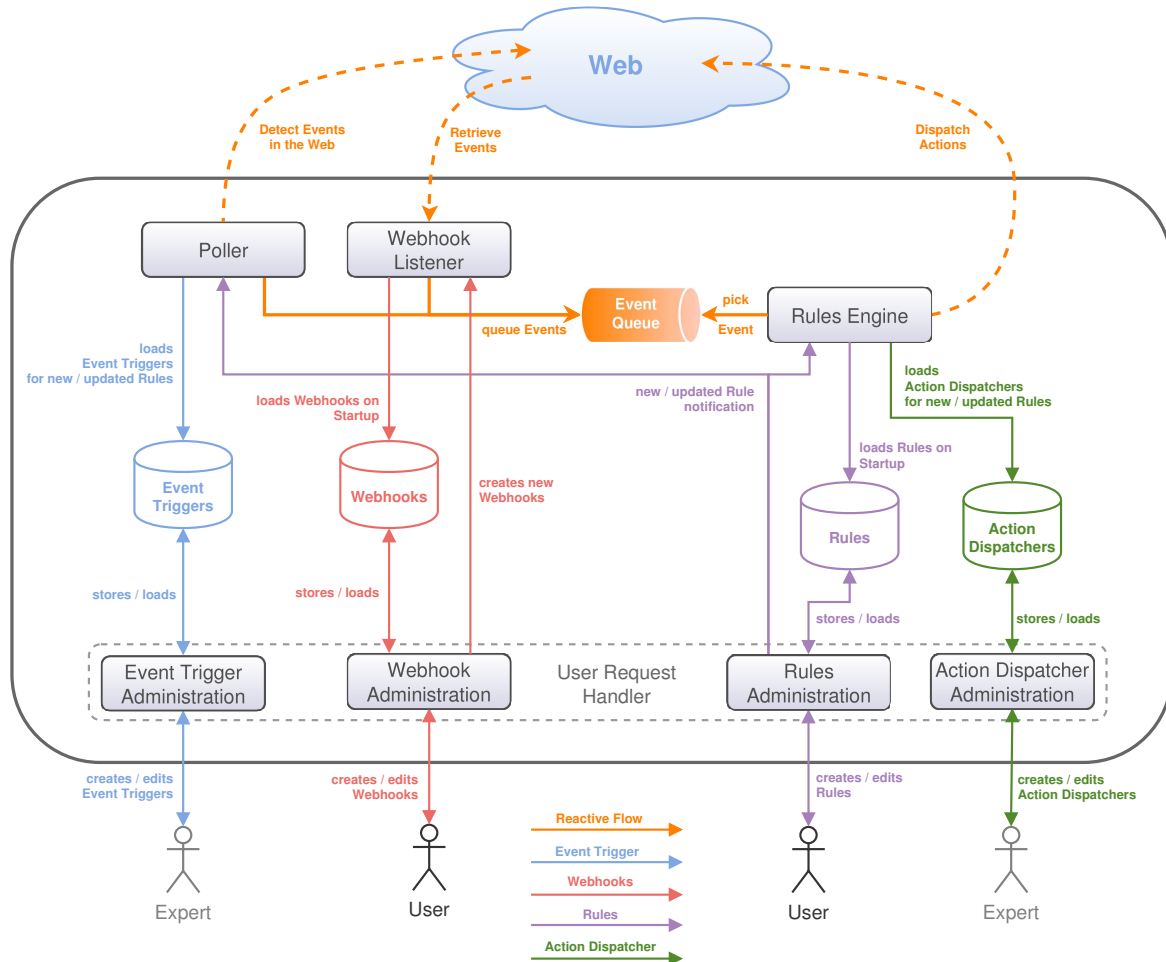


Figure 5.1: Prototype System Architecture

### 5.1.1 Data Structure for Event Parameters

Events in our prototype system are internally represented as tree structures in JSON format, which builds on only two data structures:

- **Objects:** Unordered collections of name/value pairs, which can also be implemented as a hash map, dictionary or struct in other languages.
- **Arrays:** Ordered lists of values, which can also be implemented as a record, vector or list in other languages.

A value can be an object or an array, but also a unicode string, a number, boolean or null. This allows for any arbitrary depth and chaining of the supported data representations. It is a handy feature when we assume a tree structure for events in our prototype system since the selection of nodes in tree structures has been well studied and useful libraries exist. Since all native data structures within JavaScript programming code are already in JSON format, it can easily be

marshalled into one string and communicated to other applications without overhead. JSON can be implemented in virtually every programming language, therefore received a lot of attention and is supported by many Web Resources for communication.

### 5.1.2 Dynamic Code Loading for Event Trigger and Action Dispatcher

During our research we have seen many different Web services with thoroughly different requirements in terms of communication. The easiest category among them were the Web APIs with their RESTful services. And still, in many cases it is only possible to access data which does not refer to an event. To detect a change in data over time we need to be able to store an earlier request and get the difference to the current request, which could then be transformed into a meaningful event. The derivation of a meaningful event from data can be a complex task which requires certain operations, which underlines the need for powerful Event Trigger modules. The complexity is even bigger for Action Dispatcher modules which alter data and require more complex communication to Web services. For those reasons we made the decision to keep these modules flexible in terms of communication and also to leave it open to expert users to encapsulate complicated logic into them for inexperienced users. Therefore we modeled the Event Trigger and Action Dispatcher to be JavaScript code modules that are created by expert users during runtime and loaded whenever a rule is activated, which needs them. These modules run in a sandbox and got only access to certain JavaScript libraries, which are provided by the owner of the system. Apart from those libraries there are two other important functions offered in both type of modules:

- `log`: Will store log entries on a per rule base, wherever the instruction is met during execution of a module. The log can be seen by the user who created the rule, which includes that module.
- `pushEvent`: This is the most important part for Event Trigger modules, since events are only through this function pushed into the prototype system. For Action Dispatcher modules this gives the sometimes useful option for loopback events.

### 5.1.3 Retrieving Events

In the last chapters, we always put emphasis on the two different ways how events can be retrieved from Information Systems, i.e. actively pulling events, or passively retrieving them. Even though some Information Systems offer the access to services that offer data that corresponds to events and can instantly be forwarded into the system, we have seen that there is need for the derivation of events from changes in data on the Web, therefore we need the Event Trigger modules. But still, our vision is that of an optimal real-time reactive Web which means that all possible events are offered by all Information Systems. An interested remote entity could then just announce interest in a certain kind of events over a Webhook and retrieves them in real-time. For that reason we laid out our architecture for such Webhooks, but still offer the Event Trigger modules to poll for events in the semi-static World Wide Web.

#### Polling for Event Triggers

As we have pointed out before, Event Trigger modules are JavaScript code modules with access to a set of predefined libraries, which offer the flexibility of JavaScript programming. The Poller loads those Event Trigger modules whenever they are required as part of a rule. The user of

the Event Trigger can choose a starting point and an interval for the polling to take place. An example Web service which offers polling for events is the Email Yak<sup>2</sup> Web API, which responds with new emails when requested. The code required to request the new mails from this service and forward them into the prototype system is quite short and shown in Listing 5.1, but for other examples it can get more complex. For better readability the code is written in CoffeeScript<sup>3</sup>. Only expert users are expected to store such a piece of code in our prototype, which enables inexperienced users to simply choose the "EmailYak -> newMail" Event Trigger for their rule. A great opportunity to access data from webpages via a Web API is Import.io<sup>4</sup>. By browsing through the Web with their browser it is possible to select certain parts from a webpage and store them in a mask. Data is instantly extracted from the webpage, using the stored mask, when sending a request to their Web API with the given mask id and the URI. This is a great tool for expert users to predefine desired data on webpages and then produce events whenever there is a change in that data.

```

1 url = "https://api.emailyak.com/v1/#{params.apikey}/json/get/new/email/"
2 exports.newMail = () ->
3   needle.get url, ( err, resp, body ) ->
4     if not err and resp.statusCode is 200
5       pushEvent mail for mail in body.Emails

```

Listing 5.1: Event Trigger code to poll Email Yak RESTful Web service for new Mails; written in CoffeeScript

## Webhooks

As powerful as Webhooks are in bringing us real-time notifications, as simple they are to use. In our prototype users can easily create as many new Webhooks as they choose to. The only requirement is, that they are associated to an event name by the user. A new Webhook will be created in the Webhook Listener, which from then on accepts events posted to it, and the Webhook URI is displayed to the user. Any Web Resource that supports the concept of Webhooks (e.g. GitHub<sup>5</sup>) has a place to register the Webhook URI. Whenever a remote Web Resource pushes an event to the Webhook, the user-defined event name is assigned as the implicit parameter of a freshly created internal event, while the whole incoming event body is added as explicit parameters to it. Afterwards it is forwarded to the Event Queue.

### 5.1.4 Action Dispatcher

### 5.1.5 ECA Rules in the Rule Engine

A special case is the Rules Administration because it notifies the Poller and Rules Engine about new or updated Rules, which then in turn load required Event Trigger or Action Dispatcher modules.

The engine checks the event against its stored ECA rules and fires the actions whenever a rule applies to an event. When a new rule is stored, the engine instantiates the action modules

<sup>2</sup><http://www.emailyak.com/>

<sup>3</sup><http://coffeescript.org/>

<sup>4</sup><https://import.io/>

<sup>5</sup><https://github.com/>

that are required in case an event arrives at the system which triggers the actions.

## 5.2 Rule Language for Prototype System

We have also seen that virtual events have an implicit parameter which is the event name, thus this is the one that is referred to in the event part of a rule. If an event name, of an event entering the Rule Engine, is detected in a predefined rule, the engine compares the event against the conditions of that rule and if all conditions are met, dispatches the actions defined within the rule. We also assume that Action Dispatcher modules can be accessed through common function invocation syntax ( i.e. `actionFunction(param1, param2[, ...])` ) in order to dispatch an action to an Information System. Through this we are able to define a rule language that uses tree node selectors in order to select explicit event parameters, which can then be verified against conditions and forwarded to Action Dispatcher modules as function parameters. Listing 5.2 shows an example phrase of our envisioned rule language where the retrieval of a new mail will be checked for soccer news and, if confirmed, the mail body will be forwarded to an interested person.

```

1 ON mailevent
2 IF #{ .subject } instr "Soccer News"
3 DO mail->send( "soccerfan@host.com", "News about soccer!", #{ .body } )

```

Listing 5.2: Example Phrase in Reactive Rule Language

## 5.3 Example Use Cases

During our research we found a troublesome server room that shows how the Web of Things can be exposed through our model. This server room suffered from a defective cooling system which lead to a drastical increase of temperature in certain circumstances. As a consequence certain server automatically shut themselves down as safety measurements. Eventually, these shutdowns weren't detected immediately by the people that administered these servers, therefore unnecessary downtimes were the result. As a very quick fix to inform certain administrators about the shutdown of their server, we started pinging these servers and pushed the results int

## 5.4 Web Programming

### 5.4.1 Node.js

### 5.4.2 Callback Functions & Asynchronous Closures

Often, optimization approaches and programming language concepts require special attention to avoid common pitfalls. When closures are used as asynchronous functions, developers need to be very careful not to end up with race conditions.

Looking at an example of sequential code execution in Figure 5.2, we see that function execution of `fA` is halted until function `fB` is finished. If `fB` happens to be a latency-driven I/O operation the completion of `fA` could be deferred for a relatively long time. While the application

waits for the completion of the I/O operation, some remaining operations in  $fA$  could eventually already be executed without causing any race conditions.

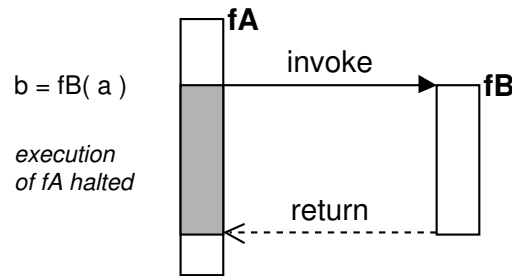


Figure 5.2: Synchronous Function Call

Asynchronous code execution, as shown in Figure 5.3, allows non-blocking and thus scalable applications. Non-blocking operations are a remedy for optimized resource allocation and open up ways to overcome previously described unnecessary resource bindings. Processing any kind of latency-driven I/O operation asynchronously ( e.g. filesystem access and socket communication ) exploits resources that would otherwise be bound while waiting for completion. Such operations are processed and completed whenever required resources are available.

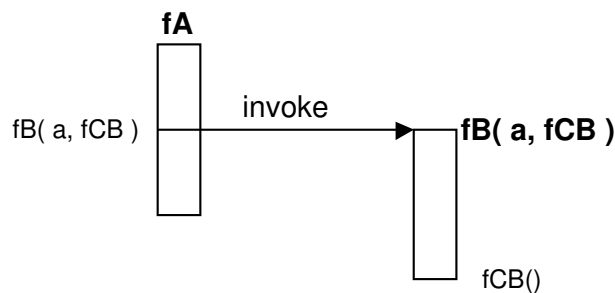


Figure 5.3: Asynchronous Function Call

Often other operations depend on the completion of asynchronous operations, hence their execution needs to be deferred. This necessary code execution deferral is achieved through the use of callback functions, denoted  $fCB$  in Figure 5.3. Any code placed in a callback function, which is assigned to an asynchronous operation, is only executed after the respective asynchronous operation completed. This allows stacking of functions and operations upon each other which automatically results in a flexible and event-driven application.

So far we didn't regard the context for such asynchronous functions. If a function has access to the enclosing context where it was invoked in, it is called a closure. Closures play an important role in ECMAScript[17], which is the base for widely-spread script languages like JavaScript, JScript and ActionScript. Closures in ECMAScript[17] are defined such as they have access to the context of the function they were created in. This is shown in Figure 5.4 where  $c$  from  $fA$ 's context is accessible from within  $fB$ , assuming that  $fB$  was created in  $fA$  and not only invoked from there. Closures make it necessary for the context of the outer function to survive past its execution so no references are broken. This is labeled "extended context lifetime" in Figure 5.4. Using asynchronous closures it becomes evident, that the context in the invoking function can change while the closure is still computing and eventually referencing the outer context, thus causing race conditions. This will be most obvious in a loop that immediately invokes  $fB$  several times, as shown in Figure 5.5. In such a setup  $c$  will have different values in the same part of different invocations of  $fB$ .

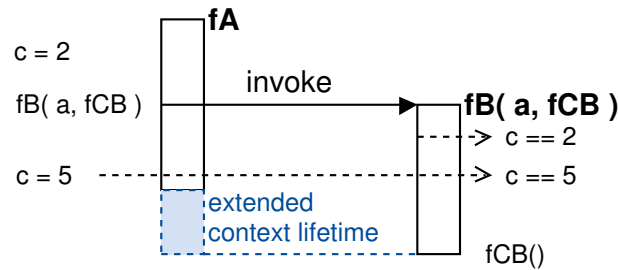


Figure 5.4: Closure Scope and referenced context

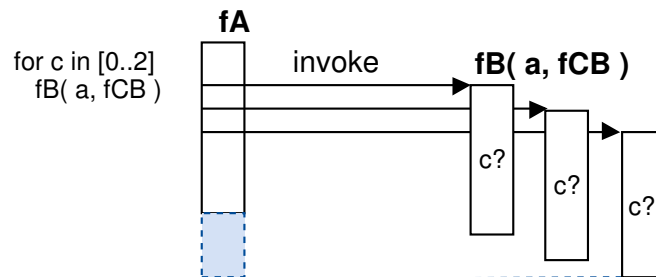


Figure 5.5: Closure context changes in a loop

Those event-driven context overwrites can be taken care of by shielding the closure from context changes, as shown in Figure 5.6. To shield the closure from context changes, closure `fB` needs to create another closure `fC` and return it to `fA`. The argument passed to `fB` is the context (`c` in Figure 5.6) that might change but requires to be persistent for one invocation. `fC` has now `c` as a fixed context, which can't be overwritten anymore. Now the only thing left is `fC` needs to be invoked and it will retain the original context. This implementation is necessary when the closure acts as a callback function for asynchronous operations, to preserve the original context in case it is required within the callback function.

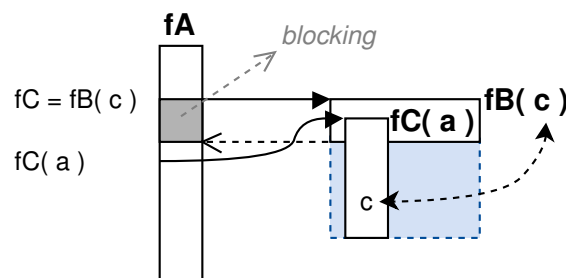


Figure 5.6: Closure Context Shielding

An example of how closure contexts can be shielded is shown in the Listing 5.3.

```
1 var fB = function( c ) { // Declare a function...
2   var fC = function( a ) { // ( <-- function to return )
3     console.log( c );
4   };
5   return fC;
6 };
7 for( var c = 0; c < 100; c++ ) {
8   // ... before you assign it to an event happening in the future:
9   var fC = fB( c );
10  setTimeout( fC, 3000 ); // will be executed after the loop ended
11 }
```

Listing 5.3: JavaScript Context Shielding



## **Chapter 6**

# **Conclusions & Future Work**

# Bibliography

- [1] Raman Adaikkalavan and Sharma Chakravarthy. Event Specification and Processing for Advanced Applications: Generalization and Formalization. In Roland Wagner, Norman Revell, and Günther Pernul, editors, *Database and Expert Systems Applications*, volume 4653 of *Lecture Notes in Computer Science*, pages 369–379. Springer Berlin Heidelberg, 2007.
- [2] Mert Akdere, Uğur Çetintemel, and Nesime Tatbul, Plan-based complex event detection across distributed sources. *Proceedings of the VLDB Endowment*, 1(1):66–77, 2008.
- [3] JoséJúlio Alferes and Ricardo Amador. r 3– A Foundational Ontology for Reactive Rules. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, volume 4803 of *Lecture Notes in Computer Science*, pages 933–952. Springer Berlin Heidelberg, 2007.
- [4] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. A Rule-Based Language for Complex Event Processing and Reasoning. In Thomas Lukasiewicz Pascal Hitzler, editor, *Web Reasoning and Rule Systems - Fourth International Conference*, volume 6333 of *LNCS*, pages 42–57. Springer, September 2010.
- [5] Alistair P. Barros and Marlon Dumas, The Rise of Web Service Ecosystems. *IT Professional*, 8(5):31–37, 2006.
- [6] Tim Berners-Lee, Robert Cailliau, Jean-François Groff, and Bernd Pollermann, World-Wide Web: The Information Universe. *Electronic Networking: Research, Applications and Policy*, 1(2):74–82, 1992.
- [7] Tim Berners-Lee, James Hendler, Ora Lassila, et al., The semantic web. *Scientific american*, 284(5):28–37, 2001.
- [8] Andrew D. Birrell and Bruce Jay Nelson, Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984.
- [9] Harold Boley. The RuleML family of web rule languages. In *Principles and Practice of Semantic Web Reasoning*, pages 1–17. Springer, 2006.
- [10] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (SOAP) 1.1, 2000.
- [11] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau, Extensible markup language (XML). *World Wide Web Consortium Recommendation REC-xml-19980210*. <http://www.w3.org/TR/1998/REC-xml-19980210>, 1998.
- [12] Francois Bry and Paula Iavinia Patranjan, Reactivity on the Web: Paradigms and Applications of the Language XChange. *J. of Web Engineering*, 5:2006, 2005.

- 
- [13] François Bry and Michael Eckert. Twelve Theses on Reactive Rules for the Web. In Torsten Grust, Hagen Höpfner, Arantza Illarramendi, Stefan Jablonski, Marco Mesiti, Sascha Müller, Paula-Lavinia Patranjan, Kai-Uwe Sattler, Myra Spiliopoulou, and Jef Wijsen, editors, *Current Trends in Database Technology – EDBT 2006*, volume 4254 of *Lecture Notes in Computer Science*, pages 842–854. Springer Berlin Heidelberg, 2006.
  - [14] Cinzia Cappiello, Maristella Matera, Matteo Picozzi, Gabriele Sprega, Donato Barbagallo, and Chiara Francalanci. DashMash: A Mashup Environment for End User Development. In Sören Auer, Oscar Díaz, and GeorgeA. Papadopoulos, editors, *Web Engineering*, volume 6757 of *Lecture Notes in Computer Science*, pages 152–166. Springer Berlin Heidelberg, 2011.
  - [15] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web services description language (WSDL) 1.1, 2001.
  - [16] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle. Consistency and scalability in event notification for embedded Web applications. In *Web Systems Evolution (WSE), 2009 11th IEEE International Symposium on*, pages 89–98, Sept 2009.
  - [17] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1 edition, June 2011.
  - [18] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec, The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.
  - [19] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.
  - [20] N. H. Gehani and H. V. Jagadish. Composite event specification in active databases: Model and implementation. pages 327–338, 1992.
  - [21] Adrian Giurca and Emilian Pascalau, Json rules. *Proceedings of the Proceedings of 4th Knowledge Engineering and Software Engineering, KESE*, 425:7–18, 2008.
  - [22] Object Management Group. *The Common Object Request Broker (CORBA): Architecture and Specification*. Object Management Group, 1995.
  - [23] D. Guinard, V. Trifa, and E. Wilde. A resource oriented architecture for the Web of Things. In *Internet of Things (IOT), 2010*, pages 1–8, Nov 2010.
  - [24] Keman Huang, Yushun Fan, and Wei Tan. An Empirical Study of Programmable Web: A Network Analysis on a Service-Mashup System. In Carole A. Goble, Peter P. Chen, and Jia Zhang, editors, *ICWS*, pages 552–559. IEEE, 2012.
  - [25] Xuanzhe Liu, Yi Hui, Wei Sun, and Haiqi Liang. Towards Service Composition Based on Mashup. In *Services, 2007 IEEE Congress on*, pages 332–339, July 2007.
  - [26] Larry Masinter, Tim Berners-Lee, and Roy T Fielding, Uniform resource identifier (URI): Generic syntax. 2005.
  - [27] Gregory B Newby. Metric multidimensional information space. In *TREC*. Citeseer, 1996.
  - [28] George Papamarkos, Alexandra Poulouvassilis, and Peter T Wood. RDDL: An event-condition-action language for RDF. In *Proc. of the 3rd International Workshop on Web Dynamics*, 2004.
  - [29] Adrian Paschke, Harold Boley, Zhili Zhao, Kia Teymourian, and Tara Athan. Reaction RuleML 1.0: Standardized Semantic Reaction Rules. In Antonis Bikakis and Adrian Giurca,

- editors, *Rules on the Web: Research and Applications*, volume 7438 of *Lecture Notes in Computer Science*, pages 100–119. Springer Berlin Heidelberg, 2012.
- [30] Paula-lavinia Patranjan. *The Language XChange*. PhD thesis, Ludwig-Maximilians-Universität München, 2005.
- [31] C. Peltz, Web services orchestration and choreography. *Computer*, 36(10):46–52, Oct 2003.
- [32] Randall Perrey and Mark Lycett. Service-oriented architecture. In *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*, pages 116–119. IEEE, 2003.
- [33] P.R. Pietzuch, B. Shand, and J. Bacon, Composite event detection as a generic middleware extension. *Network, IEEE*, 18(1):44–55, Jan 2004.
- [34] Paul Rademacher. HousingMaps. <http://www.housingmaps.com/>, 2005. Accessed: 2014-6-6.
- [35] REVERSE - Reasoning on the Web with Rules and Semantics. <http://reverse.net>. Accessed: 2014-05-08.
- [36] Leonard Richardson and Sam Ruby. *RESTful web services*. O'Reilly Media, Inc., 2008.
- [37] Sven Rizzotti and Helmar Burkhart. useKit: A Step Towards the Executable Web 3.0. In *Proceedings of the 19th International Conference on World Wide Web, WWW '10*, pages 1175–1176, New York, NY, USA, 2010. ACM.
- [38] D Robins. Complex event processing. In *Second International Workshop on Education Technology and Computer Science. Wuhan*, 2010.
- [39] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of GridRPC: A remote procedure call API for Grid computing. In *Grid Computing—GRID 2002*, pages 274–278. Springer, 2002.
- [40] Kathryn T. Stolee, Sebastian Elbaum, and Anita Sarma, Discovering how end-user programmers and their communities use public repositories: A study on Yahoo! Pipes. *Information and Software Technology*, 55(7):1289 – 1303, 2013.
- [41] P. Windley. *The Live Web: Building Event-Based Connections in the Cloud*. Cengage Learning PTR, 2011.
- [42] Detlef Zimmer and Rainer Unland. On The Semantics Of Complex Events In Active Database Management Systems. pages 392–399. IEEE Computer Society Press, 1999.

# List of Figures

1.1	Users orchestrate the Web's Data and Functionality . . . . .	2
2.1	Number of registered APIs in the ProgrammableWeb directory by date . . . . .	4
3.1	Reactivity imposed on Information Systems and their Information Space over Services	12
3.2	Conceptual Model for Reactive Information Systems and their Services . . . . .	13
3.3	Conceptual Model for Reactive Information Systems and their Services . . . . .	15
4.1	Enrich CMS Post with Remote Knowledge Data . . . . .	17
4.2	Create course resources at semester start . . . . .	17
4.3	Create course resource for registered student . . . . .	18
4.4	Notify student before exercise due date . . . . .	18
4.5	Test proper Service Functionality and Availability . . . . .	19
4.6	Measurements on Server Failure . . . . .	20
5.1	Prototype System Architecture . . . . .	22
5.2	Synchronous Function Call . . . . .	26
5.3	Asynchronous Function Call . . . . .	26
5.4	Closure Scope and referenced context . . . . .	27
5.5	Closure context changes in a loop . . . . .	27
5.6	Closure Context Shielding . . . . .	27

# List of Tables

2.1	Key Properties of existing Rule Languages with respect to our Conceptual Model	10
-----	--	----

# Listings

5.1	Event Trigger code to poll Email Yak RESTful Web service for new Mails; written in CoffeeScript . . . . .	24
5.2	Example Phrase in Reactive Rule Language . . . . .	25
5.3	JavaScript Context Shielding . . . . .	28

# **Appendices**



# Appendix A

## Rule Languages

### A.1 Example JSON Event for Rule Languages

```

1 {
2   "eventname": "email",
3   "body": {
4     "sender": "sender@mail.com",
5     "subject": "Important subject!",
6     "textbody": "Hi User,\n\nThis is a lengthy mail body"
7   }
8 }

```

### A.2 E-Mail Example Rule expressed in RDFTL

```

1 ON INSERT document("inbound_queue.xml")/mails/mail
2 IF $delta/sender[.="sender@mail.com"]
3 DO DELETE document("inbound_queue.xml")/mails/mail;
4 LET $api = resource("www.webapi.com") IN
5 INSERT ($api, newcontent,
6   <content>New mail: {$delta/subject}</content>)

```

### A.3 E-Mail Example Rule expressed in XChange/Xcerpt

### A.4 E-Mail Example Rule expressed in Notation 3

### A.5 E-Mail Example Rule expressed in JSON Rules

```

1 {
2   "id": 0,
3   "conditions": [
4     {
5       "type": "email",
6       "constraints": [

```

```

1 TRANSACTION
2   in {
3     resource { "http://www.webapi.com"},
4     newcontents {{
5       insert newcontent { var Mail }
6     }}
7   }
8   ON
9     xchange:event {{
10      xchange:sender { "http://mailserver.com" },
11      var Mail -> email {{
12        sender { "sender@mail.com" }
13      }}
14    }}
15  END

```

```

1 { ?x :event "email". ?x :sender "sender@mail.com" }
2 => { :webapi :newcontent ?x }

```

```

7     {
8       "propertyName": "sender",
9       "operator": "EQ",
10      "restriction": {
11        "type": "String",
12        "value": "sender@mail.com"
13      }
14    },
15    {
16      "bind": "$S",
17      "propertyName": "subject"
18    }
19  ]
20 }
21 ],
22 "actions": [
23   "webapi('addcontent', $S)"
24 ]
25 }

```

## A.6 E-Mail Example Rule expressed in Kinetics Rule Language (KRL)

## A.7 E-Mail Example Rule expressed in (Reaction) RuleML

```

1 <Rule style="active">
2   <on>
3     <Event>
4       <Atom>
5         <Rel per="value">mail</Rel>
6         <Var>sender</Var>
7         <Var>subject</Var>
8       </Atom>

```

```

1  rule store_mail {
2      select when mail newmail
3      sender re#sender@mail.com#
4      subject re### setting(subj)
5      http:post("http://www.webapi.com/newcontent")
6      with params = {
7          "text": subj
8      }
9  }

```

Listing A.1: E-Mail Example rule in KRL

```

9      </Event>
10     </on>
11     <if>
12         <Atom>
13             <op><Rel>equals</Rel></op>
14             <Var>sender</Var>
15             <Ind>sender@mail.com</Ind>
16         </Atom>
17     </if>
18     <do>
19         <Atom>
20             <oid><Ind uri="http://webapi.com"/></oid>
21             <Rel>newcontent</Rel>
22             <Var>subject</Var>
23         </Atom>
24     </do>
25 </Rule>

```

## A.8 Prototype Rule transformed into JSON

```

1  {
2      "event": "mail",
3      "conditions": [
4          { "sender": "sender@mail.com" },
5      ],
6      "actions": [
7          {
8              "api": "webapi",
9              "method": "newcontent",
10             "arguments": {
11                 "text": "$X.subject"
12             }
13         }
14     ]
15 }

```

## Appendix B

# Rules

### B.1 Binder Annotations

#### B.1.1 Binder Annotations

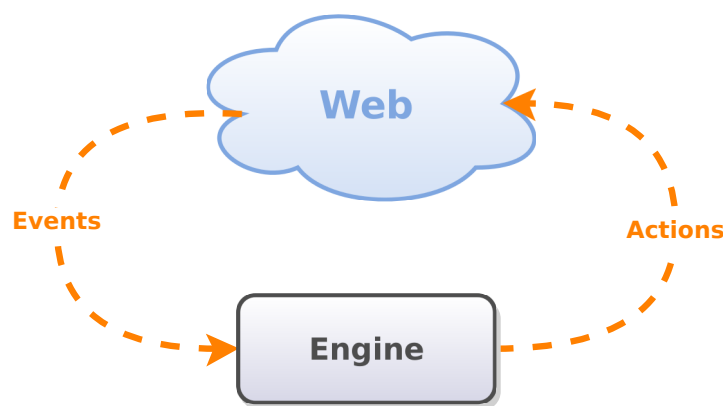


Figure B.1: SHOULD NOT SHOW UP

# Appendix C

## Benchmarking

### C.1 Java

```

1  /*
2   * BenchmarkingDeferred.java
3   */
4  import java.util.concurrent.ScheduledExecutorService;
5  import java.util.concurrent.Executors;
6  import java.util.concurrent.TimeUnit;
7  import java.util.HashMap;
8
9  public class BenchmarkingDeferred {
10
11     private static Runtime runtime = Runtime.getRuntime();
12     private static final ScheduledExecutorService worker =
13         Executors.newSingleThreadScheduledExecutor();
14
15     private static void deferFunctionCall( int numScopeVars, int delay, String scopeId ) {
16         HashMap<String, String> mapVars = new HashMap<String, String>();
17         for( int i = 0; i < numScopeVars; i++ ) {
18             mapVars.put( "id" + i, "12345678" ); // 8 bytes per stored scope variable
19         }
20         Object context = new TimeoutContext( "TimeoutFunction" );
21         Runnable task = new RunnableCallbackFunction( mapVars, context );
22         worker.schedule( task, delay, TimeUnit.SECONDS );
23     }
24
25     public static void main( String[] args ) {
26         long startTime, stopTime;
27         int numVars = 10, firstArg = 0;
28         firstArg = Integer.parseInt( args[0] );
29         numVars = Integer.parseInt( args[1] );
30         int j = 0, numFuncs = 1 << firstArg;
31
32         startTime = System.nanoTime();
33         while( j++ < numFuncs ) {
34             deferFunctionCall( numVars, numFuncs * 10, numFuncs + "(" + j + ")" );
35         }
36         stopTime = System.nanoTime();
37
38         // [...] benchmark system out
39
40         worker.shutdownNow();

```

```

41     }
42 }
43
44 /*
45  * RunnableCallbackFunction.java
46  */
47 import java.util.HashMap;
48
49 /*
50  * The Callback function instance.
51  */
52 public class RunnableCallbackFunction implements Runnable {
53
54     // The hashhmap is used to store variables and their value as the scope
55     private HashMap<String, String> mapScope;
56     private Object context;
57
58     public RunnableCallbackFunction( HashMap<String, String> scope, Object context ) {
59         this.mapScope = scope;
60         this.context = context;
61     }
62
63     // If this is executing, we didn't wait long enough and the
64     // benchmark time is compromised
65     public void run() {
66         System.out.println( mapScope.toString() );
67     }
68 }
69
70 /*
71  * TimeoutContext.java
72  */
73
74 public class TimeoutContext {
75     private long idleTimeout = 1;
76     private long idlePrev;
77     private long idleNext;
78     private long idleStart = 140000505;
79     private String onTimeout = null;
80     private boolean repeat = false;
81
82     public TimeoutContext( String cb ) {
83         this.onTimeout = cb;
84     }
85 }

```

Listing C.1: Closure Benchmarking: Java Code

## C.2 JavaScript

```

1  /*
2  The function deferral measurements in node.js
3  */
4
5  var deferredFunction = function ( numScopeVars, delay, scopeId ) {
6      var scope = {};
7      for ( var i = 0; i < numScopeVars; i++ ) {
8          scope[ "id" + i ] = "12345678"; // 8 bytes per stored scope variable
9      }
10     setTimeout( function () {
11         // If this is executed we didn't wait long enough
12         console.log( JSON.stringify( scope, null, ' ' ) );
13     }, delay );
14 }
15
16 var numOfFunctions,
17     numOfScopeVars = process.argv[ 3 ];
18
19 numOfFunctions = Math.pow( 2, process.argv[ 2 ] );
20
21 var time = process.hrtime();
22 for (var i = 0; i < numOfFunctions; i++) {
23     deferredFunction( numOfScopeVars, 1000 * numOfFunctions, numOfFunctions + "(" + i + " " );
24 };
25 var diff = process.hrtime( time );
26
27 var mem = process.memoryUsage();
28 // [...] benchmark system out
29 process.exit( 0 );

```

Listing C.2: Closure Benchmarking: JavaScript Code

# Glossary

**Information Space** *"[...] is a set of concepts and relations among them held by an Information System. Information Space is produced by a set of known procedures, and is changed through intentional manipulation of its content"*[27]. i, 12–16, 21

**Information System** is a network of software and hardware components that support collection, filtering, storing, processing and distribution of data. i, 12–16, 21, 23, 25

**Mashup** Web Applications that weave two or more different Web APIs together to provide a new perspective on data. 4, 6

**Semantic Web** Tim Berners-Lee's vision of the machine-readable Web through standard data formats and semantic metadata descriptions on top of the resources via RDF which turn the Web into a structured data collection. 1, 5, 8

**Web API** An application Programming Interface to either a Web service or the browser, meant for application to application communication. 4–7, 23, 24

**Web of Things** An evolution of the Internet of Things, which describes the integration of smart things (e.g. sensors, embedded devices or digitally enhanced objects) into the Internet. Web of Things is the adoption of the REST architectural style to the smart things in order to enable uniform access to these loosely coupled entities. 1, 13, 15

**Web Resource** Anything in the Web which can be identified, addressed and handled. Identification and addressing is often done over URIs. Resources and their semantic properties are described using RDF in the Semantic Web. 1–3, 5, 6, 8, 10–12, 14, 23, 24

**Web Service** A collection of SOAP related Web service (note the lower-case word service) standards which are widely adopted and developed in the industry. Also called "WS-\*" Web Services or the Big Web Services. 3, 4

**Web service** An interface for communication between applications over a network. They can provide access to and control over Web Resources. Web services are also called services on the Web or just services. 3, 5, 6, 13, 23, 24

**Webhook** A server-side Web API which accepts a URI that is used as a callback to drop events to another non-familiar server. 6, 7, 14

**World Wide Web** Tim Berners-Lee's vision of interlinked hypertext documents which are accessed over the Internet via browser and allow the navigation through a global information universe. iii, 3, 4, 14, 16, 23



# Acronyms

- CED** Complex Event Detection. 11, 14
- CEP** Complex Event Processing. 8, 11, 14
- CMS** Content Management System. 16, 17, 33
- CORBA** Common Object Request Broker Architecture. 4, 5
- DOM** Document Object Model. 8, 10
- ECA** Event-Condition-Action. iii, 3, 7–9, 11, 14, 15, 21
- EDA** Event-Driven Architecture. 7, 11, 14, 21
- ESB** Enterprise Service Bus. 9
- IaaS** Infrastructure as a service. 3
- IDL** Interface Definition Language. 5
- IIOP** Internet Inter-ORB Protocol. 5
- JSON** JavaScript Object Notification. 7, 8, 10, 21–23
- KR** Knowledge Representation. 8
- KRE** Kinetic Rules Engine. 9, 10
- KRL** Kinetics Rule Language. 9, 10
- ORB** Object Request Broker. 4, 5
- PaaS** Platform as a service. 3
- RDF** Resource Description Framework. 8, 10
- RDFTL** RDF Triggering Language. 8, 10
- REST** Representational State Transfer. 3, 5, 7, 23
- REVERSE** Reasoning on the Web with Rules and Semantics. 8
- RPC** Remote Procedure Call. 4, 5
- RuleML** Rule Markup Language. 8–10

**SaaS** Software as a service. 3

**SOA** Service-Oriented Architecture. 3, 21

**SOAP** Simple Object Access Protocol. 3–5

**URI** Uniform Resource Identifiers. 5–7, 24

**WSDL** Web Service Description Language. 4, 5

**XML** Extensible Markup Language. 4, 7, 8, 21

**XML-RPC** XML - Remote Procedure Call. 4

**Declaration on Scientific Integrity**

(including a Declaration on Plagiarism and Fraud)

Bachelor's / Master's Thesis (*Please cross out what does not apply*)

Title of Thesis (*Please print in capital letters*):

---

---

---

First Name, Surname (*Please print in capital letters*):

Matriculation No.: \_\_\_\_\_

I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged.

I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

In addition to this declaration, I am submitting a separate agreement regarding the publication of or public access to this work.

☐ Yes      ☐ No

Place, Date: \_\_\_\_\_

Signature: \_\_\_\_\_

*Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis .*