Towards The Reactive Web

Master Thesis

Dominic Bosch
Departement Mathematics and Computer Science
University of Basel

May 6, 2014

Acknowledgment

This master thesis is dedicated to all my dearest

Contents

Chapter 1

Introduction

The fast evolving web has brought up a trend towards easy to master interfaces to services, the so called WebAPIs. They do not only provide access to mere services but whole applications that allow access over WebAPIs. These trending WebAPIs benefit from a RESTful architecture which predominantly uses HTTP and thus relies on the most basic and powerful operations and the basis of the Web itself, the HTTP protocol.

quick (handling/mastering) accessible services and even whole web applications through so called Web APIs. WebAPIs provide powerful tools to govern data and functionality in the web independently of any user interface from the service provider. The relatively Allowing access to these services via API is increasingly popular and allows to mash up these services

Practically all services flood the user with events The web should be event driven, that's why we need an engine that deals with events and makes the web reactive There's still the challenge of filtering What's important to whom Plus the user needs to have tools to combine and add programmability to the combination,(such as conditions, selection of provided arguments and so on)

1.1 WebAPI Mashups

Mashups combine information and functionality of more than one resource in a single place. The mashing up of such resources allows new points of view on data, or even ways to interact with them. Simple functions are combined into more powerful ones which influence data and services in a way their founders eventually didn't even think of. They have been developped ever since services in the web started to exist and were accessible in a more or less convenient way. One of the earliest inventors of such a webservice mashup is Paul Rademacher. In the same year after Google Maps came up in 2005, he invented a site[12, 3] that displayed Craigslist houses on a Google Map. With no Google Maps API at that time, he needed time and skills to reverse engineer Google Map's functionalities.

A large number of such "static" mashups were and are still developped. They are static in the way that they aggregate a fixed (and mostly low) number, of either data or functionality resources, to provide an enhanced resource in a specialized domain. Of course Mashups can be mashed up again, to provide even more sophisticated functionality and data. Some latest example Mashups, taken from the ProgrammableWeb[13] directory, are:

- Wifi and Plugs[5]: MapBox, Google Docs and Import.io API's used to display where Wi-Fi and plugs are available in London.
- MapLight[7]: GovTrack.us and OpenSecrets API's used to combine political results with financial contributions to show how capital contributions affect voting.

- Shared Count[14]: Facebook, LinkedIn, Pinterest and Twitter API's used to display informations about how well spread a URL is on social media sites.

In the past few years, research and development for platforms to allow users to flexibly mashup WebAPIs got attention. With IFTTT and Zapier, two platforms have evolved out of this process. Users that register on those platforms are provided with a multitude of WebAPI functions that act as event triggers and such that are used to execute actions. The user is then free to combine these event triggers and actions in the way it suits best, creating helpful WebAPI mashups on their own.

1.2 Rule Languages

TODO To allow user-defined WebAPI mashups, a rule language is required which is capable to express ECA Rules.

TODO ECA

Several different rule languages have been developped for different purposes and vary grately in terms of usability for ECA Rules together with WebAPI mashups.

[9] gave a good oveview over existing approaches. In this section we examine different existing rule languages with respect to a simple use case. We want the rule language react on the receipt of an email (event), check for a distinct email address (condition) and store it in a remote location, via a Web API (action). The email only contains the parts we require for this use case (the sender and a subject). A JSON representation of the email would be:

```
{
  "eventname": "email",
  "body": {
    "sender": "sender@mail.com",
    "subject": "Important subject!",
    "textbody": "Hi User,\n\nThis is a lengthy mail body"
  }
}
```

Listing 1.1: Example E-Mail event expressed in JSON

1.2.1 RDF & XML

An early ECA Rule Language for XML repositories[8] was postulated in 2003 and was picked up by many researches afterwards. It was designed to react on insert and delete events within XML repositories and as an action change XML documents.

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

Listing 1.2: E-Mail Example rule expressed in RDF

Now apart from implementing a rules engine, we would also need to add an XML document event manager which interpretes and pushes events into the XML file *inbound_queue.xml*.

Then again this instance would interprete the ouptuts of the ECA engine, which would theoretically manifest in other XML documents, and produce meaningful actions on remote hosts. This wouldn't be an architecture which has its focus on the solution of our use case and, as a result, add complexity and create an unnecessary overhead.

1.2.2 Notation 3

To make the lengthy RDF definitions smaller and more readable, Notation 3[1] was designed and announced in 2005. Through the implies operator(=>) an "event" can be connected to an "action", both expressed in RDF's subject, predicate, object notation, which makes the expression of ECA rules a complicated and not very intuitive task. A solution to our use case would look as follows:

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

Listing 1.3: E-Mail Example rule expressed in Notation 3

It's obvious that this language is used to express relations between entities and thus not really suitable for our use case, since we would require another interpreter to infer the actions. But concepts and ideas of the work that was done in these consortias could eventually still find influence into our solution.

1.2.3 XChange/Xcerpt

The rule language XChange[11] was the outcome of the REWERSE project and acted as an influence in many further researches. The language was designed to add reactive behaviour to a "static" web which is represented through XML resources. Thus we have action logics to alter such resources through insertions and deletions. Since we aim to utilize web API's for our rule language we need a more generic approach which adds flexibility in term of the API provided. But the thorough research done with the language XChange holds valuable concepts, especially in terms of temporal evet composition. This could be a rule according to our use case:

```
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
```

Listing 1.4: E-Mail Example rule expressed in XChange

But XChange is designed to access other resources in an action and thus provides powerful tools:

```
TRANSACTION
  [...]
ON
  [...]
FROM
  in {
    resource { "http://www.weather.com"},
    temperatures {{
      var T -> temperature {{
        datetime { "2013-10-20-08:00:00AM" }
      }}
    }}
  }
END
```

Listing 1.5: XChange Rule accesses remote resource

1.2.4 JSON Rules

In 2008 *JSON Rules* [6] was introduced as a language to easily react on specific DOM tree compositions. The usage of JavaScript allowed them to provide simple functions which could be called directly by the actions, thus abstracting functionality from the language. This key concept found influence into our language as it allows different layers of abstractions. Through this it is possible to provide generic functions for expert user as well as very limited functions with only few possibilities for parameterization to be used by unexperienced persons. A drawback of this language is its binding to DOM tree events, where we would want to react on any events happening in the world. Also the temporal composition to complex events is not a subject of their work and needs further attention.

```
{
    "id": 0,
    "conditions": [
        {
            "type": "email",
            "constraints": [
                {
                    "propertyName": "sender",
                    "operator": "EQ",
                    "restriction": {
                        "type": "String",
                        "value": "sender@mail.com"
                    }
                },
                {
                    "bind": "$S",
                    "propertyName": "subject"
                }
            ]
        }
    ],
    "actions": [
        "webapi('addcontent', $S)"
    ]
}
```

Listing 1.6: E-Mail Example rule in JSON Rules

1.2.5 Kinetics Rule Language (KRL)

A most recent (2011) open-source development is the Kinetic Rules Engine together with the Kinetics Rule Language [15]. It is built for the purpose of adding reactivity to the cloud. The language is based on declarative syntax, enriched with imparative elements. But it is a tedious task to get into a whole new language and their caveats. *authorization?*

```
rule store_mail {
  select when mail newmail
  sender re#sender@mail.com#
  subject re#*# setting(subj)
  http:post("http://www.webapi.com/newcontent")
  with params = {
    "text": subj
  }
}
```

Listing 1.7: E-Mail Example rule in KRL

1.2.6 (Reaction) RuleML

The basis of *RuleML* [2] is datalog, a language in the intersection of SQL and Prolog. In 2012 the *Reaction RuleML* [10] language incorporated several different types of rules into the RuleML syntax, to establish a uniform syntax and interchangability of rules. *Reaction RuleML* is a valuable resource in terms of manifold research that has been done in the domain of rule languages, but the syntax is not user-friendly.

R2ML allows usage for RuleML together with many other dialects. Really!?

```
<Rule style="active">
  <on>
    <Event>
      <Atom>
        <Rel per="value">mail</Rel>
        <Var>sender</Var>
        <Var>subject</Var>
      </Atom>
    </Event>
  </on>
  <if>
    <Atom>
      <op><Rel>equals</Rel></op>
      <Var>sender</Var>
      <Ind>sender@mail.com</Ind>
    </Atom>
  </if>
  <do>
    <Atom>
      <oid><Ind uri="http://webapi.com"/></oid>
      <Rel>newcontent</Rel>
      <Var>subject</Var>
    </Atom>
  </do>
</Rule>
```

Most of the examined rule languages are designed for the interchangability of rules between different service providers. We do not attempt to jump into this domain but we rather

pick up important concepts to manifest web API's as first class citizens of our rule language. This allows the ad-hoc design and implementation of reactive rules between existing web API's without the need for their cooperation in setting up their endpoint in a special way.

Chapter 2

Conceptual Model for Reactive Web Systems

Existing ECA systems (List examples) all act on local data. Looking at (Wikipedia...) their definition is actions on local data. This does only add reactivity to these systems and not to the Web per se.

Such systems are merely event sinks which add fairly any value to the Web, except for the individual users and the system itself. We are taking a step further and allow not only the chaining up of several remote ECA engines, but also the invocation of actions on any arbitrary Web accessible service.

2.0.6.1 Own Rules

```
on mail
if sender="sender@mail.com"
do webapi->newcontent(subject)
```

Would be translated into:

```
{
  "event": "mail",
  "conditions": [
    { "sender": "sender@mail.com" },
  ],
  "actions": [
    {
      "api": "webapi",
      "method": "newcontent",
      "arguments": {
        "text": "$X.subject"
      }
    }
  ]
}
```

```
on weather->tempRaisesAbove(20)
do probinder->addContent(temp)
```

```
on emailyak->newMail
if FromAddress="dominic.bosch.db@gmail.com"
do probinder->newContent(TextBody)
```

```
on probinder->unreadContent
if serviceId=32
do probinder->markread(id),
   probinder->createContent(id, title, tab_name)
```

```
function call(args) {
  require('needle').post(
    'https://probinder.com/service/'
      + args.service + '/' + args.method,
    args.data,
    args.credentials
  );
};

function newContent(txt){
  call({
    service: '27',
    method: 'save',
    data: {
      companyId: '961',
      context: '17930',
      text: txt
    }
  });
}

on mail
do probinder->createContent(subject)

on mail
do probinder->call("27","save",
     ["961", "17930", subject]
   )

on probinder->unread
if serviceId=32
do probinder->setRead(id),
   probinder->makeFileEntry(service, id)
```

```
    "event": "emailyak->newMail",
    "condition": { "FromAddress": "dominic.bosch.db@gmail.com"},
    "actions": [
      {
        "module": "probinder->newContent",
        "arguments": {
          "content": "Received from EmailYak: $X.TextBody"
        }
      }
    ]

function newMail(callback) {
  needle.get('https://api.emailyak.com/v1/'+key+'/json/get/new/email/',
    function (error, response, body){
        var mails = JSON.parse(body).Emails;
        for(var i = 0; i < mails.length; i++) callback(mails[i]);
    }
  );
}

{

  "event": "emailyak->newMail",
  "ToAddressList": "test@mscliveweb.simpleyak.com",
  "FromAddress": "dominic.bosch.db@gmail.com",
  "TextBody": "Lengthy body [...]",
  "Subject": "Fwd: test subject",
  [...]

}
```

Chapter 3

The "XYZ" Prototype System

The "XYZ" prototype system is the realisation of a reactive web system. It was developped during the research for this thesis and acts as a platform for feasibility studies of certain use cases.

3.1 Architecture

"XYZ" consists of a queue in which all incoming events are pushed, and an engine that picks the events from the end of the queue whenever it is idle. Since "XYZ"'s core functionalty is the communication with resources in the web, the architecture bases on HTTP protocol in several parts. For example the events are meant to be retrieved completely via HTTP, the user interface is a webpage which posts requests to the system and most actions are also meant to be HTTP requests, or at least using them to gather information.

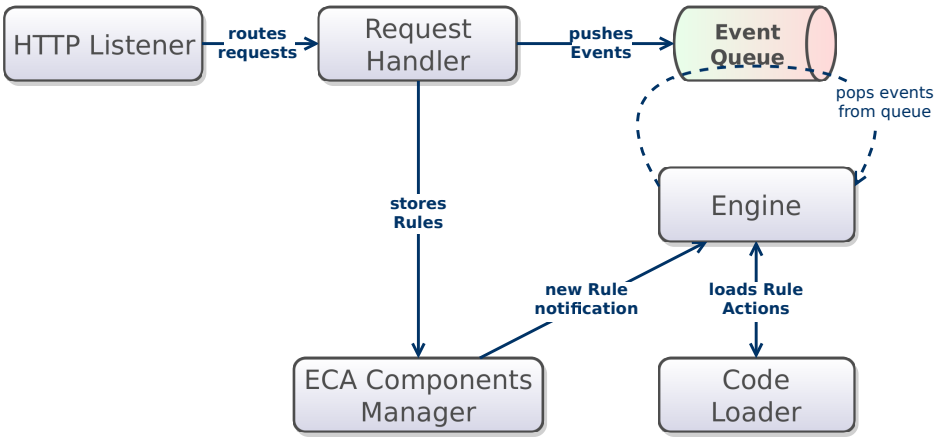

Figure 3.1: "XYZ" Architecture

3.2 Event Triggers

Event Gathering is the E in ECA and without one of these letters such a system would not run. It is of utmost importance to find as much as possible ways to get data into a system.

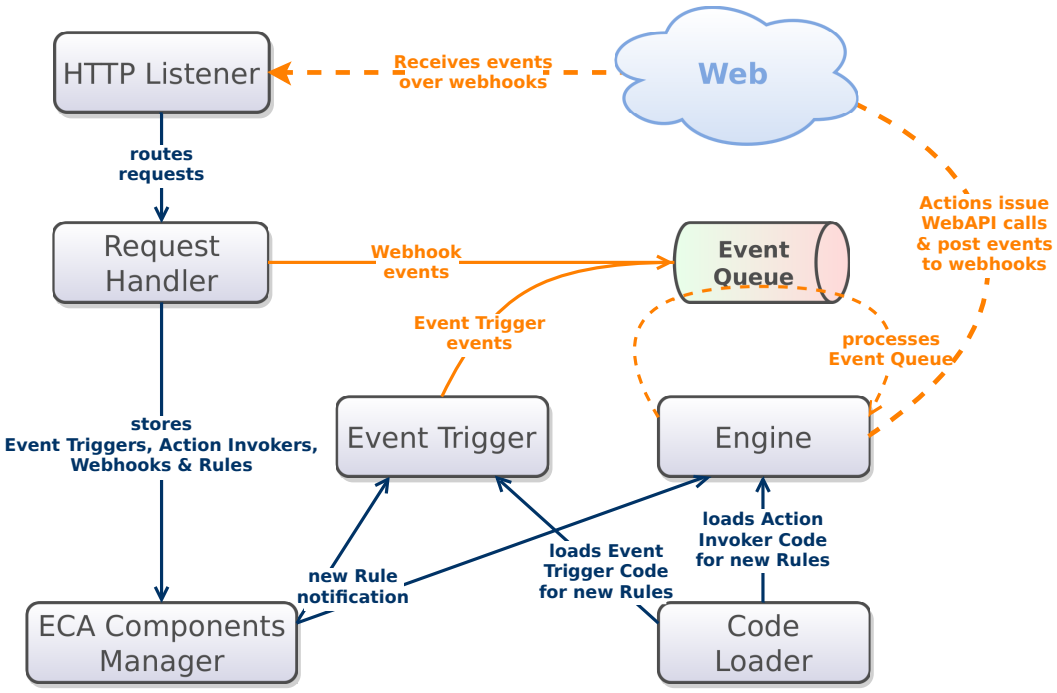

Figure 3.2: "XYZ" Architecture

3.2.1 Polling

3.2.2 Webhooks

Which

3.3 Conditions

3.4 Actions

3.4.1 Aditional Information

3.5 ECA Rules

3.6 Engine

3.7 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 3.3, 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.

Asynchronous code execution, as shown in Figure 3.4, allows non-blocking and thus scalable applications. Non-blocking operations are a remedy for optimzed resource allocation

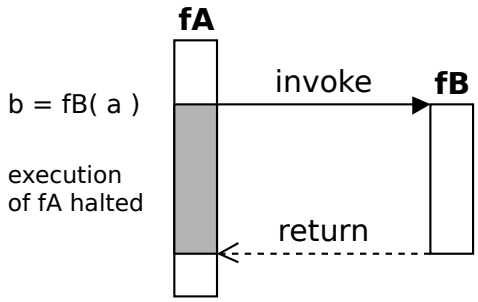

Figure 3.3: Synchronous Function Call

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.

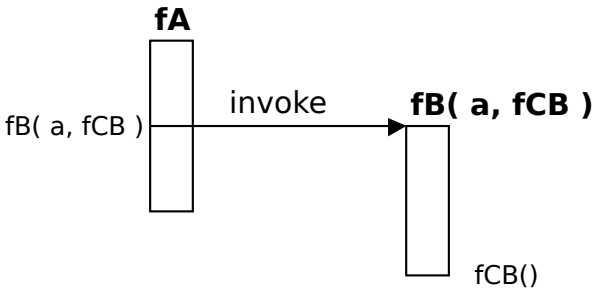

Figure 3.4: Asynchronous Function Call

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 3.4. 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.

Now we take closures into this asynchronous context, as defined in ECMAScript[4], which is the base for widely-spread script languages like JavaScript, JScript and ActionScript. Closures in ECMAScript[4] are defined such as they have access to the context of the function they were created in. This is shown in Figure 3.5 where `c` from `fA`'s context is accessible from within `fB`, assuming that `fB` was created in `fA` and not only invoked from there. 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 3.6. In such a setup `c` will have different values in the same part of different invocations of `fB`. This might be a very well hidden pitfall since often developers will take care that the context will not change during execution of `fA`. But it is likely that the context will change if `fA` is invoked again while `fB` is still running, which is a common behaviour in event-driven architectures.

Those event-driven context overwrites can be taken care of by shielding the closure from context changes, as shown in Figure 3.7. To shield the closure form context changes, closure `fB` needs to create another closure `fC` and return it to `fA`. The argument passed to `fB` is the

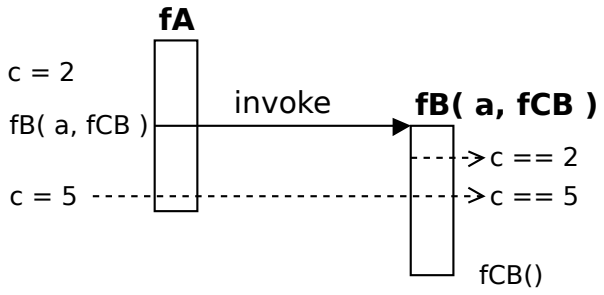

Figure 3.5: Closure Scope and referenced context

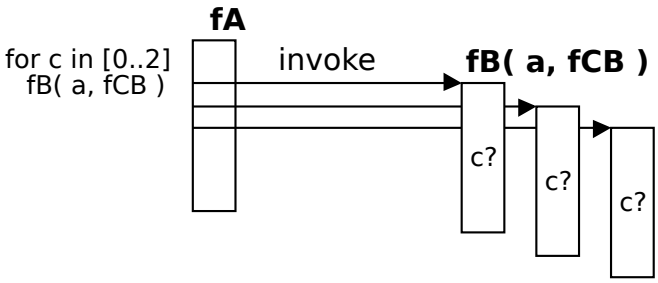

Figure 3.6: Closure context changes in a loop

context (`c` in Figure 3.7) 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.

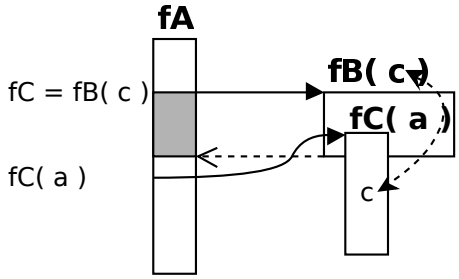

Figure 3.7: Closure context shielding

3.8 "XYZ" Name discussion

Examples for XYZ:

- **GEEK** (General-Purpose ECA Engine Kernel)
- **DECADE** (Dynamic ECA Demo Engine)
- **WECAST** (WebAPI ECA Service Trigger)
- **RECAST** (Reactive ECA Service Trigger)
- **PECAN** (Productive ECA eNgine)
- **ICECAP** (Inet-Service Calls through ECA Paradigm)

Chapter 4

Discussion & Results

4.1 Future Work

We have seen that the ECA approach is already a powerful one to make the web reactive. A future improvement of this could be to adopt Complex Event Processing (CEP). This would mean that several events could be stored in a rule and be evaluated in terms of time constraints. Through this more complex events can be created as a result of several atomic events which would lead into semantically more complex events. A change in paradigm will result in an approach where events are not just processed when they are entering the system and evaluated against rules, but these events would need to be stored for quite a long time. Also the rules will not all be checked for each event but they are subject to a scheduler. It can be decided when and how often a rule is evaluated and all events will be checked at these point in times, whether they are candidates for firing the rule. A relational database will be needed in order to search through the timestamps

Bibliography

[1] Tim Berners-Lee. Notation 3 Logic. `http://www.w3.org/DesignIssues/Notation3.html`, 2005. Accessed: 2013-10-21.

[2] Harold Boley. The RuleML family of web rule languages. In *Principles and Practice of Semantic Web Reasoning*, pages 1–17. Springer, 2006.

[3] Adam DuVander. 5 Years Ago Today the Web Mashup Was Born. `http://blog.programmableweb.com/2010/04/08/the-fifth-anniversary-of-map-mashups-on-the-web`, 2010. Accessed: 2014-05-01.

[4] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1 edition, June 2011.

[5] Free Wifi & Plugs — Your comprehensive list of where to work around London. `http://wifiandplugs.co.uk`. Accessed: 2014-05-02.

[6] Adrian Giurca and Emilian Pascalau, Json rules. *Proceedings of the Proceedings of 4th Knowledge Engineering and Software Engineering, KESE*, 425:7–18, 2008.

[7] MapLight - Money and Politics — U.S. Congress Campaign Contributions and Voting Database. `http://maplight.org`. Accessed: 2014-05-02.

[8] George Papamarkos, Alexandra Poulovassilis, Ra Poulovassilis, and Peter T. Wood. Event-Condition-Action Rule Languages for the Semantic Web. In *In: Workshop on Semantic Web and Databases*, pages 309–327, 2003.

[9] Adrian Paschke and Harold Boley. Rules Capturing Events and Reactivity. In Adrian Giurca, Dragan Gasevic, and Kuldar Taveter, editors, *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*, pages 215–252. IGI Publishing, May 2009.

[10] 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.

[11] Paula-lavinia Patranjan. *The Language XChange*. PhD thesis, Ludwig-Maximilians-Universität München, 2005.

[12] Joshua Porter. Holy Amazing Interface, Batman! Paul Rademachers Brilliant Lodging Finder. `http://bokardo.com/archives/holy-amazing-interface-batman`, 2005. Accessed: 2014-05-01.

[13] ProgrammableWeb: APIs, mashups and code. Because the world's your programmable oyster. `http://www.programmableweb.com`. Accessed: 2014-05-02.

[14] Shared Count. `http://lab.neerajkumar.name/sharedcount`. Accessed: 2014-05-02.

[15] P. Windley. *The Live Web: Building Event-Based Connections in the Cloud*. Cengage Learning PTR, 2011.

List of Figures

List of Tables

Listings

Index