Towards The Reactive Web

Master Thesis

Dominic Bosch
Departement Mathematics and Computer Science
University of Basel

May 4, 2014

Abstract

Contents

0.1 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)

0.1.1 Related Work

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 [7, 13] 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 [3] directory, are:

- Wifi and Plugs [1]: MapBox, Google Docs and Import.io API's used to display where Wi-Fi and plugs are available in London.
- MapLight [2]: GovTrack.us and OpenSecrets API's used to combine political results with financial contributions to show how capital contributions affect voting.
- Shared Count [4]: 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.

0.2 Conceptual Model for Reactive Web Systems

0.2.1 Event Condition Action (ECA) Model in the Web

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.

0.3 The "XYZ" Prototype System

0.3.1 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.

Polling for Events

Webhooks

Which

0.3.2 Actions

0.3.3 ECA Rules

0.3.4 Architecture

0.3.5 Asynchronous Systems & Closures

Certain optimization approaches and programming language concepts require special attention to avoid common pitfalls. If such approaches and concepts come together, which is the case when closures are used in asynchronous systems, random inconsistencies will be the result if they are not taken care of.

Looking at an example of sequential code execution (Figure 1), 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.

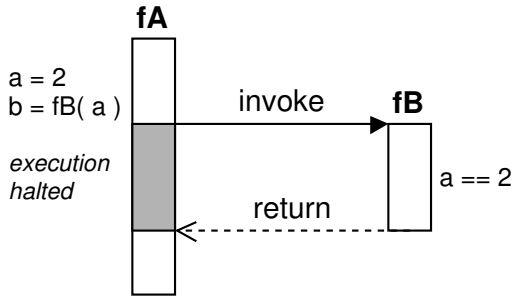

Figure 1: Synchronous Function Calls

Non-blocking operations are a remedy for optimzed resource allocation and opens ways to overcome such unnecessary resource bindings. Asynchronous code execution allows non-blocking and thus scalable applications.

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. Often other operations depend on the completion of asynchronous operations, hence their execution needs to be deferred. This required code execution deferral is achieved through the application of callback functions. Any code placed in a callback function, which is assigned to an asynchronous operation, is only executed when the respective asynchronous operation completed. This leads to stacking of functions and operations

Such callback functions are passed as arguments to asynchronous operations and executed as soon as the operation finishes.

By deferring the executiong of operations that depend on the

asynchronously results in dynamic code which exploits .

together with closures they

Test reference Figure 1

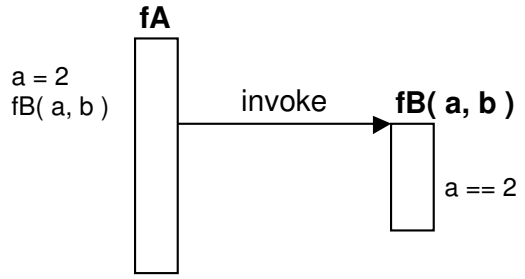

Figure 2: Asynchronous Function Calls

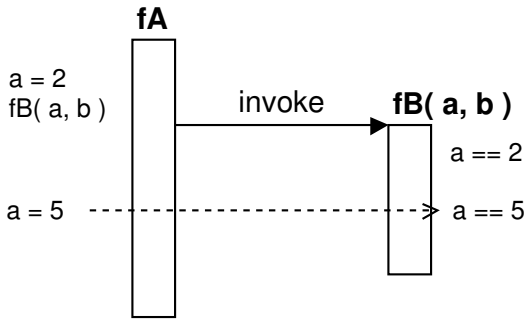

Figure 3: Closure Scope

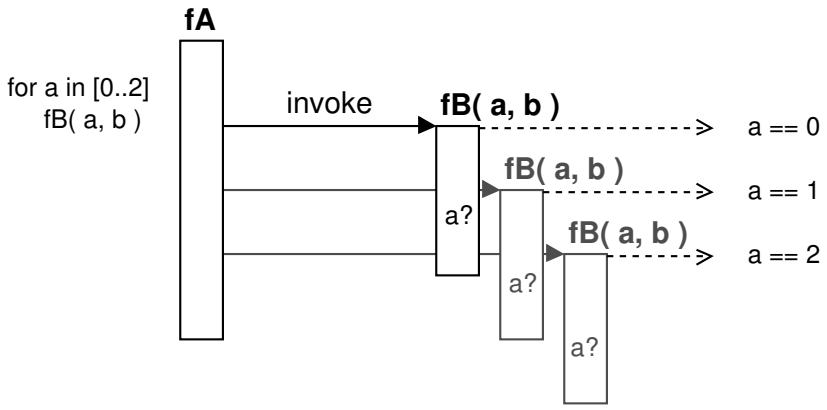

Figure 4: Closure Scope

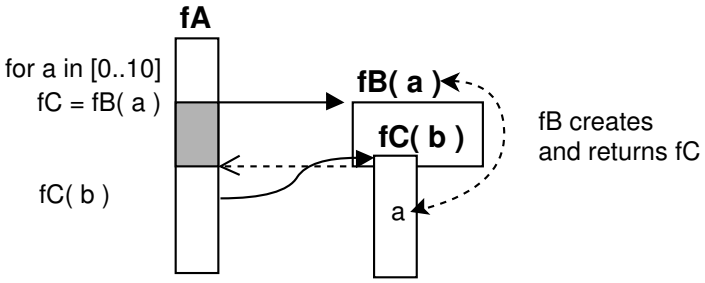

Figure 5: Closure Scope

0.4 Discussion & Results

0.4.1 Use Cases

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

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

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

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

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

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

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

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

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

```
'use strict';
var express = require('express');
var qs = require('querystring');
var engine = require('./ecainference');
```

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)

Rules Languages	Classification
Language XChange Francois Bry, Paula-Lavinia Patranjan	• EU & Swiss project • Paradigm • Event-driven reactivity • Influences into all other research in the field of web reactivity • Discontinued since 2008
JSON Rules Adrian Giurca, Emilian Pascalau	• JSON based rule language • (DOM-) Event-based reactivity • Discontinued since 2009
RuleML Harold Boley, Adrian Paschke	• XML based rule language • Event-driven reactivity • "Cloud Application Access"

Figure 6: Examined Rules Languages

[10] gave a good oveview over existing approaches in 2009. 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:

```
{
  "event": "email",
  "sender": "sender@mail.com",
  "subject": "An important message!"
}
```

An early ECA Rule Language for XML repositories [9] 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>)
```

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.

To make the lengthy RDF definitions smaller and more readable, Notation 3 [5] 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:

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

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 eventully still find influence into our solution.

The rule language XChange [12] 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:

```
TRANSACTION
  in {
    resource { "http://www.webapi.com"},
    newcontents {{
      insert newcontent { var Mail }
    }}
  }
ON
  xchange:event {{
    xchange:sender { "http://mailserver.com" },
    var Mail -> email {{
      sender { "sender@mail.com" }
    }}
  }}
END
```

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
```

In 2008 *JSON Rules* [8] 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)"
  ]
}
```

A most recent (2011) open-source development is the Kinetic Rules Engine together with the Kinetics Rule Language [14]. 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
      }
}


ruleset a2236x5 {
  rule register_temperature {
    select when temperature update
      if (event:attr("temp") > 20
        && ent:old_temp <= 20) then {}
      fired {
        raise explicit event temp_over_20;
      }
      always {
        set ent:old_temp event:attr("temp");
      }
  }

  rule temp_over_threshold {
    select when explicit event temp_over_20
      http:get("https://" + ent:credentials
        + "@probinder.com/service/27/save?companyId="
        + ent:companyID + "&context=" + ent:contextID
        + "&text=temp over 20 degrees.");
  }

}
```

The basis of *RuleML* [6] is datalog, a language in the intersection of SQL and Prolog. In 2012 the *Reaction RuleML* [11] 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>
```

```
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",
  [...]

}
```

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.