Master Thesis Preparation Report

Dominic Bosch
Departement Mathematics and Computer Science
University of Basel

July 17, 2013

Abstract. In this report we summarize the research efforts during the preparation work for the master thesis. While identifying related work, it became clear that huge efforts in the research fields of event-driven architectures and user-driven mashup development has been done. On the other hand cloud application utilization is a most recent research field and continues to be a grand challenge as they evolve. Moreover the use of cloud applications seems to be a nice feature, rather than fonding the basis of research. To the best of our knowledge, no related work was found which funnels all these research fields into one, providing novel powerful ways to govern the web.

1 Introduction

The web continuously evolves into bigger complexity, allowing for ever more powerful applications. The grand challenge is to retain manageable interactions between cloud applications, while adopting reactivity to them. To get a hold on this, we anticipate the next change in the evolution of the web: the live web, or reactive web. By considering cloud applications as event producers and action consumers we are able to apply a new level of abstraction to the web, which allows new perspectives and approaches to manifest the reactive web. Such an architecture has the potential to overcome limitations that have been encountered during previous research.

2 Related Work

2.1 Reactivity in the Web

Research approaches to apply reactivity to the web has been done by several researchers in many fields. One notable outcome is the language *XChange* [5, 22], which introduces reactivity to the web. *XChange* uses *Xcerpt* [24], a rule-based query and transformation Language for the web, to express web queries.

In [19] the authors provide an overview of general descriptions and classifications from different research efforts in terms of events, rules and reactiveness. The different research domains they point out are:

- Event/Action Logics, Transition Logics and Process Calculi. Used in [2] to specify complex actions
- Dynamic/Update/Transition Logics
- Production Rule Systems (if-do)
- Active Databases and ECA Rule Systems (on-if-do)
- Rule-Based Complex Event Processing (CEP) and Event Notification Systems

In contrast to standard ECA rules, which typically only have one global state, messaging reaction rules maintain a local conversation state that refelects the process execution state. This supports the performing of different activities within process instances managed in simultaneous conversation branches. Today, powerful event languages (such as *RuleML*) exist, which allow to express each of the afore mentioned categories.

In [10] semantic application vocabulary and business rules are examined, which together allow smart suggestions to the user in a rule-based system. Their approach bases on Distributed Object Model (DOM) interactions and mashups would have to be constructed indirectly via the DOM. Their work is interesting when allowing the users to customize their online context but it doesn't allow for cloud applications to be first-class citizens as part of on- and offline mashups.

From [29] follows, that enhancing a Service oriented Architecture (SOA) with an event-driven SOA (EDSOA), leads to more flexible and adaptive SOA applications that can be informed about states of neighbouring components. In their approach, events are only an aid to react on unexpected behaviour. But our research path leads us towards a system which builds entirely on events, using their expressive power as a basis.

2.2 Rule Engines and Languages

En excerpt of rule engines and languages is shortly introduced here. The recent trend seems to go towards funneling the different event-based approaches into CEP as highly expressive paradigm to incorporate all others. Of course more rule engines or languages exist, but they either go into similar categories as the presented ones, aren't developed anymore, or haven't been identified in related work as recent or active research.

2.2.1 Kynetx

The Kinetic Rules Engine (KRE) is a platform presented in [27]. It is realized in Perl and uses its very own rule language, the Kinetic Rules Language (KRL). It is layed out to support CEP as well as a tight coupling with the user's browser through plugins or libraries loaded via the webpage. It allows the access to remote resources and the processing of such data before passing it along to internal storage or again external resources, such as cloud applications. A live system [28] is available for testing and if desired also for productive use. Creating an own instance is quite a challenge due to it's numerous libraries. KRL needs quite some efforts to get used to and can't be entrusted to unexperienced users, thus a layer on top of this system would have to be implemented for our purposes.

2.2.2 (Reaction) RuleML

RuleML [3] is a XML-based rule specification standard to express both forward and backward rules for derivation, reaction, rewriting, messaging, verification and transformation. The building blocks of *RuleML* are predicates, derivation rules, facts, queries, integrity constraints and transformation rules. Its development is driven by the Rule Markup Initiative [4].

With *RuleML* being already a useful specification, *Reaction RuleML* [21] extends *RuleML* towards reaction rules and complex event/action messages, e.g. for CEP. It adds various kinds of production, action, reaction and knowledge representation (KR) temporal/event/action logic rules, as well as (complex) event/action messages. It consists of one general reaction rule form that can be specialized, e.g. into production rules, trigger rules, ECA rules or messaging rules. Three different execution styles (active, messaging and reasoning) of rules are incorporated. Definition of inbound or outbound event messages and are used to interchange events and rule bases. A reaction rule can be globally or locally nested within other reaction or derivation rules. Additionally the *RuleML* Interface Description Language

(*RuleML IDL*) was provided in the same paper, a sub-language of *Reaction RuleML* and allows the description of public rule functions as interfaces to hide program logic.

2.2.3 (OO) jDrew

Java Deductive Reasoning Engine for the Web (*jDrew* [25]) is a reasoning engine written in Java for definite clause reasoning. jDrew can be embedded into larger systems through its APIs. Object-Oriented jDrew (*OO jDrew* [1, 26]) is a Java based rule engine, it serves as a reference implementation of *RuleML*.

2.2.4 Prova

Prova [12] is an expressive rule language and engine, both written in Java, with a main orientation to ECA rules. It uses backward-reasoning logic to formalize decisions in terms of derivation. Forward-directed messaging of reaction rules supports distributed event and action processing. It allows dynamic access to external data sources and is used by the authors of [20, 30] for the *RuleResponder's* proof of concept for transformations between different rule languages over *RuleML*.

2.2.5 JSON Rules

JSON Rules [9] has been invented due to an increasing need for rules in terms of semantic web applications and the emerging Rich Internet Applications (RIAs). *JSON Rules* is capable of expressing production rules (if-do) as well as ECA rules (on-if-do). The rules engine is a forward chaining rule engine using a modified RETE [8] algorithm. The RETE working memory in this rule engine is the (event-based) Document Object Model (DOM) itself. An example [17] of (DOM-)rule-based creation and execution of mashups illustrates the powerful aspects of *JSON Rules*. The current limitation to the DOM tree would require a generalization to adopt it also to other memory layouts.

2.3 Mashups

In [15], one of the founders of *JSON Rules* proposes to look at mashups as user-behaviour in a certain context. A mashup, to achieve a user-defined goal, is modelled via Unified Modeling Language (UML) as a map containing contexts and behaviour desriptions. Concepts are defined as unit of knowledge created by unique combination of characteristics. Contexts are defined as a set of concepts. A mashup is then defined as set of contexts behaviour, where behaviour consists of rules and processes. The conceptual work done in this paper is interesting but the promoted example isn't accessible.

It is also important to understand what users expect from service mashups, in order to provide a useful platform to them. In [14] research is done in identifying user perceptions of services, their composition, user working ways and expectations towards a composition tool. They come up with a set of recommendations to aid the development of mashup environments. A survey [7] that went into a similar direction categorizes the different frameworks for user driven mashup development as based on:

- Programming Paradigm
- Scripting Languages
- Spreadsheets
- Wiring Paradigm
- Programming by Demonstration
- Automatic Creation of Mashups

But even though large efforts are made in all these research fields, unexperienced users are still not able to build mashups without knowledge about numerous aspects of the framework or programming.

2.3.1 useKit

The idea of *useKit* [23] missions shows us the potential of user-manageable cloud application mashups. While their approach is not event-based, it can be regarded as a base for the web's evolution towards user-programmable reactive cloud application mashups.

2.3.2 Rule Responder

Rule Responder [20] is a project to extend the Semantic Web towards a Pragmatic Web infrastructure for collaborative human-computer networks, which they call an architecture of a Pragmatic Agent Web (PAW). It supports the formation of virtual groupings and allows semi-automated agents with their individual contexts, decisions and actions. The authors postulate agents empowered with automatic rule-driven data transformation, decision derivation from existing knowledge and reaction according to changed situations or occurred events. The work done in this project concentrates on a layer on top of a rule engine and language, and thus allows for a combination of arbitrary rule-based systems via their framework. This is achieved through the usage of general message oriented communication interfaces and a platform-independent rule interchange format (*RuleML*).

The authors of Rule Responder built their reference system [18] on top of the Mule [13] open-source Enterprise Service Bus (ESB) which acts as a communication middleware. The decision to use Mule was made because it goes beyond the typical definition of an ESB by providing a distributable object broker to manage all sorts of service components. Each agent runs its own arbitrary rule engine. For demonstration purposes *Prova* and *OO jDrew* were used to demonstrate the rule interchange between different rule engines.

As research continued in terms of reaction rules and *Rule Responder*, the authors of [30] showed the adoption of event paradigms to support scientific workflow execution. In their work they point out the limitations of ECA frameworks when adopted to their use case. For highly distributed and loosely coupled scientific workflows, complicated conditional procedures and rules, which can also have local scopes, are required. This shows us their work is going towards large distributed systems with a highly developed rule language that subsumes research from several fields.

2.3.3 DashMash

The DashMash [6] platform is an approach to give end-users the graphical tools in a browser to mash up web applications in a dashboard. A resource of (for stability reasons) trimmed services (such as GoogleMaps or TripAdvisor), filters, viewers and generic components is accessible to the users. DashMash uses an event-driven model of the presentation level, similar to a JSON Rules approach in [17]. There are events sent by the client to the server, but they are only used to update all viewers with the actual data the user is looking at.

3 Use Case Study

In order to verify some of the identified related work, use cases around the successor of useKit [23] (ProBinder [**?**]) have been derived and investigated.

Three use cases have been identified to verify the feasbility of certain existing infrastructures.

3.1 Binder Watcher

Binder Watcher is about binders being watched and actions that are taken after certain changes to a binder. Users of ProBinder, which are involved in many different companies and project binders, tend to be confronted with a large amount of information. It is a tedious task to get the user's context back into a clean state, where the ProBinder system is ready to reflect new recent changes in an optimal way to the user. By allowing the users to identify resources (binder tabs in this case, but it could also be complete binders, persons, companies, ...) of interest, the user task can be automated to a certain extent. As soon as changes are made to the resources of interest, they are marked as read and summarized. These summaries are then provided to the user, which allows him to identify the most important changes.

The Binder Watcher use case was implemented in KRL (see Appendix A) and provided the important insight that the realization of such a use case in an ECA is a time-consuming challenge.

3.2 Web Watcher

To bring reactivity to the static web, we have to enhance legacy resources, which are not event-ready. Changes in such resources have to be detected via e.g. web spiders or crawlers and transformed into events that can be processed by the rules engine. Such an auxiliary cloud application would turn the static web into a reactive system and free a future architecture from incorporating the pulling of events, thus allowing it to concentrate on its main business, i.e. event handling. A web watcher was implemented to monitor a predefined ProBinder tab and push changes as events into our proof of concept system, which is based on *node.js*. These events are processed according to predefined rules and the action consumer was again *ProBinder* which was used to creat an entry in another binder. This use case was mainly used to proof the accessibility of ProBinder via a remote system and the activation of the static web.

4 Conclusion

In [16], the founders of *JSON Rules* [9] describe a lightweight architecture that allows to react and proact on behalf of events in the ontology of web browsers. *JSON Rules* is promising for our work because of its lightweight architecture and specialization on production and ECA rules. But the existing working memory architecture needs to be generalized to allow a different environment, other than just the DOM tree. The existing architecture could be used to allow the user to create local rules that do not access remote systems and thus runs into authentication issues.

Other approaches are server side rule engines either written in Perl or Java, where powerful tools are provided to process events and invoke manifold actions. But these approaches all lack an abstraction layer to introduce the programmability of the reactive web to a large audience of unexperienced users. Also, current approaches concentrate on data flows instead of event flows, thus not incorporating reactivity to the web.

With *RuleML* a powerful, interchangeable expression language for event-based systems is present. It paves the way for distributed rule engines, such as one running in the browser and another on a server for cloud application access. But a rule engine that incorporates the requirements of a lightweighted architecture and user-readiness is missing. Under related work, [14] pointed out the difficulties of unexperienced users to tackle the execution flow. This issue arose from their approach of displaying all services as very similar UI's. Adopting an event-based system. Event producers are clearly different from action consumers would address this issue in an user-intuitive way.

Large systems such as *RuleResponder* weave stubs or proxies of existing service into a message oriented middleware (MoM). We envision the web itself is used as the middleware. Through this a lightweighted and performant event-based architecture can be realized, which allows the orchestration of existing web and cloud applications.

5 Future Work

Developing an event-driven architecture which regards cloud applications as event-based first-class citizens and allows for an intuitive user-driven mashup development is a research field that has, to the best of our knowledge, not been addressed yet. Looking at cloud-based applications as event producers and action consumers gives new ways to bring reactivity into the existing web. Such representations require a rule-based system that allows their interweaving. Not solely the interaction between cloud applications should be addressed, but also with the browser itself, since it is a tool which is predominantly used to access the web. This would empower the user to predefine influences and interactions on existing cloud applications before they are accessed, providing novel powerful ways to govern the web.

Since the vision of on- and offline rules can't be covered with a single server application, the utilization of a fast, flexible and widely used technology such as JavaScript to tackle this challenge seems to be favorable future work. JavaScript was mainly invented for browsers and is spread all over the web by now. Additionally, applications such as *node.js* [11], bring JavaScript to the server-side and tear down the communication efforts between cloud applications through JSON messages which are directly understood by modern browsers and cloud applications. Preferably a lightweighted rules engine would be used to run the user-generated mashups. The KRE suits the demand for a certain coupling between the users browser and the remote rules engine to provide a powerful system. On the other hand the rules engine is not (yet) well documented, not lightweighted and forged in Perl, a programming language that wasn't encountered during the research for related work on rule based systems.

Sharing and thus exchanging of rules gives new ways for collaboration and the possibility for expert users to aid less experienced ones, giving them the chance to catch up with the future platform. In terms of usability an easy to understand way to create rules would have a large benefit. We envision a graphical toolkit that empowers users to build their own complex event-based cloud application mashups, powerful RIAs. A first approach could be to write rules in a certain language, e.g. *RuleML* or *JSON Rules*, then simplify the vocabulary until meaningful graphical representations are possible.

It is striking but not surprising, that all related work we looked at, only used public cloud applications, thus omitting the challenge of authorization and the safety of the user's private context. If users are provided the tools to access public cloud APIs and create mashups with them, they are going to benefit from this. But mostly users these days are traveling in the web within their private, secured context. Thus the access to such resources are providing even more power- and meaningful tools to the user. The future reactive web requires cloud applications that allow the registration of webhooks in order for the rules engine to receive events from resources other than the own system, thus using the internet as middleware for communication of push events.

References

[1] M. Ball, H. Boley, D. Hirtle, J. Mei, and B. Spencer. The OO jDREW Reference Implementation of RuleML, 2005.

[2] E. Behrends, O. Fritzen, W. May, and F. Schenk. Embedding Event Algebras and Process for ECA Rules for the Semantic Web. *Fundam. Inf.*, 82(3):237–263, August 2008.

[3] H. Boley. The RuleML Family of Web Rule Languages, 2006.

[4] H. Boley and S. Tabet. The Rule Markup Initiative. `http://ruleml.org`. Accessed: 2013-07-07.

[5] F. Bry and P.-L. Patranjan. Reactivity on the Web: Paradigms and Applications of the Language XChange, 2005.

[6] C. Cappiello, M. Matera, M. Picozzi, G. Sprega, D. Barbagallo, and C. Francalanci. DashMash: A Mashup Environment for End User Development, 2011.

[7] T. Fischer, F. Bakalov, and A. Nauerz. Mashups: Behavior in Context(s), 2009.

[8] C. L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, 1982.

[9] A. Giurca and E. Pascalau. JSON Rules, 2008.

[10] A. Giurca, M. Tylkowski, and M. Mueller. RuleTheWeb!: Rule-based Adaptive User Experience, 2012.

[11] Joyent Inc. node.js. `http://nodejs.org/`. Accessed: 2013-07-07.

[12] A. Kozlenkov and A. Paschke. Prova Rule Language. `https://prova.ws/`. Accessed: 2013-07-07.

[13] R. Mason. muleESB. `http://www.mulesoft.org`. Accessed: 2013-07-07.

[14] A. Namoun, T. Nestler, and A. De Angeli. End User Requirements for the Composable Web, 2010.

[15] E. Pascalau. Mashups: Behavior in Context(s), 2011.

[16] E. Pascalau and A. Giurca. A Lightweight Architecture of an ECA Rule Engine for Web Browsers, 2009.

[17] E. Pascalau and A. Giurca. A Rule-Based Approach of Creating and Executing Mashups, 2009.

[18] A. Paschke. Rule Responder - Rule-based Semantic Agent Architecture. `http://www.corporate-semantic-web.de/rule-responder.html`. Accessed: 2013-07-14.

[19] A. Paschke and H. Boley. Rules Capturing Events and Reactivity, 2009.

[20] A. Paschke, H. Boley, B. Craig, and A. Kozlenkov. Rule Responder: RuleML-Based Agents for Distributed Collaboration on the Pragmatic Web, 2007.

[21] A. Paschke, H. Boley, Z. Zhao, K. Teymourian, and T. Athan. Reaction RuleML 1.0: Standardized Semantic Reaction Rules, 2012.

[22] P.-L. Patranjan. *The Language XChange*. PhD thesis, Ludwig-Maximilians-Universität München, 2005.

[23] S. Rizzotti and H. Burkhart. useKit - Lightweight Mashups for the Personalized Web, 2010.

[24] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. PhD thesis, Ludwig-Maximilians-Universität München, 2004.

[25] B. Spencer. jDREW. `http://sourceforge.net/projects/jdrew/`. Accessed: 2013-07-07.

[26] B Spencer, T. Athan, and B. Craig. OO jDREW. `http://extoojdrew.weebly.com/index.html`. Accessed: 2013-07-07.

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

[28] P.J. Windley. kynetx apps. `http://apps.kynetx.com/`. Accessed: 2013-07-07.

[29] C. Ye and H.-A. Jacobsen. Event Exposure for Web Services: A Grey-box Approach to Compose and Evolve Web Services, 2010.

[30] Z. Zhao and A. Paschke. Event-Driven Scientific Workflow Execution, 2013.

Appendix A Binder Watcher KRL code

```
ruleset a2236x4 {
  meta {
        name "ProBinder Flag Notification Handler"
        description "This is a first example on how to react on ProBinder Events"
        author "dominic.bosch"
    //ProBinder IDs:
    // userID: 10595
    // companyID: 643
    // contextID: 16694
    // followerID: 12613

        logging on
  }

  dispatch {}

  global {}

  // Reset all entitiy variables
  rule resetAll {
    select when probinder resetall
      send_directive("Full Reset");
      fired {
        clear ent:userID;
        clear ent:companyID;
        clear ent:contextID;
        clear ent:credentials;
        clear ent:followers;
        clear ent:newContents;
        clear ent:summary;
        clear ent:temp;
      }
  }

  // reset the unread content data structures
  rule reset {
    select when probinder reset
      send_directive("Reset, user credentials and followers still kept");
      fired {
        clear ent:newContents;
        clear ent:summary;
        clear ent:temp;
      }
  }

  // The user registers himself with email and password for the ProBinder API...
  rule register_user {
    select when probinder register
      if (event:attr('userID').as("str") neq 'null'
          && event:attr('companyID').as("str") neq 'null'
          && event:attr('contextID').as("str") neq 'null'
          && event:attr('email').as("str") neq 'null'
          && event:attr('password').as("str") neq 'null') then {
        send_directive("user registered");
      }
      fired {
        set ent:userID event:attr('userID');
        set ent:companyID event:attr('companyID');
        set ent:contextID event:attr('contextID');
        set ent:credentials uri:escape(event:attr('email')) + ":" + uri:escape(
            event:attr('password'));
      }
  }

  // The user sent an event that tells us he wants to follow somebody
  rule new_user_to_follow {
    select when probinder newfollower
      pre{
        listFollowers = ent:followers || {};
        newfollower = event:attr('followerID').as("str");
        listFollowers = listFollowers.put([newfollower], "true");
      }
      if (event:attr('userID') == ent:userID
```

```
        && newfollower neq "null") then {
          send_directive("New ProBinder User added to followers");
    }
    fired{
      set ent:followers listFollowers
    }
}

// Let the KRE check ProBinder for new unread content and process it
     immediately
rule check_for_unread_content {
  select when probinder check
    pre {
      r = http:get("https://" + ent:credentials + "@probinder.com/service/36/
           unreadcontent");
      arr = r{"content"}.decode();
    }
    send_directive("Checked ProBinder for unread content, found: " + arr.length
         ());
    fired {
      set ent:newContents arr;
      raise explicit event processnewcontents;
    }

}

// Work (new unread content) from ProBinder to process
rule process_new_contents {
  select when explicit processnewcontents
  // Process only the unread contents from people we are following,
  // filter condition omits unnecessary rules invocation
  foreach ent:newContents.filter(
    function(d) {ent:followers.pick("$."+d.pick("$.userId")) != null}
  ) setting(nc)
    pre {
      s = ent:summary || {};
      cid = nc.pick("$.id");
      r = http:get("https://" + ent:credentials
        + "@probinder.com/service/2/get?id=" + cid
        + "&service=" + nc.pick("$.serviceId"));
      arr = r{"content"}.decode();

      userid = arr.pick("$.userId");
      storeKey = arr.pick("$.lastModified");
      truncStr = arr.pick("$.text");//.extract(re/^.{100}/gi); // should
           shorten the text...

    //TODO Process different kind of unread contents differently
      str = {"content": truncStr}; //[0]
      s = s.put([userid, storeKey], str);
    }
    http:get("https://" + ent:credentials + "@probinder.com/service/2/setread?
         id=" + cid);
    always {
      set ent:summary s;
    }

}

rule send_summary{
  select when probinder heartbeat
  always {
    clear ent:temp;
    raise explicit event filltemp;
  }
}

rule fill_temp{
  select when explicit filltemp
  always {
    set ent:temp ent:summary;
    raise explicit event mergecontent;
  }
}

// When somebody sends a periodic heartbeat, this summary is produced
```

```
  // The periodic invocation of this rule might be possible to implement in the
      KRE
  rule merge_content {
    select when explicit mergecontent
    foreach ent:temp setting (userID)
      pre {
        s = ent:temp;
        userBulk = s.pick("$."+userID);
        sumry = userBulk.pick("$..content").join(" ");
      }
      http:get("https://" + ent:credentials + "@probinder.com/service/27/save?
          companyId="
        + ent:companyID + "&context=" + ent:contextID + "&text=test");
      send_directive("Stored summary in your predefined binder:" + sumry);

  }

  rule print_summary {
    select when probinder printsum
      send_directive(ent:summary);
  }
}
```

Appendix B Web Watcher

Appendix B.1 node.js code

Appendix B.2 web crawler code