Master Thesis Preparation

Dominic Bosch
Departement Mathematics and Computer Science
University of Basel

July 15, 2013

Abstract. The web continously evolves into bigger complexity, allowing for ever more powerful applications. The grand challange is to retain manageable interactions between cloud applications, while applying 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 consumers we are able to apply a different level of abstraction to the web, which allows new perspectives and approaches to manifest the reactive web.

1 Introduction

2 Related Work

In [7] the authors supplied general descriptions and classifications of different research efforts in terms of events, rules and reactiveness. Particularly of interest is their identification and summarization of existing research:

- Event/Action Logics, Transition Logics and Process Calculi: Events/Actions transit states and effect the lifetime of changeable properties (fluents). Used in [1] to specify complex actions, or in to model the communication behaviour of inbound and outbound message links in rules.
- Dynamic/Update/Transition Logics:
- Production Rule Systems:
- Active Databases and ECA Rule Systems:

- Rule-Based Complex Event Processing and Event Notification Systems: In such approaches the communication is often eased by using a middelware such as service buses. The upcoming paradigms of service-oriented architecture (SOA) and event-driven architecture (EDA) such systems allow for the reaction to the fashionable complex events. The applied reactive rules are executed to a certain context,

Language XChange uses Xcerpt(?) to express web queries. MARS was postulated in 2008 (not available?). 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 exe3cution state. This supports supports the performing of different activities within process instances managed in simultaneous conversation branches. CEP provides enhanced situation awareness.

2.1 Rule Engines

2.1.1 Kynetic Rules Engine

A framework presented in [12]

2.1.2 Rule Responder

Rule Responder [8] 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 messge oriented communication interfaces and a platform-independent rule interchange format.
The authors of Rule Responder built their reference system on top of the Mule [5] 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 rbitrary rule engine. For demonstartion purposes Prova and OO jDrew were

used to demonstrate the rule interchange between different rule engines. An investigated use case for Rule Responder was a symposium organization as a virtual organization.

2.1.3 Prova

Prova is a highly expressive rule language with a clear orientation towards CEP. 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 is used by the authors of [8, 13]. It allows for dynamic access to external data sources.

2.2 Rule Languages

2.2.1 Kinetics Rule Language

2.2.2 (Reaction) RuleML

RuleML [2] 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 [3].

With *RuleML* being already a large specification, *Reaction RuleML* [9] extends *RuleML* towards reaction rules and complex event/action messages, e.g. for complex event processing (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, reasoning). Messages define 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, a sublanguage of *Reaction RuleML* and allows the description of public rule functions as interfaces to hide program logic.

As research continued in terms of reaction rules and *Rule Responder*, the authors of [13] showed the adoption of event paradigms to support scientific workflow execution. In their work they point out the limitation of ECA frameworks when adopted to their use case. For highly distributed and loosley 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 systems with a highly developed rule language that subsumes research from several fields.

2.2.3 JSON Rules

2.2.4 OO jDrew

2.3 Towards ECA Mashups

In [6], the founders of JSON Rules [4] describe a lightweight architecture that allows to react and proact on behalf of events in the ontology of web browsers.

3 Use Case Study

In order to verify some of the identified related work, use cases around the succcessor of useKit [11] (ProBinder [10]) have been derived and investigated.

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 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 Appendix A –) and provided the important insight that the realization of such a use case in an ECA is a time-consuming challange.

3.2 Web Watcher

4 Conclusion

What did we look at How is the evolving of the web possible with these approaches how are we going to make it reactive where does the reactiveness happen interchangeable events important, rules not so much? we only need events communicated and the communiation between them doesn't happen via a clumsy bus. mashups? flexibility and agility of the solution granted by interchangable rules, do we really need this? are we not more going towards a system which consumes events and has user defined rules. there the rules come into play and of course it would be nice to have them in a Reaction RuleML style. let users write rules in RuleMl in a first stage, then simplify the vocabulariy as good as possible. instead of weaving stubs or proxies of existing service into a message oriented middleware (MoM), the web itself is used as the MoM. Through this a lightweighted and performant event-based architecture can be realized, which allows the orchestration of existing web and cloud applications.

a choreography is when an agent forms a collaboration with another one in order to fulfill the task.

5 Future Work

graphical representation of the vocabulary to define rules.

References

[1] Erik Behrends, Oliver Fritzen, Wolfgang May, and Franz Schenk. Embedding Event Algebras and Process for ECA Rules for the Semantic Web. *Fundam. Inf.*, 82(3):237–263, August 2008.

[2] H. Boley. The Rule-ML Family of Web Rule Languages, 2006.

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

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

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

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

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

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

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

[10] S. Rizzotti. ProBinder - Your secure online teamwork platform. `https://probinder.com`. Accessed: 2013-07-07.

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

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

[13] 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);
  }
}
```