A Rule-Based Approach of Creating and Executing Mashups

Emilian Pascalau[1] and Adrian Giurca[2]

[1] Hasso Plattner Institute, Germany
emilian.pascalau@hpi.uni-potsdam.de
[2] Brandenburg University of Technology, Germany
giurca@tu-cottbus.de

Abstract. This paper shows how business rules and particularly how JSON Rules can be used to model mashups together with underlining the advantages of this solution compared to traditional techniques. To achieve this, a concrete use case combining Monster Job Search and Google Maps is developed. In addition, we study the similarities between the conceptual models of *mashup* and *Software as Service* and argue towards a common sense by using their common root: the services choreography.

Keywords. Mashup, Software as Service, Web 2.0 applications, Business rules, JSON Rules

1 Introduction

A common perception regarding Future Internet is that of an Internet of Services and Internet of Things. As described by SAP co-CEO Henning Kagermann [1], *"The Internet of Services is largely based on a service-oriented architecture (SOA), which is a flexible, standardized architecture that facilitates the combination of various applications into inter-operable services. The Internet of Services also uses semantic tools technologies that understand the meaning of information and facilitate the accessibility of content (video, audio, print). Thus, data from various sources and different formats can easily be combined and processed toward a wealth of innovative Web-based services."*

In this context several major paradigms get mixed: Software as a Service (SaaS) [2], Web 2.0 [3], and Enterprise 2.0 [4].

We are witnessing a continuous growth in interest for the services that embraces different shapes such as SaaS, mashups, combined with Semantic Web technologies, all of them towards improving interactivity and collaboration on the Web.

According to Gartner's study Market Trends: Software as a Service, Worldwide, 2007-2012 the worldwide market for software as a service will grow from 4.25 billion in 2006 to 13.02 billions in 2011. Major players such as Salesforce and Google already experiment and provide services that meet the requirements of SaaS applications. They provide several APIs and interconnectivity technologies

either between their services or between other services (such as Facebook for example).

Despite open APIs, another perspective of services aggregation on the Web is offered by mashups. In its early stage this concept has been seen more like a tool approach, but in the last time it started to receive attention also from the academia's side (see for example, [5], [6], [7]). Looking to both SaaS and mashups paradigms we can see a common root: both of them deal with the same major concept - *service aggregation*.

This paper strives to bring to a common sense several paradigms that have already been enumerated here, in particular SaaS and Mashups. We introduce a rule modeling and execution approach to build up mashups. Rules already attract the interest of some players such as Google (i.e. in Google Spreadsheet the user is allowed to change colors using rules) or Adobe (interested in emulating a rule parser in Adobe Flex) simply because, using rules, offers to both mashup creators and mashup users the ability to dynamically change according to their taste their experience on the Web.

In the area of rule modeling there are different developer communities like UML modelers and ontology architects. The former uses rules in business modeling and in software development, while the latter uses rules in collaborative Web applications. The main reason is that rules can be easily captured and modified, and they empower applications with grater flexibility. Therefore, using rules to model and to execute mashups seems to be an appealing solution that could: (1) offer another solution for service aggregation (the main focus is on SOAP/REST based services) and (2) provide a simple way to understand, model and define behavior/interaction between services.

In overall, this work proposes a rule-based approach of modeling and creating mashups. This approach uses JSON Rules, a JavaScript-based rule language and rule engine. This language was introduced in one of our previous work ([8]) with the goal to empower Web 2.0 applications with rule-based inference capabilities.

2 Introduction to JSON Rules

JSON Rules language [8] was built by following two requirements: (1) The "Working Memory" is the Document Object Model (DOM) [9] of the page i.e. the main effect of rules execution is the DOM update, and (2) Rules are executed in the browser. The reason of these requirements is that the content displayed in a web page, besides multimedia content, is mainly a DOM tree. Therefore the main constructs of the language are strongly influenced by this particular environment where the rules are going to be executed.

The language uses a condition language similar with other rule systems (for example Drools, [10]) and employs any JavaScript function call as actions. The syntax was influenced by the JSON Notation [11] a well known notation to express JavaScript objects. In addition to the classical production systems, JSON Rules deals with *Event-Condition-Action (ECA) rules* triggered by DOM Events [12].

A condensed version of the rule metamodel is depicted in Figure 1.

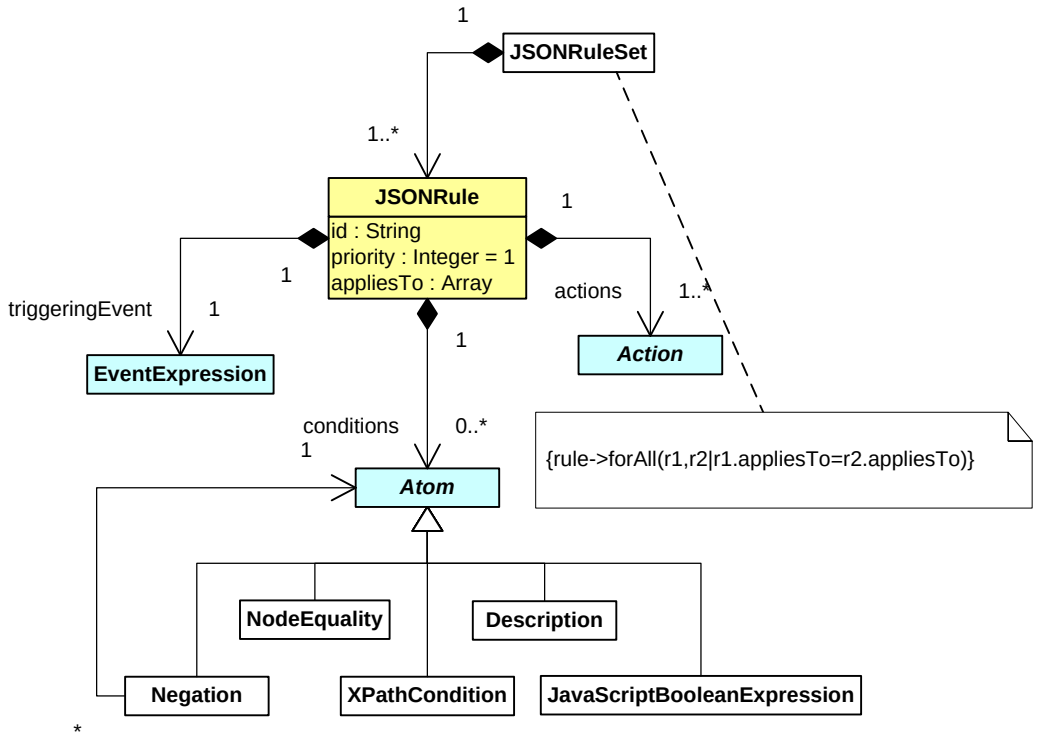

Fig. 1. An excerpt of JSON Rules language metamodel

In brief we identify the following properties, many of them found in all classical production systems:

- A JSON Rule is uniquely identified by an `id`.
- A `priority` optional attribute is used to express the order in which rules get ordered in the activation queue (`1` by default). The execution order for rules with the same priority is irrelevant.
- The required `appliesTo` attribute holds a list of URL's on which the rule can be executed.
- An EventExpression is used to match any DOM Event. Therefore it contains a number of standard DOM Events properties such as: `type`, `target`, `timeStamp` and `phaseType`.
- The rule may contain a list of conditions (logically interpreted as a conjunction of atoms). There are four types of conditions that can be actually expressed using JSON Rules:

(1) `JavaScriptBooleanCondition` - the simplest atom allowing JavaScript boolean, (2) `Descriptions` - a simplified version of Drools description pattern, (3) `==` - DOM Node equality, (4) `XPathCondition` - evaluates the membership of a DOM Node against the nodelist obtained by evaluating an XPath [13] expression and (5) `not` - negation of conditions. Further extensions of the language may envision other types of conditionals.

– The rule must contain a nonempty array of actions, all of them being executed sequentially. Any JavaScript function call is allowed as an action. If the function is not available, no call is performed.

3 Creating Mashups with JSON Rules

There are several concrete use cases on which this approach could be illustrated. We picked a simple one interesting and practical in the same time. In a world were speed is one of the most important decision factors and were businesses changes also at a great speed, people are required to relocate all the time, and change jobs, especially for those working in the IT field. There are several important aspects that usually are taken into account when looking for job, especially if you have family and kids including where is the job located - and you would like to see that on a map. Based on the job location you would like to see the quality of family environment, for example, if there are any schools/universities in the area and their quality facts. The neighborhood is also important and you may want to see some photos in the area.

There are many helpful public services such as Monster Job Search Service, Google Maps, Wikipedia or Flicker which can be used to reach the goal.

One may use these services in a separate way collecting the data he needs. However this process is time consuming and not always with out difficulties, because you have to run between several open tabs, you have to remember and manually insert again and again data from different services. A nice approach would be to have all these services *interacting each other* in the same page.

A programmer may choose one of the available mashup editors but, in this case, he will not be able to run the mashup on an arbitrary server. Usually they run on the tools provider's server. Using JSON Rules, this shortcoming is removed since the engine can be packed in different ways: either as a stand alone JavaScript application that can be imported and used in any page or as a browser add-on. In addition, the service interaction is difficult to be expressed using mashup editors and ends into large and complex code to be written.

For simplicity our example mashup will only use the Monster Job Search Engine and Google Maps.

Lets consider the following case:

> We are looking for a job using the Monster Job Search Service. Once the job is obtained the location is shown on a Google Maps and if it is possible, some supplementary information is provided.

The following requirements were considered:

– All these services must interact in one page. This page is called *choreographer*.
– The search term must be inserted manually in a form invoking the Monster Job Search Service. The other services must react on the returned data.
– When the mouse is over a job, the job should be visually indicated on Google Maps.
– When the mouse is over a job, the information regarding the job, if any, should be retrieved from another service (such as Wikipedia).
– Involved services should be personalized.

To be able to define rules that will power our use case we must know how involved services look like, how they can be interrogated and how does their response look like.

3.1 Monster Job Search

The Monster start page provides a number of components that may not be necessary in our mashup, as, for example the `Sign Up!/Sign In` component.

Figure 2 shows an excerpt of the HTML output of the Monster search and it is necessary to understand the rules related to the usage of Monster inside of our mashup.

3.2 Google Maps

The maps service from Google is well known probably to everyone reading this paper, so we will present here only the DOM tree view (Figure 3) of the search field which is necessary to understand the rules that will implement the search.

3.3 Modeling the Mashup Rules

Having presented the input and output of the involved services we can now present the rules that power up our mashup. To be able to define rules that will make the several involved services work together, they must be available to the `Choreographer`.

Rule: *Load services so that the Choreographer can use them*

```
{ "id":"loadServices",
   "appliesTo":["http://www.jsonrules.org/examples/i3e/"],
  "eventExpression":{
                    "type":"load"
                    },
  "condition":true,
  "actions":["load('http://jobsearch.monster.ca')",
             "load('http://maps.google.com')"]
 }
```

```
<div class="jobSearchResultDiv" id="jobSearchresult">
  <div id="sortOptions" class="sortOptions">
    ...
  </div>
  <div class="stackedView">
    <div class="stackedRowPurple">
      <div class="jobInfo" style="width: 673px;">
        <div class="stackedViewJobViewLink"
          id="stackedViewJobViewLink0">
          <div id="joblink_0" class="joblink">
            <a onclick="jobViewOnClickSaveCookie(0);"
              href="..." onmouseover="ctlMouseOverRender(0);"
              id="jobviewlink_0" class="joblinks">
              Technical Solution Architect - IT
            </a>
          </div>
          </div>
        <div class="stackedViewWidth1" style="width: 224.333px;">
            <div class="stackedViewCompanyLogo"
              id="stackedViewCompanyLogo0"/>
        </div>
        <div class="stackedViewWidth2" style="width: 224.333px;">
            <div class="stackedViewCompany">
              Lockheed Martin Canada
            </div>
            <div class="stackedViewDate">
              Posted:March 23
            </div>
        </div>
        <div class="stackedViewWidth3" style="width: 224.333px;">
            <div class="stackedViewJobPlace">
              <div class="jobPlace">
                Ottawa, ON
              </div>
            </div>
            <div class="stackedViewMiles">
              <div class="distanceTextMsg">
                Distance:
              </div>
            </div>
        </div>
      </div>
      <div class="jobIcons" style="width: 92px;">
        ...
      </div>
    </div>
  ...
  </div>
</div>
```

Fig. 2. Monster Job Search Service - excerpt of search output

```
<form id="q_form" action="/maps" ...>
  <div class="srchcol controls">
    <input type="text"
      value="" autocomplete="off"
      maxlength="2048" tabindex="1"
      title="Search the map" name="q" id="q_d"
      style="width: 33em;"/>
    ....
  </div>
</form>
```

Fig. 3. Google Maps - search field

The above rule applies to the choreographer URL and states that whenever a DOM event of type `load` occurs then two `load` action are executed. The side effect is the loading of the services we need in the mashup.

Filtering the DOM to eliminate undesired elements can be easily performed by using `XPathConditions`. For example, the below rule identifies the `Sign Up!/Sign In` component of Monster by using the XPath expression: `/html/body/form/div[3]/div[2]/div` and remove all elements:

Rule : *Remove `Sign Up!/Sign In` component rule*

```
{"id":"ruleDeleteMonsterLoginComponent",
  "appliesTo": ["http://www.jsonrules.org/examples/i3e/",
                "http://jobsearch.monster.ca/"],
  "eventExpression":{"type":"load"},
  "condition": ["$X in '/html/body/form/div[3]/div[2]/div'"],
  "actions":["document.removeChild($X)"]
}
```

On event of type `load`, remove all nodes returned by evaluating the `XPathCondition`.

Having data received from Monster (as in Figure 2) we find the results in a specific `div` element (having `id="jobSearchresult"` and `class="jobSearchResultDiv"`). Although such a `div` has several children we are particulary interested in those `div` children having `class="jobInfo"`. The children of this `div` provide us with all the information needed further for the Google Maps service.

To find out the location of a job identified as stated above we must retrieve the company name and the location and provide this information as input value for the Google Maps input field (See Figure 3). In addition, our requirements impose that the location of a job should be displayed when a `mouseover` event occurs on the element containing the specific job.

Rule: *Find a job location on a Google Map*

```
{"id":"findJobLocation",
```

```
  "appliesTo": ["http://www.jsonrules.org/examples/i3e/",
                "http://jobsearch.monster.ca/"],
  "eventExpression":{"type":"mouseover",
                     "target":"$X"
                    },
  "condition":
        [
         "$X:HTMLElement(
             tagName=='div',
             className=='jobInfo'
             )",
         "$Y in 'child::$X'",
         "$Y:HTMLElement(
             tagName=='div',
             className=='stackedViewCompany'
            )",
         "$Z in 'child::$X'",
         "$Z:HTMLElement(
             tagName=='div',
             className=='jobPlace'
            )",
         "not($Y==$Z)",
         "$T:HTMLElement(
             tagName=='input',
             id=='q_d'
            )",
         "$companyName == $Y.nodeValue",
         "$jobLocation == $Z.nodeValue",
         "$form:HTMLElement(
            tagName=='form',
            id=='q_form'
            )"
         ],
  "actions":
         [
          "update($T,'nodeValue',
                  '$companyName+' '+$jobLocation')",
          "autoSubmitForm($form)"
         ]
}
```

The `findJobLocation` rule is triggered by a `mouseover` event. However as already stated the required information needed to be able to find the location of the job using Google Maps is provided by the children of a `div` element having `class='jobInfo'`. In accordance with this the rule verifies if the `mouseover` event has been raised from a `div` element having `class='jobInfo'` and among

the children of this particular `div` element there are other different `div` elements having `class='stackedViewCompany'` (bounded to `$Y` variable) and `class='jobPlace'` (bounded to `$Z` variable). If the DOM contains an `input` element having `id='q_d'` and a `form` element having `id='q_form'` validates the availability of the Google Maps Service. To be able to search the location for the current job the employer name is needed and it's location. This information is bound to `$companyName` and `$jobLocation` variables. If all the above conditions hold then the `update` and `autoSubmitForm` actions can be executed. The `update` action performs an update of the `nodeValue` property of the element `$T` with the value (`$companyName+' '+$jobLocation`). The `autoSubmitForm` action performs an automatic submission of the form provided as parameter.

4 Towards a Common Sense for Mashup and Software as Service

[14] analyzes terms such as *software as a service*, *software on demand*, *adaptive enterprise* and *mashups* and concludes that they are overlapping to many extents. This section tries to argue towards a common sense, to create a bridge between *software as a service* and *mashups*.

It is well known that in the nowadays business environment there is a strong need and a general request of being capable to change software easily to meet the fast evolving business requirements.

As stated in [2], "*the term software as a service is beginning to gain acceptance in the market-place; however the notion of service-based software extends beyond these emerging concepts*".

In such an approach a service conforms with the much accepted definition stating that a service is "*an act or performance offered by one party to another. Although the process may be tied to a physical product, the performance is essentially intangible and does not normally result in ownership of any of the factors of production*" ([15]).

In addition, [2] argued that the "*service-based model of software is one in which services are configured to meet a specific set of requirements at a point in time*". Components may be bound instantly, based on the needs and discarded afterwards.

There are some key points that characterize *software as a service* (see for example, [16]):

- network-based access and management of commercially available software
- activities managed from central locations rather than at each customer's site,
- enabling customers to access applications remotely via the Web
- application delivery typically closer to a *one-to-many model* (single instance, multi-tenant architecture) rather than to a *one-to-one model*, including architecture, pricing, partnering, and management characteristics
- centralized feature updating, which obviates the need for end-users to download patches and upgrades

– frequent integration into a larger network of communicating software - either as part of a mashup or as a plugin to a platform as a service.

Mashups are hybrid web applications, usually found out under the association of SOA plus REST principles. Mash-up content is usually accessed through APIs from third party providers, (sites, services), processed and then presented to the user in a different format and with new insights. In such way new value is provided. One simple way to explain mashups was introduced by ZDNet Executive Editor David Berlind in a video presentation, What is a mashup?, where, among other issues, he claims that mashups are the fastest growing ecosystem on the Web.

Berlind introduced the mashup model by comparing it with the well known software stack on the traditional computers. In traditional computer systems we have an operating system, and, on top of this, a number of application programming interfaces (APIs) to access different services (i.e. the network, the display, the file system) and UIs to get to different applications (i.e. the mouse, the keyboard) as in Figure 4. Developers use these APIs to access different necessary services to create their applications.

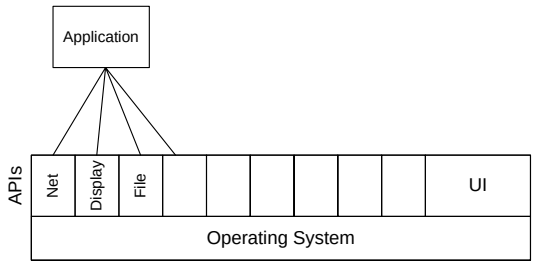

Fig. 4. Computer Model (David Berlind, ZDNet)

Somehow mashups follow the same model but with a different infrastructure. The Operating System is replaced by the Internet, and the old APIs are replaced with APIs offered by different service providers such as Yahoo, Google, Technorati, Amazon etc. as depicted in the Figure 5. In the same way developers use these APIs to get access the available services. These services reside in the Internet, or we may also say on top of the infrastructure offered by the Internet. In this way new applications are created from old ones.

The *software as a service* approach uses a very similar model (see Figure 6). Salesforce is a concrete example of this approach. However in this case the Operating System/Internet is represented by the Platform as a Service (i.e. Force.com).

Therefore, we see that while mashups are generally based on various service sources available on the Web, software as service is mainly based on a proprietary

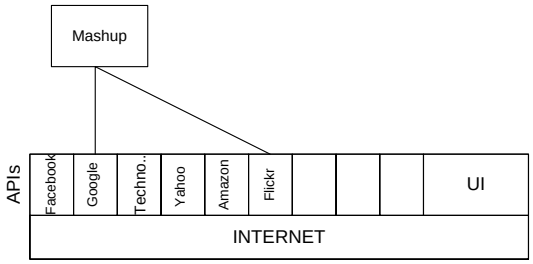

Fig. 5. Mashup Model, (David Berlind, ZDNet)

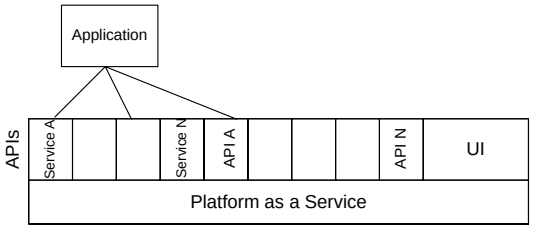

Fig. 6. SaaS Model

centric platform (which may handle different services). In SaaS, new applications are generated by using only the platform services (i.e. *platform as a service*).

Due to their *"open"* character, mashups besides the characteristics that define SaaS applications have some other characteristics that arise from the way they are implemented. Although some of these characteristics might overlap we present them here:

- Mashups are usually created with a mashup editor such as: Google Mashup Editor, JackBe,Lotus Mashups, Microsoft Popfly, Mozilla Ubiquity, Yahoo Pipes.
- Mashups use APIs from different platforms to aggregate and reuse the content.
- Usually mashups operate on XML based content such as Atom [17], [18], RSS 2.0 [19], and RDF [20], sometimes directly on the HTML level, strictly for presentation
- "Melting Pot" style such that content is aggregated arbitrarily
- Create, read, update and delete (CRUD) operations are preferred to be based on REST principles

However, an important restriction of actual mashup editors is that they allow users to build and run mashups on specific platforms.

The rule based approach presented here removes the specific platform level and works directly on the content that any user has access to through a Web

browser. In this way a simple model of mashups can be imagined as in the Figure 7.

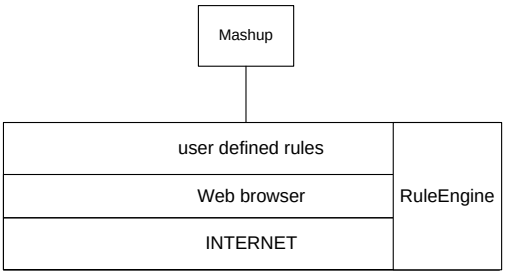

Fig. 7. Our Model

Because DOM and JavaScript are the fundamental assets that power the content that we see every day in browsers, it is natural to use an approach that uses exactly the same assets because in this way you get finer granularity, and avoid learning of different APIs.

There are several features that the presented approach offers:

- Mashups can be executed on any browser allowing JavaScript.
- Services can be accessed directly without intermediary parties such as APIs.
- Flows can be defined on top of any component that is available in the service answer (i.e. any DOM entity can be used)
- The behavior is defined declaratively.
- Data can be accessed as usual (Atom, RSS , RDF) but also in raw format.
- Concerns that are very actual, regarding creation of UI's based on the service are overcome because the UI provided as default by the server in the form of a web page can be used, and modified as desired; Look and aspect can be managed with rules too.
- Data mapping can be very easy implemented (such as data from Monster to be submitted to the Google Maps service)

Recall that in our use case there is a service called Choreographer since it corresponds to the application that puts together data from different services and defines the way they interact. According to [21] process choreography is used to define cooperation between process orchestrations. Moreover this collaboration is specified by collaboration rules. With respect to our approach, rule-based modeling and execution of mashups are nothing else but collaboration rules and the mashup itself is a *browser-based service choreography.*

5 Conclusions and Future work

We have presented how business rules and in particular JSON Rules can be used to model mashups together with underlining the advantages of this solution

compared to traditional techniques. To achieve this a concrete use has been presented together with the rules modeling it.

In addition, we studied the similarities between the conceptual models of SaaS and Mashups and observed that both of them have a common root: service choreography.

Future work concerns the study of how choreography principles can be related to JSON rule modeling of mashups towards a methodology of creation and maintenance of rule-based mashups. Another topic is related to application of visual modeling techniques for rules as well as building an infrastructure allowing sharing and reusing.

Acknowledgements

We want to express our gratitude to Prof. Mathias Weske who shared his ideas with us and gave us his time, comments and valuable insights on SaaS and choreographies issues.

References

1. Kagermann, H.: Toward a European Strategy for the Future Internet A Call for Action. White paper, SAP AG (2008) `http://www.sap.com/about/company/research/fields/internet_services/index.epx`.
2. Bennett, K., Layzell, P., Budgen, D., Brereton, P., Macaulay, L., Munro, M.: Service-Based Software: The Future for Flexible Software. In: Proceedings of the Seventh Asia-Pacific Software Engineering Conference (APSEC2000), IEEE Computer Society (2000) 214–221 `http://www.bds.ie/Pdf/ServiceOriented1.pdf`.
3. O'Reilly, T.: What is web 2.0. design patterns and business models for the next generation of software. Oreillynet.com (September 2005) `http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html`.
4. McAfee, A.P.: Enterprise 2.0: The Dawn of Emergent Collaboration. MIT Sloan Management Review **47**(3) (2006) 21–28
5. Abiteboul, S., Greenshpan, O., Milo, T.: Modeling the mashup space. In: WIDM'08: Proceeding of the 10th ACM workshop on Web information and data management. (2008) 87–94
6. Jarrar, M., Dikaiakos, M.D.: Mashql: a query-by-diagram topping sparql. In: ONISW '08: Proceeding of the 2nd international workshop on Ontologies and nformation systems for the semantic web, New York, NY, USA, ACM (2008) 89–96
7. Phuoc, D.L., Polleres, A., Morbidoni, C., , Hauswirth, M., Tummarello, G.: Rapid semantic web mashup development through semantic web pipes. In: Proceedings of the 18th World Wide Web Conference (WWW2009). (April 2009) `http://pipes.deri.org/attachments/004_fp160-lephuoc.pdf`.
8. Giurca, A., Pascalau, E.: JSON Rules. In: Proceedings of the Proceedings of 4th Knowledge Engineering and Software Engineering, KESE 2008, collocated with KI 2008. Volume 425., CEUR Workshop Proceedings (2008) 7–18
9. Hors, A.L., Hegaret, P.L., Wood, L., Nicol, G., Robie, J., Champion, M., Byrne, S.: Document Object Model (DOM) Level 3 Core Specification. W3C Recommendation (April 2004) `http://www.w3.org/TR/DOM-Level-3-Core/`.

10. Proctor, M., Neale, M., Frandsen, M., Jr., S.G., Tirelli, E., Meyer, F., Verlaenen, K.: Drools 4.0.7. http://downloads.jboss.com/drools/docs/4.0.7.19894.GA/html_single/index.html (May 2008)
11. Crockford, D.: The application/json Media Type for JavaScript Object Notation (JSON). http://tools.ietf.org/html/rfc4627 (July 2006)
12. Pixley, T.: Document Object Model (DOM) Level 2 Events Specification. W3C Recommendation (November 2000) http://www.w3.org/TR/DOM-Level-2-Events/.
13. Berglund, A., Boag, S., Chamberlin, D., Fernandez, M.F., Kay, M., Robie, J., Simeon, J.: XML Path Language (XPath) 2.0. W3C Recommendation (November 2007) http://www.w3.org/TR/xpath20/.
14. Foster, I., Tuecke, S.: Describing the Elephant: The Different Faces of IT as Service. Enterprise Distributed Computing **3**(6) (July/August 2005) 26–34
15. Lovelock, C., Vandermerwe, S., Lewis, B.: Services Marketing . Prentice Hall Europe (1996)
16. Traudt, E., Konary, A.: Software as a Service Taxonomy and Research Guide. Technical report, IDC.com (2005)
17. Nottingham, M., Sayre, R.: Atom Publishing Format (RFC4287). http://tools.ietf.org/html/rfc4287 (2005)
18. Gregorio, J., de hOra, B.: Atom Publishing Format (RFC5023). http://tools.ietf.org/html/rfc5023 (2007)
19. RSS: RSS 2.0 Specification, version 2.0.11. http://www.rssboard.org/rss-specification (March 2009)
20. Klyne, G., Caroll, J.: Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation (February 2004) http://www.w3.org/TR/rdf-concepts/.
21. Weske, M.: Business Process Management: Concepts, Languages, Architectures . Springer-Verlag Berlin Heidelberg (2007)