Facilitating the Reactive Web
A Condition Action System using Node.js

Alexander Gröflin[1], Dominic Bosch[1], Helmar Burkhart[1] and Martin Guggisberg[1]
[1]*High Performance and Web Computing Group, University of Basel, Spiegelgasse 1, CH-4051 Basel, Switzerland*
{*alexander.groeflin, dominic.bosch, helmar.burkhart, martin.guggisberg*}*@unibas.ch*

Keywords: Reactive Web, Condition Action System, Event Aggregation, Node.js.

Abstract: The orchestration of the Web is a big issue for Web users all around the world. Web users have a high interest in services, which are able to personalise and customise the Web. However, for Web reactivity there exists only a few limited solutions that allow the aggregation of Web resources. This paper takes a look at existing event-based methods that build upon Event-Condition-Action (ECA) Rules and Complex Event Processing (CEP). Moreover, this paper illustrates the architecture of a fully functioning Condition Action System prototype for the creation of reactivity in between Web resources. In a proof of concept, we could detect and determine the change interval of electronic newspaper headlines. With the proposed system, we are able to orchestrate Web resources e.g. Detecting Web Changes.

1 INTRODUCTION

The invention of the Web is without doubt one of the most influencing technological developments with respect to economy, education and social life. While the first decade of Web existence is characterised by the proliferation of home and business websites including web-shops, the last decade transformed the Web to a manifold information service. Service-Oriented Architectures (SOA) and Web services have been proposed for quite a time. While business services turned out to be successful from the economic side, the Web still has unused potentials in terms of programmable automation, customisation and personalisation.

Web users find themselves mashing up data and functionality from different Web resources and services manually, e.g. a press release provokes a Twitter post. Consequently, Web users use the Web without full automation. Manual use of services always implies a delay of reaction; real-time orchestration of services is limited. Great value would be added for Web users if specific interaction had been automatically achieved e.g. detecting and reacting on certain updates. This would require an instrument for the identification and filtering of changes, the adoption of time constraints, composition and placement of the outcome in the users preferred Web resource or service (Windley, 2012).

To achieve this reactivity, users need Mashup platforms and tools with which they are able to automate tedious tasks and to react on certain events in a predefined way. There have been several steps undertaken to create such tools and platforms in order to tackle basic automation (Akbar et al., 2014). For example, IFTTT (https://ifttt.com/) or Zapier (https://zapier.com) focus on a one to one connection between Web services at the same time e.g. weather change triggers a Twitter tweet. A shortcoming of these platforms is that events, which have occurred in several distributed systems, cannot be detected as part of one rule. It is certainly not possible to create custom event listeners and actions. For additional customisation, these service providers have to set up such a service connection for each Web resource themselves. Furthermore, selecting more actions on one single trigger is not possible. They rather create a tunnel from one Web service to another, than creating reactivity in between several resources.

A variety of languages for Web service orchestration has been reported in the last years, e.g. (Dijkman et al., 2008), (Wohed et al., 2003). Most of these languages address workflows, business collaborations and long running interactions for well-defined processes. Our goal was to create a Condition Action System with focus on real-time reaction, which allows inbuilt coding for the reactivity between Web resources (Blackstock and Lea, 2014).

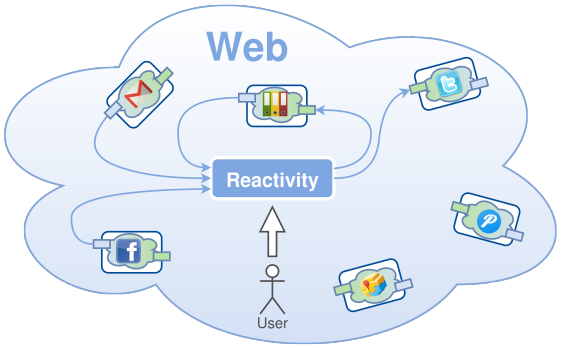

Figure 1: Proposed scheme of a system in which a user aggregates Web resources. Observed changes in these resources form events that are processed in a reactivity entity. The output is an action that control Web resources. Personalised settings allow a user-specific orchestration of Web resources e.g. Facebook, Google Mail, and Twitter etc.

In this paper, we assess event-based technologies to create a reactive system, which is able to orchestrate Web resources. In addition, we demonstrate the architecture of a prototype system, show web technologies used, and address the issue how to create and to detect events for a more efficient Web experience. Our approach takes the direction towards the reactive Web. In Section 2, we outline the need for an orchestration of Web resources and describe the Event-Condition-Action (ECA) paradigm. Section 3 describes a particular use case. In Section 4, we present the architecture of our Condition Action System prototype, its functionalities and a proof of concept. Conclusively, Section 6 specifies a conclusion.

2 AGGREGATION OF WEB RESOURCES

In recent years, many mashup tools and platforms appeared on the Web that enable Web users to create reactive behaviour. One of the earliest attempts is Yahoo! Pipes, which defines itself as a data mashup tool (Yahoo!, 2010). It aggregates content from different RSS feeds. Desired data are selected from different feeds; the output also consists of a RSS feed or data in JavaScript Object Notation (JSON). This also means that the field of application is also very limited because it uses RSS as input. Real mashup tools for example IFTTT or Zapier focus on directing the data flow from one Web service to another. By orchestrating Web APIs real reactivity is being created (Ovadia, 2014). Such an approach opens up the possibilities of automation in between different Web services. However, the key problem with this conception is that they do not offer generic access to any arbitrary Web resource. Unfortunately, not every Web resource has an accessible interface in these tools. This means that these platforms have to provide the interface to the desired Web services. Another problem of Web mashups like IFTTT is the restricted programmability. IFTTT does not allow Boolean operators, which hinders solving expressive problems. Sometimes it would be useful to react on more complex rules than the simple If This Then That scheme. Concluding, these services rather interconnect existing services and remove the need for any coding (Blackstock and Lea, 2014).

2.1 Event-Condition-Action (ECA) Paradigm

Increasing amount of literature emerges on reactivity related to the Web (Paschke, 2014), (Hausmann and Bry, 2013), (Paschke et al., 2012). The rationale behind these studies is an event-based system, which in turn rely on ECA Rules. ECA Rules consist of three different parts:

- Event: An identifier which detects events
- Condition: An expression which determines whether an action should be triggered or not
- Action: A set of instructions which describe the reactive behaviour

Event-based characteristics allow users to make use of loosely coupled Web services, which significantly improves scalability in information exchange and distributed workflows. It semantically decouples space, time and synchronisation between event producer and event consumer (Hasan et al., 2012). Thus, the resulting goal is to remove *"explicit dependencies between the interacting participants"* (Eugster et al., 2003). Simple events occur at a single point in time (e.g. a website update). Event-based systems may react on simple events, but this is often not enough to detect meaningful situations among different Web resources. A composition of events reflects a complex event, which is multi-layered and has a distinct duration. It would enable Web users to detect meaningful situations consisting of simple events and reacting on them.

2.2 Retrieving Events Using Webhooks

Webhooks are services, which report state changes in a push manner (Trifa et al., 2010). Thus, they allow real-time propagation of events. Existing manifestations of Webhooks are available for server to browser communication, such as Comet (Duquennoy et al., 2009) or Server-Sent Events

(http://dev.w3.org/html5/eventsource/). The instant delivery of information whenever it gets available makes this notion valuable for real-time handling.

In principle, Webhooks consist of an URI, which points to a Web resource and a sender. It forwards data as soon as it gets available within the publish/subscribe pattern (Eugster et al., 2003). Whenever an event occurs, new data is delivered via a Webhook, which passes it along to a specific URI. Therefore, Webhooks use Web services as callbacks on remote Web APIs (Benslimane et al., 2008) e.g. delivering event data to predefined Web resources. An open server-to-server publish/subscribe protocol is PubSubHubbub (https://code.google.com/p/pubsubhubbub/), which uses Webhooks to announce updates from other servers with interesting content. Through push notifications and instant event propagation it is possible to create a reactive real-time system.

2.3 Evolutionary Event Aggregation

Complex Event Processing (CEP) is rather important for event aggregation (Anicic et al., 2010) and may be used for event compositions. CEP focuses on predefined relations and temporal patterns, typically on different streams of data or events. CEP processes complex event patterns; in other words, it may derive complex events from simple events. However, we believe in an asynchrony event aggregation together with ECA Rules. This enables processing large amounts of data and assembling compositions in real-time.

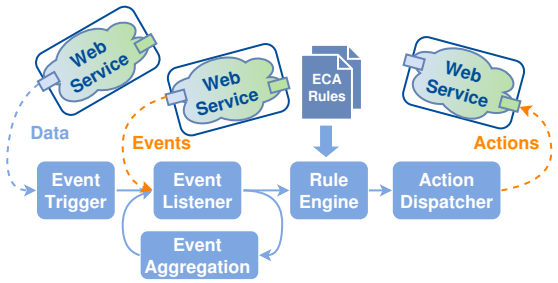

Figure 2: Condition Action System connects to Web services. Web services deliver data to the Event Trigger or events to the Event Listener. Subsequently, the Rule Engine processes events according to predefined ECA Rules and instructs the Action Dispatcher for further actions.

After all, ECA Rules may help the Rule Engine to detect aggregated data changes in order to dispatch predefined actions. The last piece in our scheme is the Action Dispatcher, which is responsible for the binding of data flow and actions between heterogeneous Web resources. An Action Dispatcher allows flexible coupling with Web Resources, similar to the Event Trigger module. Event Trigger and Action Dispatcher are abstractions of the communication flow in between Web services and resources.

3 CONDITION ACTION SYSTEM IMPLEMENTATION

We have developed a Condition Action System prototype, which fulfils our architectural model for a reactive Web. Since Web service communication is latency-driven, we assumed asynchronous communication and scalability mechanisms to be core features for our prototype system. Another key aspect is the programmability of Events, which means that JavaScript code can be programmed and ran in the system straightaway (See 3.2 Web Change Detection Rule).

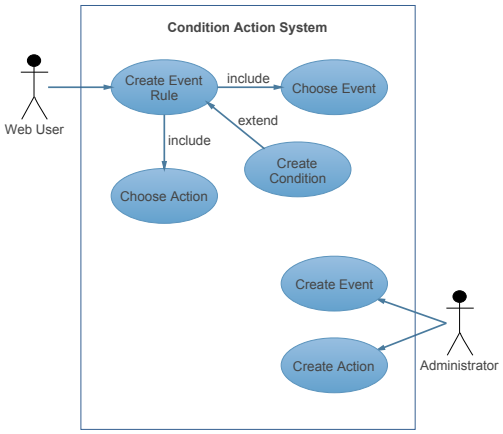

Figure 3: Web users create Event Rules in order to define the desired reaction out of all available Events. Specific Conditions and Actions are programmable during the creation of an Event Rule. Administrators may either create (program) Events or Actions.

3.1 Detection Interval

New information confronts us all over the Web. Unfortunately, we waste a lot of time until we realise that these updates have happened. The challenge remains to be the fastest to know about these updates. Manual capturing of this knowledge implies a massive delay in reaction and it is certainly not in real-time. Great value would be added for academics and businesses but also for Web users if desired information from any source had been automatically retrieved within seconds of a change. Detecting and reacting to Web changes would require an instrument for the identification, the capturing, the aggregation and the placement of knowledge in the users preferred way.

Sometimes Web users become aware of knowledge within seconds. More frequently, new information stays undetected for several hours when not days e.g. a deadline extension. To capture such events our Condition Action System provides reaction mechanisms regarding time.

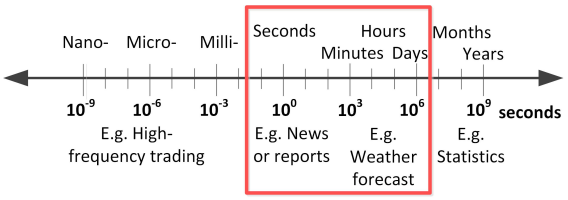

Figure 4: The Condition Action System focuses on reaction times within the red boundaries. An interval time must be set for the detection of events and is limited to seconds, minutes, hours or days. Different areas are in scope of the system e.g. reports, weather forecast and press releases.

3.2 Web Change Detection Rule

Many documents in the Web are dynamically modified over time. Some might change in an interval of a few minutes e.g. reports on news portals (minutes), while other content changes much slower e.g. entries on encyclopaedia sites such as Wikipedia (days). To detect these changes, a Condition Action System must keep track of the document for example with the help of a history. This would allow to compare Web resources and in case of a specific change, it would trigger an action.

Consequently, Web users have to set up a detection rule, which checks a website (URI) for changes. The idea is to detect for example a paper submission deadline for a conference. On a date change on the website, the system sends the new submission date to an email address.

The rule has to use an Event Trigger, we name it DetectWebsiteChange, which actually identifies changes on a defined website. The rule only encompasses the URI and the HTML tag e.g. an id or a class. If the Condition Action System detects a change, a mail will be sent to the provided email address. Below the Web change detection rule is represented in the JSON format:

```
{
  "id": "Web Detection Rule",
  "event": {
    "DetectWebsiteChange->byId": {
   "uri": "http://webist.org",
   "id": "DOM element"
 }
  },
  "conditions": [],
  "actions": {
    "OnWebsiteChange->writemail": {
      "receipient": "mail@address.com",
      "text": "Deadline has changed!"
 }
  }
}
```

The JSON rule code requests the Event Trigger DetectWebsiteChange to look for changes on the given URI and selects the HTML element with the given id attribute. As soon as a change in this particular HTML element is detected, it is pushed forward as an event. In this case, conditions and actions are taking over. If for example the action write mail is correctly configured for this rule, a mail will be sent with the changed deadline. For the means of customisation, actions can also access data from events.

3.3 Condition Action System Architecture

The Condition Action System prototype consists of five modules (see Figure 5):

- Poller: Loads Event Trigger modules and forwards their emitted events to the Event Queue. Event Trigger modules poll for changes in the Web and transform them into events.
- Event Listener: Listens on local URIs of active Webhooks for events and forwards them to the Event Queue.
- Event Queue: Acts as an event buffer.
- Rules Engine: Processes events from the Event Queue whenever they get available.
- User Request Handler: The user interfaces to administrate Event Triggers, Webhooks, Rules and Action Dispatchers.

These modules deal with event and action objects. An event object has a time and source attribute while an action has a target URI and additional configuration parameters. Condition objects consist of a logical statement with on event based parameters.

On start-up, the Condition Action System loads all stored rules and for each rule it loads all related Action Dispatchers. The system also notifies the Poller in case of a new rule, which in turn loads an Event Trigger. Subsequently, the Event Listener loads all stored Webhooks and starts to listen for new events. At this stage, the Condition Action System is up and running and accepts administration requests for Event Triggers, Webhooks, Rules and Action Dispatchers. If an event occurs, data is forwarded to the Event Queue. Whenever a rule is met, the Poller and the Rule Engine start up the required Event Triggers

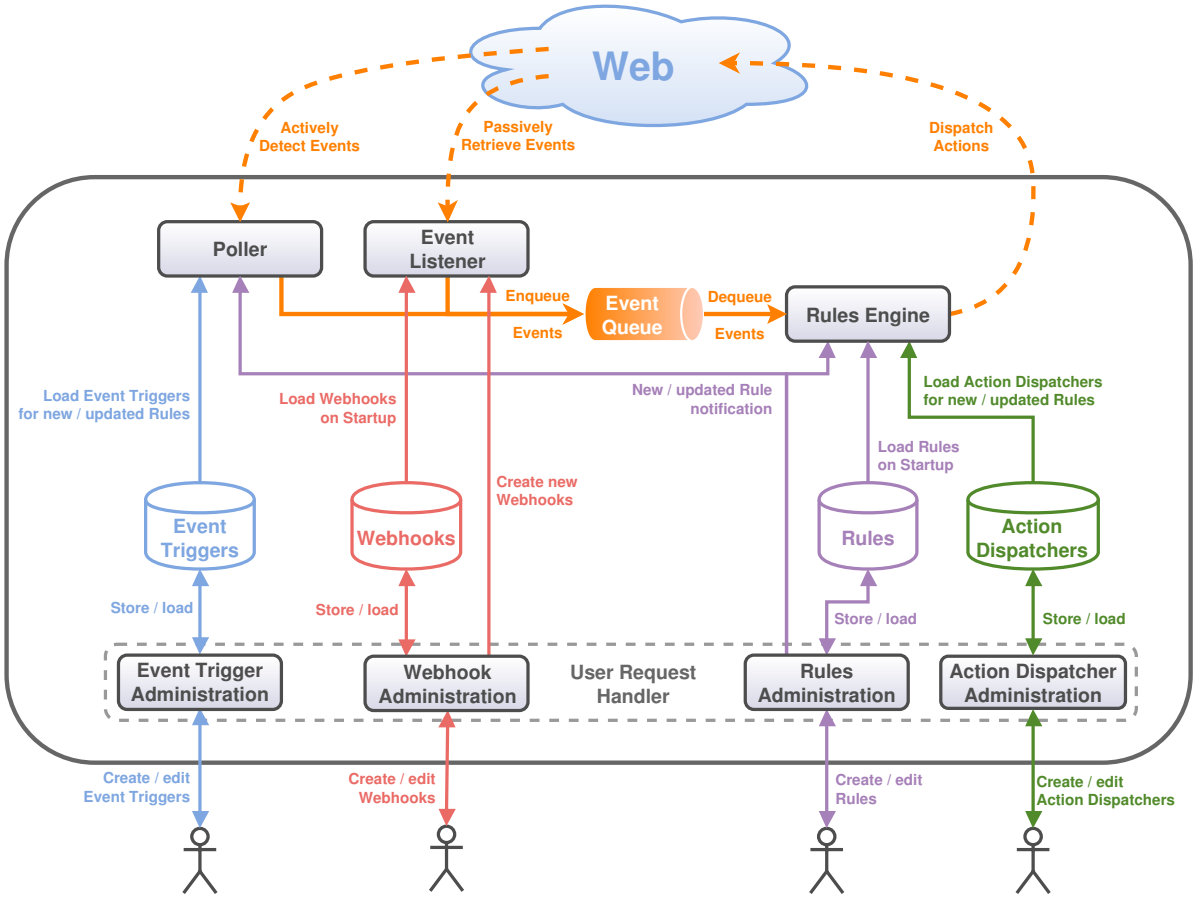

Figure 5: Architecture of the Condition Action System. In the upper part of the figure, the core functionalities of the Condition Action System are shown. Poller and Event Listener forward events to the Event Queue. The Rules Engine evaluates events one after the other. In the middle of the figure, the personalised configuration is persisted in databases. Configuration of the whole system is manageable via a user interface, which is shown in the lower part.

or Action Dispatchers. The Event Queue processes all incoming events in which ECA Rules may be applied.

3.4 Technologies Used

XML and JSON are standard formats for information exchange between Web services. Both formats describe data in tree structures, which our Condition Action System can use. Furthermore, it is a communication format, which the system uses for the structure of events. The event-driven architecture was built upon the recent adoption of server side JavaScript through Node.js and its both human and machine-readable JSON communication format.

Three platforms for the prototype system were tested. Early testing instances were on a 64-bit, i5 2.3 GHz dual-core system with 3GB of DDR3 SDRAM and 100GB of storage capacity running with Linux 3.2 kernel, Node.js version 0.8.24 and the in-memory key-value store database redis 2.7.105.

One instance was running on a virtual machine with 64-bit Intel Xeon 2 GHz single-core CPU, 1GB RAM and 120GB disk space as well as Node.js version 0.8.24 and redis 2.7.105. The prototype system is running on Amazon Elastic Compute Cloud Instance with a single-core 64-bit CPU, 1GB of memory and 8GB of attached storage, which runs on a Linux 3.2 kernel, Node.js version 0.10.2 and redis 2.8.7.

As shown in the architecture schema below, the prototype system is a centralised solution. Thanks to the asynchronous communication paradigm, an implementation of several parallel running systems in the amazon cloud will be a minor next step and bring scalability for a larger use.

A list of important node.js libraries used for the prototype is given here:

- Cheerio: Parsing of a webpage and access to its structure, similar to jQuery.

- CoffeeScript: Domain specific programming language used as a transcompiler to JavaScript.
- Express: Eased running of a server instance in terms of request mapping to handler functions and folder structures.
- Gulp: Stream-based workflow automation.
- Import.io: Webpage querying over their API with helps of personalised masked generated via their own browser.
- JS-select: Selectors for data nodes in tree structures.
- Request: Enables communication between servers.

3.5 Proof of Concept

One key problem with Web resources is that users (customers) cannot verify the quality of these services. With the help of our prototype system, a measurement can be realised. Such a system may check whether the website is reachable and may pay attention to relevant changes. As a result, a reactive system creates a reliable source for the quality of Web applications and services e.g. uptime and downtime.

The prototype system has been used to test various scenarios. A first proof of concept has focused on two Swiss electronic newspapers. Over a period of one week, the Condition Action System collected data from their websites in order to observe the frequency of headline changes. What is interesting in this particular data set is that the headline of the classical newspaper, which is available on the Web and as well on ordinary paper, changed on average after two to six hours. The other news website, which has a web presence only, tends to change the headline on average within two hours and therefore in a higher frequency. In addition, we were also able to determine a webserver failure of one news portal, which lead to a 20-minute long service disruption. This example outlines the potentials of the web change detection scenario (see Section 3.2).

Many documents in the Web are dynamically modified over time. Some might change in an interval of a few minutes e.g. news on news portals, while other content changes much slower e.g. entries on encyclopaedia sites such as Wikipedia. To detect these changes, our prototype system must keep track of the document history. An Event Listener monitors Web resources and as soon as differences are detected, an action is triggered. Consequently, the user is able to set up a rule, which checks whether changes are related to a certain category of interest or not. Moreover, it could highlight certain areas of interest or another specified website and compare how significant these changes are. If for example a Wikipedia article has been changed in more than 10% of its original content, a moderator should be informed to revise the article. Thus, the system is able to directly inform the moderator via mail.

4 CONCLUSION

Existing Condition Action Systems are limited to Web services and discourage Web users from programming own program code. Our goal was to create a Condition Action System with both options; inbuilt programming window and already existing Event Rules. However, the system has qualified for future performance tests against existing Condition Action Systems.

Novelties have their greatest value if they are perceived immediately, in the best case in real-time. Reactivity stands for the focal point in instantly notifying users or creating something new out of changes in the Web e.g. a tweet on Twitter. Such a Reactivity entity in the Web may help users in orchestrating such behaviour.

This paper has addressed promises of a Condition Action System that can be valuable for Web users in orchestrating tedious tasks of Web services. Based on one scenario, the system architecture has been assessed in detecting Web changes; the system is running for more than half a year. Moreover, the proposed architecture of the Condition Action System is very powerful for the measurement of all kinds of Web resources. Difficulties arise, however, when an attempt is made to store data by using the architecture. Event Triggers, Webhooks, Rules and Action Dispatchers are stored in a database, the changes on the Web, which create an event are not. It might be interesting in future to store data of these changes in order to use it for further data mining.

REFERENCES

Akbar, Z., Garca, J., Toma, I., and Fensel, D. (2014). *On Using Semantically-Aware Rules for Efficient Online Communication*, volume 8620 of *Lecture Notes in Computer Science*. Springer International Publishing.

Anicic, D., Fodor, P., Rudolph, S., Sthmer, R., Stojanovic, N., and Studer, R. (2010). *A Rule-Based Language for Complex Event Processing and Reasoning*, volume 6333 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg.

Benslimane, D., Dustdar, S., and Sheth, A. (2008). Services

mashups: The new generation of web applications. *Internet Computing, IEEE*, 12(5):13–15.

Blackstock, M. and Lea, R. (2014). Towards a distributed data flow platform for the web of things. *5th International Workshop on the Web of Things, 2014*, pages 1–6.

Dijkman, R. M., Dumas, M., and Ouyang, C. (2008). Semantics and analysis of business process models in bpmn. *Information and Software Technology*, 50(12):1281 – 1294.

Duquennoy, S., Grimaud, G., and Vandewalle, J.-J. (2009). Consistency and scalability in event notification for embedded web applications. In *Web Systems Evolution (WSE), 2009 11th IEEE International Symposium on*, pages 89–98.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131.

Hasan, S., O'Riain, S., and Curry, E. (2012). Approximate semantic matching of heterogeneous events. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 252–263, New York, NY, USA. ACM.

Hausmann, S. and Bry, F. (2013). Towards complex actions for complex event processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 135–146, New York, NY, USA. ACM.

Ovadia, S. (2014). Automate the internet with if this then that (ifttt). *Behavioral & Social Sciences Librarian*, 33(4):208–211.

Paschke, A. (2014). Reaction ruleml 1.0 for rules, events and actions in semantic complex event processing. In *Rules on the Web. From Theory to Applications*, volume 8620 of *Lecture Notes in Computer Science*, pages 1–21. Springer International Publishing.

Paschke, A., Boley, H., Zhao, Z., Teymourian, K., and Athan, T. (2012). Reaction ruleml 1.0: Standardized semantic reaction rules. In *Rules on the Web: Research and Applications*, volume 7438 of *Lecture Notes in Computer Science*, pages 100–119. Springer Berlin Heidelberg.

Trifa, V., Guinard, D., Davidovski, V., Kamilaris, A., and Delchev, I. (2010). Web messaging for open and scalable distributed sensing applications. In *Proceedings of the 10th International Conference on Web Engineering*, ICWE'10, pages 129–143, Berlin, Heidelberg. Springer-Verlag.

Windley, P. (2012). *The Live Web: Building Event-Based Connections in the Cloud*. Course Technology.

Wohed, P., van der Aalst, W., Dumas, M., and ter Hofstede, A. (2003). Analysis of web services composition languages: The case of bpel4ws. In *Conceptual Modeling - ER 2003*, volume 2813 of *Lecture Notes in Computer Science*, pages 200–215. Springer Berlin Heidelberg.

Yahoo! (2010). Pipes [www document], http://pipes.yahoo.com/pipes/docs?doc=overview (accessed 18.12.14).