Twelve Theses on Reactive Rules for the Web

François Bry and Michael Eckert

University of Munich, Institute for Informatics
Oettingenstr. 67, D-80538 München
{bry, eckert}@pms.ifi.lmu.de
http://www.pms.ifi.lmu.de

Abstract. Reactivity, the ability to detect and react to events, is an essential functionality in many information systems. In particular, Web systems such as online marketplaces, adaptive (e.g., recommender) systems, and Web services, react to events such as Web page updates or data posted to a server.
This article investigates issues of relevance in designing high-level programming languages dedicated to reactivity on the Web. It presents twelve theses on features desirable for a language of reactive rules tuned to programming Web and Semantic Web applications.

1 Introduction

A common perception of the Web is that of a distributed repository of hypermedia documents with clients (in general browsers) that download documents, and servers that store and update documents. Although reflecting a widespread use of the Web, this perception is not completely accurate.

In fact, many Web applications build upon servers or clients updating data in reaction to events or to messages exchanged on the Web. Examples are online marketplaces, adaptive (e.g., recommender) systems, and Web services. The Web's communication protocol, HTTP, provides an infrastructure for exchanging events or messages. In addition, SOAP provides conventions for exchanging structured and typed information on the Web as XML messages. For transport of messages between Web nodes, SOAP can use HTTP (or other protocols).

This article first argues that complementing HTTP and SOAP with high-level languages for updates and reactivity is needed for both standard Web and Semantic Web applications. It then presents twelve theses on features desirable for a language of reactive rules tuned to programming Web applications.

After providing motivation and background (Section 2), this article successively addresses the need for ECA rules on the Web (Section 3), paradigms for communicating events between Web sites (Section 4), specifying composite events (Section 5), specifying conditions as Web queries (Section 6), specifying state-changing actions (Section 7), structuring constructs for rules and programs (Section 8), and additional issues (Section 9). The views reported about in this article have emerged during the design of the Web and Semantic Web query language Xcerpt [1, 2] and of the reactive language XChange [3, 4] as well as from experiences with programming in Xcerpt and XChange [5, 6].

2 Motivation and Background

Updates on the Web Many Web applications build upon servers that update data according to client requests or actions. This is the case in online marketplaces that receive and process orders, e-learning systems that select and deliver teaching materials depending on a student's test performances, recommender systems that select goods or services depending on a customer's previous orders or expressed preferences, and communication platforms such as Wikis, where several users modify the same documents. Conversely, some Web applications also build upon clients that update data according to server requests: a server can request a client to store information in a cookie. Typically a cookie is used to store information such as user identification or the contents of a user's shopping basket on the client-side. The server can then later retrieve this information, thus freeing the client from reentering this information.

Reactivity on the Web Many Web applications not only build upon the updating of data, but also upon complex reactions to messages or events exchanged not only between clients and servers but also (via servers) between clients. This is the case when contributors to a Web-based communication platform are informed of other contributors joining or leaving a session. It is the case for Web-based business management systems, e.g., for business travel applications, planning, and reimbursement in large companies, that rely upon complex workflows of actions and messages, possibly realized using Web services. It is also the case for Web-based systems offering context-dependent services, e.g., a time- and location-dependent car park directory that adapts the information it delivers and reacts to changes.

Updates and Reactivity on the Semantic Web Updates and reactivity are as much a Semantic Web issue as they are a standard Web issues. The application scenarios stressed above might involve both standard Web and Semantic Web data and techniques, such as HTML, XML, RDF, Topic Maps, and OWL data, as well as inference from RDF triples. For example, e-commerce offers might be described by RDF meta-data and an e-learning system might refer to inference rules expressed in terms of RDF triples, RDF Schema, and OWL.

Infrastructure The basis for updates and reactivity is that Web sites inform each other about update requests and events by exchanging messages. The Hypertext Transfer Protocol (HTTP) [7], the Web's communication protocol, allows Web sites to send data to each other. The important commands are GET, which is primarily used to retrieve data identified by a URI, and POST, which is primarily used to send data to some Web resource (again identified by a URI).[1]

[1] Data can also be sent from the client to the server using GET. Though against the original philosophy of HTTP (GET should not have side-effects), this use is quite common.

A framework for message exchange on the Web is given by SOAP [8].[2] SOAP's main components are (1) message envelope and (2) transport binding. The envelope defines an XML format for representing message content and processing information. It consist of the header, which provides information about the message (e.g., date when sent), and the body, which carries application-dependent data (the "payload"). The transport binding specifies the method used to communicate message from one node to the other via some lower-level protocol such as HTTP or SMTP (Simple Mail Transfer Protocol [9]).

High-Level Languages Cost-efficient development of Web applications such as those mentioned above, requires high-level languages tailored to updates and reactivity. Although HTTP and SOAP help implement updates and reactivity on the Web, more abstract and higher-level languages are needed that

- abstract away network communication and system issues,
- ease the specification of complex updates of Web resources (e.g., XML, RDF, and OWL data),
- are convenient for specifying complex flows of actions and reactions on the Web.

The need for high-level Web update and reactivity languages is similar to the need for high-level (Semantic) Web query languages (see [10] for a survey). High-level reactive languages will complement, not replace, HTTP and SOAP.

3 The Need for ECA Rules on the Web

Thesis 1: **High-level reactive languages are needed on the Web. Event-Condition-Action rules are well-suited to specify reactivity on the Web. In particular, they are better suited than production rules for a large class of Web applications.**

High-level reactive languages are needed on the Web. Programming reactive behavior using the primitives of the Hypertext Transfer Protocol, HTTP, (such as POST and GET) is rather cumbersome and distracts from the principal task. Efforts like the Common Gateway Interface (CGI) or Application Programming Interfaces to access Web Services and exchange SOAP messages such as the Java API for XML Web Services (JAX-WS) seek to overcome this burden of low-level programming. These efforts are based on general purpose programming languages not specifically tailored for the Web and for reactivity.

Rules are very convenient for a high-level expression of reactive behavior. This is amply demonstrated by the intensive use of reactive, also called "dynamic," rules among other kinds of "business rules."

[2] "SOAP" was originally an acronym for "Simple Object Access Protocol," but this name has been dropped with SOAP 1.2 as the concern is rather object interoperability than object access.

A rule-based approach to reactivity on the Web provides the following benefits over the conventional approach using (imperative or object-oriented) general purpose programming languages:

- Rules are easy to understand for humans. Requirements specification often already comes in the form of rules expressed either in a natural or formal language.
- Rule-based specifications are flexible, therefore easy to adapt, alter, and maintain as requirements change, which is quite frequently the case with business rules.
- Rules are well-suited for processing and analyzing by machines. Methods for automatic optimization, verification, and transformation into other types of rules (e.g., derive ECA rules from integrity constraints) have been well-studied and applied successfully in the past.
- Rules can be managed in a single rule base as well as in several rule bases possibly distributed over the Web.

Common reactive rules are production rules, also called Condition-Action (CA) rules, and Event-Condition-Action (ECA) rules.[3] Production rules have the form "if *condition* do *action*" and specify to execute the *action* automatically when the *condition* (typically expressed as a query to data in some fact base) becomes true (in general due to previous actions). In contrast, ECA rules have the form "on *event* if *condition* do *action*" and specify to execute the *action* automatically when the *event* happens, provided the *condition* holds.

On the Web, reactive rules explicitely referring to events, i.e., Event-Condition-Action (ECA) rules, are more appropriate than production rules without explicit reference to events for the following reasons:

- "Real-world reactive rules" often come with an explicit specification of an event, for example: "a credit card application (event) will be granted (action) if the applicant has a monthly income of more than EUR 1 500 and no outstanding debts (condition)."
- Events exchanged as messages between Web nodes are a natural, high-level communication paradigm, also exploited in Service-Oriented and Event-Driven Architecture.
- Events can carry data between Web nodes that is relevant for the condition and action part of a rule.
- ECA rules allow an easy handling of errors and exceptional situations that can conveniently be expressed as (special) events.

Finally, in situations where production rules are more appropriate, it is often possible to derive ECA rules automatically or semi-automatically from production rules[4] and provide an efficient implementation mechanism this way.

[3] In some works, the term production rules is used to mean both ECA and CA rules. We use it here in the more customary sense meaning only CA rules.

[4] It is tempting to claim that the production rule "`if` C `do` A" can be trivially expressed as the ECA rule "`on` *true* `if` C `do` A", where *true* matches any event. In

Thesis 2: **Reactive Web rules should be processed locally and act globally through event-based communication and access to persistent Web data.**

Reactive Web rules should be processed locally at each Web site. In particular, each Web site manages its own rule base and determines locally which of the rules fire. Approaches assuming a central rule processing entity are not suitable for the Web's highly distributed and loosely coupled architecture. Even distributed approaches that assume collaboration and cooperation between different Web sites for rule processing are too tightly coupled for the Web.

Observe that requiring rules to run locally does not make any restrictions on their source: rule interchange formats such as the one developed at W3C [11] allow a Web site to incorporate rules from documents anywhere on the Web or receive rules as messages from other Web sites. An example where rule exchange by messages is necessary will be given in Thesis 11.

Global behavior can be achieved by using event-based communication. Local reactive rules can generate new events which are sent to remote Web sites, where they in turn trigger other (remote) reactive rules. This also enables Web sites to coordinate and synchronize activities in a choreography-based manner without a central coordinator. Further, reactive rules should be able to access data from anywhere on the Web, for example, query a (remote) RDF document identified by its URI. (See Thesis 4 for the distinction of event data and persistent Web data.)

4 Event Communication Paradigms

Thesis 3: **Events are best exchanged directly between Web sites in a push manner.**

Because of the Web's decentralized architecture, events must be exchanged directly, point-to-point, between Web sites without the involvement of central servers, super-peers, and the like.

Events are best sent in a push manner from the Web sites where the event happens or originates to other, interested Web sites. Periodical polling, where interested Web sites retrieve remote Web resources periodically to check if an event has happened, is less favorable, since it causes more network traffic, increases reaction time, and requires more local resources.

general, this is however not the case: the production rule fires *only once*, when the condition becomes true. The ECA rules fires *every time* an event happens, as long as the condition holds. Only with further assumptions, such as that the action is idempotent (executing it a second time has no further effects, e.g., insertion into a set) and not undone by some other action while the condition holds (e.g., once the insertion of an item is made, it will not be deleted), the above production rule and ECA rule are equivalent.

5 Specifying Composite Events: Towards High-Level Event Query Languages

Thesis 4: **Events are volatile data and should be kept distinct from persistent data.**

On a reactive Web, there are two kinds of data: "normal" data from Web resources such as XML or RDF documents ("persistent data") and data from events ("volatile data"). "Normal" Web data is retrieved upon request in a pull manner, *persistent*, and can be *modified*. It typically signifies a *state* of (an abstraction of) the world. Event data is communicated between Web nodes (typically in a push manner), *volatile*, and *not modifiable*. It is typically used to signal *changes in state*.

The distinction of persistent data and volatile data can be illustrated with a metaphor. Persistent data is like (computer-)*written text*. Once produced, it is available permanently for anyone to read (or rather anyone who is allowed to do so). Later, the text can be modified directly by editing it. Volatile data is like *spoken words*. Once a sentence is spoken, its information is available only to the listeners and only as long as they remember. A spoken sentence cannot be changed; the only way to correct, complete, or invalidate its information then is through speaking new sentences.

Due to their different nature, there should be a clean separation of persistent Web data and volatile event data in a reactive language. It should be ensured that volatile data stays volatile, i.e., is disposed of after finite time. This avoids growing storage requirements for event data. If some data from an event must be stored indefinitely, it should explicitly be made persistent in a Web resource. Most importantly, not having a clean separation of Web data and event data could lead to a "shadow Web," a hidden collection of (then non-volatile) event data that lives in parallel to the normal Web with its persistent Web resources.

Thesis 5: **Recognizing composite events is essential for a reactive Web language. Composite events are conveniently specified by (event) queries. There are (at least) four complementary dimensions to event queries: data extraction, event composition, temporal conditions, and event accumulation.**

Often, a situation that requires a reaction cannot be detected from a single incoming event (called *atomic event* henceforth). For example, the cancellation of a flight (atomic event) might not by itself require a reaction by a passenger. However, if a flight has been canceled, and there is no notification within the next two hours that the passenger is put onto another flight, this might well require a reaction.

Such situations are called *composite events* (as opposed to single atomic events), and they are especially important on the Web: In a carefully developed application, atomic events might suffice as designers have the freedom to choose events according to their goal. On the Web, however, many different applications are integrated and have to cooperate. Situations which have not been considered in an application's design must then be inferred from several atomic events.

Composite events as patterns of events do not exists explicitly "by themselves" in the stream of incoming atomic events. Rather they are implicit and the patterns are conveniently specified by event queries.

There are at least the following complementary four dimensions that need to be considered for an event query language:

- Data extraction: Event messages contain data that is relevant to whether and how to react. The data must be provided (typically as bindings for variables) to the condition and action part of an ECA rule.
- Event composition: To support composite events, event queries must support composition constructs such as the conjunction, disjunction, and negation of events (or more precisely of event queries).
- Temporal conditions: Time plays an important role in many reactive Web applications. Event queries must be able to express temporal conditions such as "events A and B happen within 1 hour and A happens before B."
- Event accumulation: Event queries must be able to accumulate events of the same type to aggregate data or detect repetitions. For example, a stock market application might require notification if "the average over the last 5 reported stock prices raises by 5%," or a service level agreement might require a reaction when "3 server outages have been reported within 1 hour."

Some applications might also require features not mentioned above such as event instance selection (choose only one out of several available answers to an event query) or event instance consumption ("use-up" atomic events so they are not available for generating future answers) [12].

Thesis 6: **A data-driven, incremental evaluation of event queries is the approach of choice.**

A data-driven evaluation of (composite) event queries is the best-suited approach. It can work incrementally and is thus preferable for efficiency reasons: work done in one evaluation step of an event query should not be redone in future evaluation. For example, the composite event query "events A and B happen" requires to check every incoming event if it is A or B and thus multiple evaluation steps. When event A is detected, we want to remember this for later when B is detected to signal the composite event. In contrast, a non-incremental, query-driven (backward-chaining) evaluation would have to check the entire history of events for an A when a B is detected.

6 Specifying Conditions: Embedding a Web Query Language

Thesis 7: **Data from persistent Web resources plays an essential role for Web reactivity. A reactive language thus should embed or build upon a Web query language.**

A reactive Web language has to integrate in the current Web of retrievable, persistent data sources. Programmers must be able to easily access and query persistent Web data. Querying XML, RDF, and other Web data is well-studied and a multitude of query languages has been devised. Instead of reinventing the wheel, a reactive language should thus embed or build upon an existing Web query language.

Data from Web resources is usually persistent and reflects a state (see Thesis 4). The natural place to deploy a Web query language in ECA rules is thus the condition part. However, the Web query language should also be used to query data in atomic events in the event part of ECA rules — after all, the same data models (XML, RDF, etc.) are used. This gives language coherency [13] and makes learning the new reactive language much faster. The language coherency can be even further increased if an update language based on the query language is available for the action part.

Criteria to be considered for a Web query language include:

– What is the query language's notion of answers (variable bindings, newly constructed data)?
– How are answers delivered, can they be used to "parameterize" further queries or the action? Can, for example, a variable bound in an event query be a parameter in a condition query, i.e., the value delivered by the event query be accessed and used in the condition query?
– What evaluation methods for queries are possible (backward chaining, forward chaining)?
– Which data models are supported (XML, RDF, OWL)? Is it possible to access data in different data models within one query?
– How does the query language deal with identity (see Thesis 9)?
– Which reasoning or deductive capabilites does the query language provide (views, deductive rules; see also Thesis 8)?

The choice of a Web query language has significant influence on the design of a reactive language and should thus be made carefully.

7 Specifying State-Changing Actions

Thesis 8: **The Web is a dynamic, state-changing system. Reactions to state changes (events) through reactive rules are state-changing actions such as updates to persistent data. Reactive rules are needed where compound actions can be constructed from primitive actions.**

The Web is a dynamic, state-changing system. To act in this world, reactive rules need the ability to perform state-changing actions such as updates to persistent data. The drawback that this makes reactive languages less declarative than pure (side-effect free) logic or functional programming languages is inherent to the task and should not withhold efforts to make reactive languages as declarative as possible.

The most important actions are updating persistent data on the Web and communicating with other Web sites (through raising new events). Complex reactions can conveniently be built as compounds of primitive actions such as insertions, deletions, or modifications of XML elements, RDF triples, or OWL facts. The most common compound is a sequence of actions, but other compounds such as a the specification of alternative actions are needed, too.

8 Structuring Rules and Rule Programs

Thesis 9: **Development and maintenance of reactive rule programs can be considerably supported by structuring mechanisms such as: branching in rules, deductive rules for event queries and Web queries, procedural abstractions for actions, and grouping of rules.**

Like in any other programming language, development and maintenance of reactive rule programs can be considerably supported by a language's structuring mechanisms. In particular, we propose that reactive rule languages have the following structuring mechanisms:

- Branching in rules: it is more convenient to write one rule "`on` E `if` C `do` A_1 `else` A_2" than writing two rules "`on` E `if` C `do` A_1" and "`on` E `if` $\neg C$ `do` A_2". Rules of this kind are sometimes called ECAA rules (since they specify an action and an alternative action), and there are also more general forms such as $\mathrm{EC}^n\mathrm{A}^n$ rules [14], which specify several condition-action pairs. Rules of this kind are not only more convenient to write, but also are easier to maintain because replication (of C in this example) is avoided. Avoiding replication is also good for execution efficiency: the condition C is only tested once in an ECAA rule.[5]
- Deductive rules for event queries and Web queries: deductive rules can be compared to views in relational databases and their advantages for Web data (XML documents, RDF sources, etc.) are well understood. They avoid replication of complicated queries, allow to derive intensional data from extensional data, and can be used to mediate data in different schemas. The same advantages apply for querying and reasoning with event data, and we propose to also have deductive rules for events. However, since event queries have to evaluated very frequently, a reactive language can be made more restrictive about rules for events for efficency reasons (e.g., reject recursive rules).
- Procedural abstractions for actions: often several rules will share the same action. For example, an electronic shop will have rules for different forms of payment, all having the same reaction (e.g., ship item). The reaction can be rather complicated and composed of many smaller actions (e.g., update the customer database and the warehouse database, e-mail the customer the expected date of delivery and the tracking number). A procedure mechanism, where the action is specified once and given a name, is clearly a better approach than writing the same code in several rules.

[5] Testing C only once for two ECA rules with C and $\neg C$ is of course possible, but requires optimization techniques detecting and exploiting similarities in rules.

- Grouping of rules: a flat, unstructured set or list of rules offers no guidance where a certain functionality is to be found and which rules interact. Virtually all wide-spread programming languages offer modules, packages, or similar constructs to structure programs. Grouping rules into separate, named rule sets and possibly also building hierarchies of rule sets exposes the structure of a rule program and eases considerably human understanding. Also, rule sets could introduce scopes for identifiers, alleviating the danger of unwanted interaction of rules due to name-clashes of identifiers.

These structuring mechanisms aim primarily at avoiding redundancy, i.e., write code needed in several places only once, and at exposing the organization of a program, i.e., keep related pieces code together and unrelated code separate in the program layout. This not only eases authoring and maintenance for human programmers; it is also good for the execution of the rules on a machine: recognizing redundancy becomes less important for query optimization and division of programs allows to execute smaller units.

9 Miscellanea

Thesis 10: **Identity of data items is an issue for reactive languages due to their ability to react to changes of data objects on the Web.**

Reactive languages with the ability to monitor data items (or objects) and react to their changes need to deal with identity of the data items. Consider monitoring a news Web site for updates to a particular article: for this task, it is necessary to (uniquely) identify the article of interest.

There are basically two approaches to identity: extensional identity and surrogate identity.

Extensional identity defines identity based on an object's structure or value (its extension). Objects which are equal in structure and value are thus treated as identical. Examples of this are relational databases (forgetting multiset semantics for the moment), logic programming, and functional programming. When an object's value changes, it looses its identity. To alleviate this, objects are often given an auxiliary attribute such as a primary key having a unique value for each object. Of course, with a change of the value of the key, identity is lost.

Surrogate identity (also called object identity) defines identity independent from an object's extension as an external surrogate (e.g., the object's address in memory). It is thus possible for objects to be non-identical, even though they have the same structure and value, leading to a distinction of identity and equality. Examples of this are object-oriented databases and object-oriented programming languages. An important advantage of surrogate identity is that it allows an object to keep its identity when its value changes.

For monitoring changes of objects, surrogate identity is advantageous. However, to communicate with remote Web sites, surrogate identity has to become part of the data, i.e., made extensional. Even worse, Web resources such as XML or RDF documents usually do not provide a surrogate identity for their data at all and only rarely provide auxiliary identity-defining attributes (keys such as `xml:id` attributes) as part of the extension.

Thesis 11: **Meta-programming and meta-circularity, that is, the ability to use rules to exchange and evaluate (other) rules, are needed in some important cases.**

Certain important reactive Web applications require a mutual exchange and evaluation of rules. An example are (automatic) policy-based trust negotiations [15]. Consider the following scenario of online-shopping. Customer and (electronic) shop do not know (or trust) each other in advance and are thus sensitive about giving out certain information (e.g., credit card number) or committing to certain actions (e.g., shipping without prior payment). Using policy-based negotiation they establish a basis of trust sufficient to make a deal.

1. Customer Franz requests to buy ten soccer balls from `fussbaelle.biz`, an online shop which he has found with a Web search and not heard of before.
2. In reply, the shop sends its policy governing sales and payment, that is, a set of rules describing, e.g., what identification the customer has to provide and which methods of payment (credit card, check, money transfer, etc.) are acceptable under what conditions.
3. Franz determines that paying by credit card satisfies the shop's policy as well as his own preferences. However, he is not willing to provide sensitive information such as his credit card number to some untrusted shop. Instead he sends back to the shop a policy stating conditions under which he is willing to disclose it.
4. `fussbaelle.biz` evaluates the customer's policy, determines that its membership in the Better Business Bureau of Internet satisfies the customer's conditions, and sends its membership certificate.
5. Franz checks the certificate, reveals his credit card information, and closes the deal.

Observe that in this example, customer and shop do not give out all their policies at once. Instead they exchange policies reactively during the course of the trust negotiation. Which policies are actually exchanged depends on the previous course of action. Such a reactive approach has several advantages: (1) it is more efficient since only small sets of relevant rules are exchanged, (2) policies themselves can be sensitive information and thus only given out when a certain stage in the negotiation (e.g., trust level) has been reached, (3) it allows policies to be determined dynamically (e.g., using game-theoretic approaches in the fashion of [16]).

Realizing the above exchange of rules and rule sets (policies) with reactive rules leads to a requirement for meta-programming or meta-circularity in reactive, rule-based languages. In meta-programming, programs can "have other programs as data and exploit their semantics"[17]. A particular form of meta-programming is meta-circularity, where the same language is used on both levels (i.e., the rules realizing the exchange and the rules being exchanged are written in the same language).

Thesis 12: **Reactivity in the Web's open and uncontrolled world requires language support for authentication, authorization, and accounting.**

The Web is an open, uncontrolled system allowing for anyone to retrieve data from anywhere on the Web in an anonymous way. For reactive Web applications this is usually not acceptable; services such as electronic shops or on-demand computing require controlled access, in particular:

- authentication to establish that users of the service really are who they claim to be,
- authorization to control access to sensitive information or services, and
- accounting to monitor and log service accesses and resource consumption for management, planning, and billing (the latter in particular when pay-per-use pricing in employed).

These "three As" are non-functional requirements;[6] a reactive language should thus come to the programmers aid and provide easy-to-use support for them, so programmer can concentrate on the functional requirements.

We discuss only accounting in more detail since it is maybe the most interesting issue, as far as reactivity is concerned. Authentication and authorization are relatively static issues which can be treated as simple conditions in ECA rules (though they can be negotiated dynamically, see the previous thesis) and also exist for pure information access on the non-reactive Web.

Accounting in contrast is a dynamic issue: it reacts to incoming service request to monitor and log them. This leads to a "double reactivity:" on the one hand there is the reactive service itself, on the other hand the accounting service, which in turn reacts to uses of the reactive service. Note, however, that these are orthogonal axes of reactivity and no meta-programming (see previous thesis) has to be employed: the accounting service does not contain the reactive service or have to reason about its interiors.

10 Conclusion

In this article we have presented twelve theses on reactive rules for the Web. We have argued that reactivity in the Web needs reactive rules, in particular ECA rules, and established a list of desiderata for reactive, ECA-rule-based languages.

The theses reflect our experiences from designing the reactive language XChange [4, 18, 3, 19] and programming in it. An initial design and a prototype implementation of XChange are complete, and we hope for the positions presented in this paper to provide directions in the future development of XChange.

[6] In software engineering, non-functional requirements are constraints on the functionality offered by the system (e.g., performance, security). Functional requirements, in contrast, are statements about the actual functionality of a system (e.g., accepted input, produced output).

Many of the desiderata postulated in this article are very general. They apply not only to reactive languages based on ECA rules, but also to other rule-based reactive languages (e.g., based on production rules) and even languages, frameworks, and program libraries not based on rules at all.

Acknowledgments

The ideas expressed in this article have been significantly influenced by the research project REWERSE (Reasoning on the Web with Rules and Semantics, http://rewerse.net) and the W3C RIF Working Group (http://w3.org/2005/rules). The authors thank their colleagues of REWERSE and of the W3C RIF Working Group for many fruitful exchanges on the subject of this article.

The authors thank Tim Furche, Paula-Lavinia Pătrânjan, and Inna Romanenko for their insights and numerous discussions.

References

1. Schaffert, S., Bry, F.: Querying the Web reconsidered: A practical introduction to Xcerpt. In: Proc. Extreme Markup Languages. (2004)
2. Xcerpt. http://xcerpt.org (2006)
3. Bailey, J., Bry, F., Eckert, M., Pătrânjan, P.L.: Flavours of XChange, a rule-based reactive language for the (Semantic) Web. In: Proc. Intl. Conf. on Rules and Rule Markup Languages for the Semantic Web. Volume 3791 of LNCS, Springer (2005)
4. Bry, F., Eckert, M., Pătrânjan, P.L.: Reactivity on the Web: Paradigms and applications of the language XChange. J. of Web Engineering **5**(1) (2006) 3–24
5. Kraus, S.: Use Cases für Xcerpt: Eine positionelle Anfrage- und Transformationssprache für das Web. Master's thesis (in German), Inst. for Informatics, Univ. of Munich (2004)
6. Romanenko, I.: Use cases for reactivity on the Web: Using ECA rules for business process modeling. Master's thesis, Inst. for Informatics, Univ. of Munich (2006)
7. Fielding, R., et al.: Hypertext transfer protocol – HTTP/1.1. RFC 2616, The Internet Society (1999)
8. Gudgin, M., et al.: SOAP version 1.2. W3C recommendation, World Wide Web Consortium (2003)
9. Klensin, J.: Simple mail transfer protocol. RFC 2821, The Internet Society (2001)
10. Bailey, J., Bry, F., Furche, T., Schaffert, S.: Web and Semantic Web query languages: A survey. In: Reasoning Web, Int. Summer School. Volume 3564 of LNCS, Springer (2005) 35–133
11. World Wide Web Consortium: Rule interchange format working group charter. http://www.w3.org/2005/rules/wg/charter (2005)
12. Zimmer, D., Unland, R.: On the semantics of complex events in active database management systems. In: Proc. Int. Conf. on Data Engineering. (1999)
13. Bry, F., Marchiori, M.: Ten theses on logic languages for the Semantic Web. In: Proc. Int. Workshop on Principles and Practice of Semantic Web Reasoning. Volume 3703 of LNCS, Springer (2005) 42–49

14. Knolmayer, G., Endl, R., Pfahrer, M.: Modeling processes and workflows by business rules. In: Business Process Management, Models, Techniques, and Empirical Studies. Volume 1806 of LNCS, Springer (2000) 16–29
15. Winslett, M.: An introduction to trust negotiation. In: Proc. Int. Conf. on Trust Management (iTrust). Volume 2692 of LNCS, Springer (2003) 275–283
16. Preibusch, S.: Implementing privacy negotiations in e-commerce. In: Proc. Asia-Pacific Web Conference. Volume 3841 of LNCS, Springer (2006) 604–615
17. Apt, K.R., Turini, F., eds.: Meta-Logics and Logic Programming. MIT Press (1995)
18. Bry, F., Eckert, M., Pătrânjan, P.L.: Querying composite events for reactivity on the Web. In: Proc. Int. Workshop on XML Research and Applications (XRA) at APWeb 2006. Volume 3842 of LNCS, Springer (2006) 38–47
19. XChange. http://www.pms.ifi.lmu.de/projekte/xchange (2006)