Gaining Reactivity for Rich Internet Applications by Introducing Client-side Complex Event Processing and Declarative Rules

Kay-Uwe Schmidt
SAP AG, Research
Vincenz-Prießnitz-Straße 1
76131 Karlsruhe
kay-uwe.schmidt@sap.com

Roland Stühmer
FZI Forschungszentrum Informatik
Haid-und-Neu-Straße 10-14
76131 Karlsruhe
roland.stuehmer@fzi.de

Ljiljana Stojanovic
FZI Forschungszentrum Informatik
Haid-und-Neu-Straße 10-14
76131 Karlsruhe
ljiljana.stojanovic@fzi.de

Abstract

Rich Internet Applications provide, in conjunction with Internet push technologies, a powerful framework to bring use cases formerly reserved to server applications to the client, and to ease their use. In this paper we present a novel approach of monitoring and processing event streams directly in the browser. Our proposed general-purpose framework aims at the design of event-driven, reactive and adaptive Rich Internet Applications. We propose to interweave complex event processing with declarative rule execution directly on the client-side. Our work is based on a novel event-condition-action rule language tailored to the needs of Rich Internet Applications as well as algorithms capable of detecting complex events and executing rules. The whole approach will be illustrated by means of an example originating from the field of algorithmic stock trading.

Introduction

Recently, the design paradigm of an event-driven architecture (EDA) gained momentum as there is a need in the service-oriented world to trigger and monitor the execution of services. In parallel AJAX-based Rich Internet Applications (RIAs) appeared on the Web developer scene leading to desktop-like Web applications. As a third programming paradigm declarative ontology- and rule-based systems already proved their effectiveness of making the semantics of business objects and application logic explicit. Taking the best of the three worlds, we propose the enhancement of RIAs by complex event processing[1] (CEP), rule execution and formal semantics in order to deal with challenges like algorithmic trading, ad-hoc workflows, personal and community task management as well as with adaptive user interfaces to name only a few use cases.

State of the art event-driven RIAs rely either on the *Comet* architecture introduced by Alex Russell in 2006 (Russell 2006) or on client polling strategies like RSS feeds. The Comet architecture particularly allows pushing events to the client. Such technologies provide the basic infrastructure for transporting events to the client but they provide no means for processing and reacting to those events. This gap will be closed by our approach.

Our approach proposes *intelligent RIAs* (IRIA) combining complex event processing and rule execution with an ontology-based object model directly on the client. We propose a framework built on meaningful business vocabularies, declarative rules and design patterns for achieving reactive and adaptive RIAs. Our main objective is to show the advantages of such an approach and to show that our approach scales well.

The paper is structured as follows: First we outline related work and give an example in order to motivate our research. In the following section we present the logical system architecture. In the section JSON-Rules we describe our declarative rule language tailored to the needs of RIAs. After that we detail our client-side, event-driven rule engine. A performance evaluation is given in the Evaluation section and the paper sums up with conclusions and future work.

Related Work

Although the idea of combining CEP, production rules and formal business vocabulary in RIAs is new, we built our approach on already existing work. Possibly the earliest languages for event specification have been developed for use in active databases to realize complex, composite trigger functionality. Apart from their original purpose of e.g. describing transactions or watching method calls in object oriented databases or the like, these languages are universal in their capabilities of building complex expressions from an arbitrary set of simple events. Several event detection strategies have been developed over the last decades like graph-based approaches (Chakravarthy et al. 1994), finite state automata (Gehani, Jagadish, and Shmueli 1993) and colored petri nets (Gatziu and Dittrich 1994). The amount of complex event pattern languages is also immense: ODE (Gehani, Jagadish, and Shmueli 1992), Snoop/SnoopIB (Adaikkalavan and Chakravarthy 2006), Reaction RuleML (Paschke, Kozlenkov, and Boley 2007) are examples of pattern languages, CQL, CCL, StreamSQL are some of many examples with a syntax reminiscent of SQL. Our work was inspired by Snoop and Reaction RuleML. The definition of

[1]Luckham and Schulte define CEP as: "Computing that performs operations on complex events, including reading, creating, transforming, or abstracting them." (Luckham and Schulte 2008)

complex event patterns in our JSON[2] ECA language is to a large extent based on the SnoopIB operators. Our detection algorithms for complex events are graph-based as proposed in (Chakravarthy et al. 1994).

In (Carughi et al. 2007) an architecture for RIA based on events is proposed very close to the Comet architecture. In this paper RIAs have the ability to receive events and to react to those events. Our solution goes one step further as we equip RIAs with the ability to construct complex events out of simple events. This architecture makes our solution more powerful, flexible and open for personalization. The main difference is that we moved the detection of complex events from the server to the client.

In (Schmidt et al. 2007; 2008) an architecture for adapting RIAs based on production rules is proposed and in (Schmidt and Stojanovic 2008) the drawbacks of production rules and the need for complex event processing on the client-side are explained. Our current work takes and lifts the ideas by adding complex event processing capabilities to RIAs.

Motivating Example: Client-side Algorithmic Trading

IRIAs can be applied to application areas whenever the computation and reaction to events is crucial and whenever the application is delivered over the Web. One example, out of many, is monitoring of stock quotes. The example is taken from the domain of financial trading. So far algorithmic trading was the domain of server-side applications only. But with the increase of bandwidth of Internet connections and the growth of computing power of client machines computer intensive event-streaming applications are able to move from the server to the client.

The use case is as follows: A user wants to monitor his/her stock portfolio. In order to do so he/she subscribes to a Comet stream of stock quotes provided by his/her online trader. Having the stock quotes in the form of change events on the client he/she can write an application that processes them. Event processing ranges from simply displaying the actual stock quotes in a table to executing rules triggered by computed complex events in order to provide visual feedback if a user interaction like buying or selling stocks is needed.

Logical System Architecture

The logical system architecture of our framework, enabling the design and execution of IRIAs, is decomposed into the design- and the run-time architecture. The design-time architecture embraces all components in charge of definition, configuration and maintenance of IRIAs. The run-time architecture, in turn, comprises all components covering the real-time algorithmic detection of events and the execution of rules.

Design-time Architecture

The aim of the design-time architecture is to configure the IRIA in order to properly react to incoming events. The component consists of sub-components that allow the user to define, edit or import business vocabularies, stored in an ontology as well as ECA rule sets and event-sources scattered on the Internet. The design and configuration component is part of the whole IRIA and can be easily accessed via the browser. The format of the imported sources must conform to our JSON-Format for representing ontologies (Schmidt et al. 2008), ECA rules and configuration data.

The GUI is the convenient entry point for the user or administrator to configure the IRIA and serves as the visual access point for the subsequent design and import components. It provides an intuitive graphical editor for the business vocabulary and the ECA rules. Besides the rules and the business vocabulary the IRIA has to know: How to connect to the event sources. This information is provided by the configuration component.

The import functionality enables the user to bring in predefined rules, business vocabularies and adapter configurations already encoded in JSON. The files containing the JSON representation can reside on the Internet or on the local file store. We recommend a public repository accessible via HTTP or HTTPS as a single point of access in order to comfortably find the needed information. Coming back to our example, the user could search for already predefined ontologies for stock quotes and stock related events in the Internet repository.

Currently only the import functionality is implemented. We did not yet finish the implementation of a uniform GUI for editing the ontologies, rules and configurations. From a usability point of view all components responsible for the configuration of the IRIA should be hosted as a part of the IRIA itself, because switching between applications and tediously importing different data formats is always an annoying and error-prone job. So, for instance, ontologies must be currently designed in an ontology editor like Protégé[3].

Run-time Architecture

In order to produce a compelling event driven application, different event sources greatly enhance the possibilities. Client-side events are available from user interaction with the browser and Document Object Model (DOM) events. In addition temporal events are available, i.e. timeouts, intervals, recurrence, offsets, etc. Our approach, however, puts the focus on server-triggered events to tap further event sources and thereby opening up to events triggered across the network. For our JavaScript implementation we utilized a server push technique from the general family of Comet architectures. In our example, the Comet server provides a stream of stock market events, e.g. changing quotes. RSS feeds are another type of event sources. This option has the advantage of tapping the host of publicly available data sources today, which broadens the scope of an event enabled application. The option, however, has the downside of not truly being event driven, since events only happen at predefined polling intervals. Feed items, however, can be locally interpreted as events.

[2]JavaScript Object Notation: `http://www.json.org/`

[3]Protégé: `http://protege.stanford.edu`

Incoming events are categorized into types depending on their origin, their purpose or other criteria the event source may impose. The events are then fed into the complex event detection part of our framework, which uses the occurrences according to their type to satisfy complex event specifications from the user's rules. The detection part queues the simple events until they can be used. Alternatively, it discards them, when they can no longer be used in a more complex event. When a complex event belonging to an ECA rule is detected, the associated condition is checked. If an event is present and the condition memory in Rete (Forgy 1982) has also found a match for the condition, the rule action is executed. Actions may be arbitrary JavaScript code or the triggering of a further event or the manipulation of the working memory. Triggering events and altering the working memory can change the outcome of other rules, activating them in turn. Also the DOM may be manipulated for visual or other interaction with the user.

JSON-Rules: A Client-side Rule Language

We propose a lightweight reactive rule language tailored to the needs of Internet applications, specifically applications that profit from or need complex event processing, condition evaluation in working memory and rule actions written in JavaScript.

As a representation for our rules we use JSON, because it is almost directly usable within JavaScript. JSON can specify objects, arrays and primitives. A rule object contains the three attributes `event`, `condition` and `action`. The event part consists of Snoop operators. The condition part uses filters and joins like those in traditional production systems. The action part contains one or more JavaScript code blocks. For the event part the usual Snoop operators are available:

- $Or(e_1, e_2)$ — either of the two events must occur for the complex event to occur
- $And(e_1, e_2)$ — both events must occur
- $Any(m, e_1, e_2, \dots)$ — m of the specified events must occur
- $e_1; e_2$ — the strict sequence of the specified events (the constituent events are not allowed to overlap if they are complex themselves and are detected over an interval of time)
- $A(e_1, e_2, e_3)$ — the aperiodic event is signaled each time e_2 is detected within the time interval formed by the other two events
- $A^*(e_1, e_2, e_3)$ — the cumulative version of the former event is triggered at the end of the interval and accumulates all occurrences (if any) of event e_2
- $P(e_1, TI$[:parameters]$, e_3)$ — the periodic event which is triggered regularly after the time interval TI, an optional list of working memory elements or JavaScript identifiers may be given, the values of which are added to the event occurrences as parameters
- $P^*(e_1, TI$:parameters$, e_3)$ — the cumulative version of the former event, it is detected at the end of e_3 and accumulates all intervals with their parameters, the parameters are mandatory here, because a set of plain, past temporal events would in itself not be of any use
- $Not(e_1, e_2, e_3)$ — this event occurs if no e_2 is detected in the specified interval
- $Plus(e_1, TI)$ — it occurs at TI time after the detection of e_1
- $Mask(e_1,$ condition$)$ — modeled after the event masks from ODE (Gehani, Jagadish, and Shmueli 1992) enforces a condition on the event e_1 allowing e.g. for fine-grained constraints of event types, that may utilize the business vocabulary

The event operators in our rule language are represented as tree nodes. The simple, atomic events form the leaves. This hierarchical representation allows a lean, abstract syntax without constructs from concrete syntax (like parentheses) compared to textual event expressions.

A condition in our language may use comparison operators, set operators, identifiers from the working memory and direct literal values. Comparison operators are $<$, $>$, $=$, $<=$ and $>=$. Identifiers specify items from the working memory.

Rule actions are JavaScript code blocks or events to be triggered or working memory elements (WMEs) to be manipulated. A code block has access to the set of events that has led to the firing of the rule and the WMEs leading to the fulfilled condition part. Thus rule authors may create applications that do calculations on the parameters of the collected events and the matching working memory elements. Use of the business vocabulary provides the necessary means of finding the event parameters and attributes of interest.

Client-side Event-enabled Rule engine

For our implementation we chose JavaScript from the available Web programming languages. The data structures and program logic we implemented are roughly divided into the following areas: adapters for the rule language and the remote event sources, the working memory, condition representation and evaluation as well as complex event detection (see Figure 1).

For complex event detection we are using a graph based approach as proposed in (Chakravarthy et al. 1994). Initially the graph is a tree with nested complex events being parents of their less deeply nested sub-events, down to the leaves being simple events. However, common subtrees may be shared by more than one parent. This saves space and time compared to detecting the same sub-events multiple times, and renders the former tree a directed acyclic graph. The graph is built starting at the leaves, bottom-up. The simple event types from the available rules are stored in a hash map, and form the leaves of the tree. The hash keys are the event names. Each hash value (i.e. leaf) has a list of parents containing pointers to inner tree nodes. These in turn carry references to their parents.

When using the term *event*, a distinction must be drawn between event occurrences (i.e. instances) and event types,

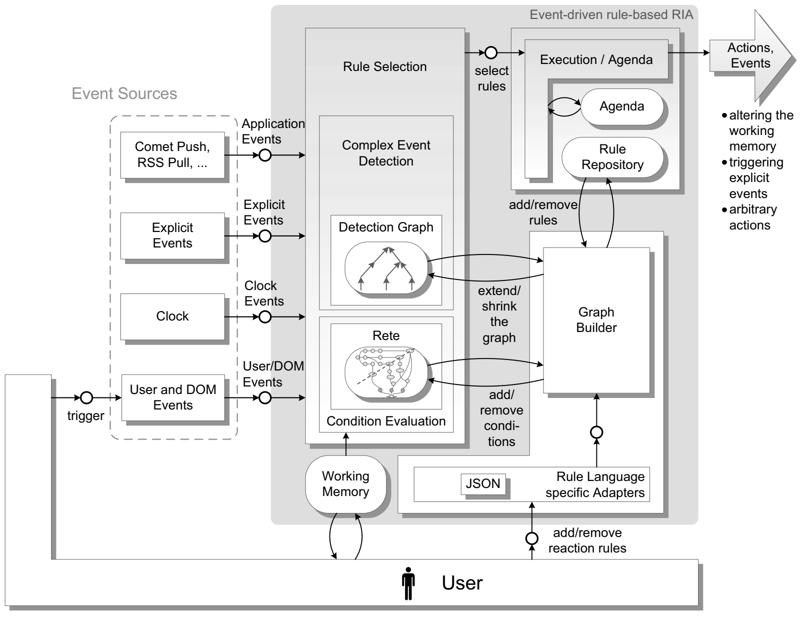

Figure 1: System overview (block diagram)

usually done implicitly. In the detection graph the nodes are event types, they exist before there are any instances. Event instances exist after simple instances arrive and are fed into the graph at the leaves. Complex instances are then formed at the parent nodes, which in turn propagate their results upwards. Every complex event occurrence carries pointers to the set of its constituent event occurrences, so that the events and their parameters can be accessed later. Once an occurrence is computed at a node which is attached to a rule, the evaluation of the associated condition is started.

A condition is converted into a branch of the Rete network, emulating the nesting of operators from the rule language, shown in the Section JSON-Rules. Upon loading a new rule, its condition is used to extend the network. The Rete network is a matching algorithm to find patterns (conditions) in large sets of objects. Objects are contained in a so-called working memory and consist of an identifying string and a type from the set of classes in the business vocabulary. Thus any working memory element can be queried by its name or its type. Hash maps are used to implement fast lookup of the objects by these criteria. Hash maps are also used to represent the *Is-A* relationship for types from the business vocabulary to find matches by supertype. We do not use object-oriented inheritance here because faceted classification[4] is more expressive. Primitives may also be added to the working memory which are autoboxed and unboxed[5] to be handled transparently. Subsequently an item can be retracted, whereupon all references in the working memory and the Rete network are deleted.

Rule execution is done by inspecting the action parts in the rule specification. For actions triggering events a new simple event occurrence is fed into the detection graph at the leaf of the corresponding event type. As the leaves are stored in a hash map, finding the leaf to a name is a simple lookup. For every JavaScript action that is specified in the action part of the rule, the code runs inside a new function that is created at the time of adding the rule to the system. The set of events that led to the activation of a rule is passed to this function. Thus the rule action may employ the data from the constituent events and working memory elements in its computation. That includes the occurrence and duration times, the number and sequence of events, and the parameters carrying all values collected at the occurrence of the events.

Adapters had to be implemented in several components of our framework. The incoming events from the Comet server must be instantiated as first class objects in order to propagate and store them in the detection tree. This is done by an adapter, which constructs new objects for the events, adds the timestamp when it was received, and adds parameters according to the type of event in accordance with the business vocabulary. The event is then fed into the detection graph. Adapters for other event sources may be added, e.g. to facilitate the polling of RSS feeds, and the construction

[4]Faceted classifications “do not require complete knowledge of the entities or their relationships; they are hospitable (can accommodate new entities easily); they are flexible; they are expressive; they can be ad hoc and free-form; and they allow many different perspectives on and approaches to the things classified. She lists three major problems: the difficulty of choosing the right facets; the lack of the ability to express the relationships between them; and the difficulty of visualizing it all.” (Kwasnik 1999; Denton 2003)

[5]Autoboxing is the term for automatically altering a value type into a reference type without a programmer’s being involved . The system automatically supplies the extra code needed to perform the type conversion. Unboxing it the other way around.

of event objects for the feed items or for detected changes in repetitive feed items.

Another adapter converts the declarative rule language into internal data structures using the graph builder component. The adapter dissects the rules. For the event part the rules are turned into nodes for the detection graph. The graph builder incorporates them into the graph, reusing common subtrees that it can detect among the newly added nodes and the existing graph. Among the detected similarities are identical subtrees, commutations of the children of operator nodes And and Or, identical temporal events and identical simple events. For the condition part the graph builder extends the Rete network accordingly to start matching the new expressions. For the JavaScript blocks in the action part of each rule the adapter creates functions. As functions are first class objects in JavaScript, they need to be compiled only once and can be stored by the graph builder for later invocation.

Evaluation

The performance evaluation aims to demonstrate that CEP and a client-side rule engine for the Web are indeed feasible. In this section we will show that an event rate of about 64 events per second is possible with a given rule set on our test machine. Concerning the implementation of the Rete algorithm a rate of about 32 modifications per second is possible.

Our test machine is a 2.4 GHz Intel Core2 CPU with four cores. Since JavaScript execution is inherently single-threaded it profits only from one CPU core. Having spare cores for other tasks and a generally low operating system load provides results uninfluenced by other running tasks. The chosen JavaScript engine is Mozilla Firefox 3.0.3 for Windows using the Firebug[6] profiler. The browser was installed freshly with no extra plugins.

We start out with the BEAST benchmark (Geppert et al. 1998). BEAST is an attempt at measuring CEP performance in early CEP applications from active database systems. We borrow some of the benchmarking rules which are applicable to our CEP engine. Some of the event operators were not applicable to our event pattern language (Snoop), like the count based window operator (cf. Figure 3 on page 8 of (Geppert et al. 1998)).

The remaining rules which are tested are: SEQ(EvED-061, EvED-062), a sequence of two events, NOT(ED07_TX, EvED-07, ED07_TX), the non-occurrence of EvED-07 in the interval of two other specified events and finally: SEQ(EvED-091, SEQ(OR(EvED-092, EvED-093), EvED-094), the sequence of one event, followed by a disjunction and followed by another event.

The tested rules contain empty actions, so only the CPU load for complex event detection is measured. We run each test for 30 seconds at various frequencies of simple events per second. The simple events in the previously mentioned patterns are entered into the detection system in a round robin manner. We then measure the load percentage the detection system takes to match the incoming events and produce complex events.

[6]Firebug Web site: `http://getfirebug.com/`

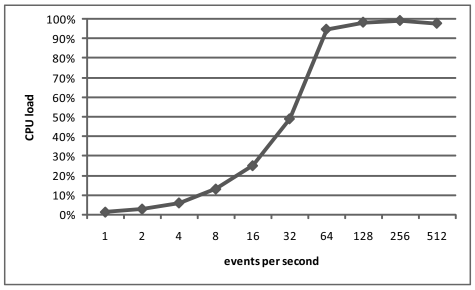

Figure 2: CPU load by increasing event frequency

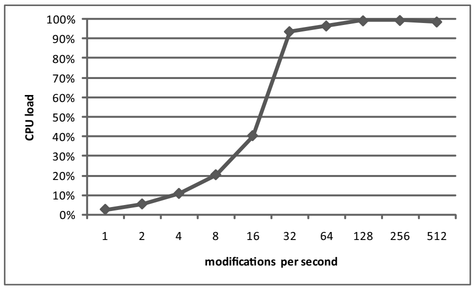

Figure 3: CPU load by increasing fact modifications

The results are shown in Figure 2. The chart shows that our event detector can handle a maximum of about 64 events per second in real time. After that the JavaScript engine is used up to capacity and further incoming events are queued up.

We take a similar approach for rule conditions to measure the performance of our Rete implementation. Three condition expressions are added to the Rete network with the same nesting depths as the event expressions. The working memory is initially filled with 1000 working memory elements (WMEs). Then we start modifying WMEs at different frequencies, round robin. Again, we measure the CPU load caused by the pattern matching functions. Figure 3 shows the result. Our rule engine is capable of processing about 32 modifications per second without being overloaded.

The general use case for our framework is to aid in making RIAs more reactive and adaptive. This means reacting upon user input, as well as server-generated events of interest to the user, etc. Events from the human user and events received across the Internet are not occurring at millisecond rates and our framework is fast enough. However, it should be mentioned that expensive rule actions may lessen these results. On the other hand, upcoming new browsers promise a significant increase in JavaScript performance due to newer compiling techniques. Nevertheless, currently we expect the results to be sufficient for most client-side applications including our example application.

Conclusions and Future Work

In this paper we presented a novel type of event-driven, reactive and adaptive Rich Internet Application: IRIA. We demonstrated that the use of declarative event patterns and rule expressions yields IRIAs capable of processing and reacting to continuous event streams. We showed that IRIAs are an alternative to pure server-side event processing applications. With the help of an example derived from financial trading we explained our design- and run-time architecture and the JSON ECA rules. In the evaluation section we proved the efficiency of our approach to event detection and condition evaluation. In future work we will present the formal definition of our JSON ECA rule language and another example application demonstrating the use of client-side reactive rules to enhance an e-Government Web portal.

References

Adaikkalavan, R., and Chakravarthy, S. 2006. Snoopib: Interval-based event specification and detection for active databases. *Data Knowl. Eng.* 59(1):139–165.

Carughi, G. T.; Comai, S.; Bozzon, A.; and Fraternali, P. 2007. Modeling distributed events in data-intensive rich internet applications. In Benatallah, B.; Casati, F.; Georgakopoulos, D.; Bartolini, C.; Sadiq, W.; and Godart, C., eds., *WISE*, volume 4831 of *Lecture Notes in Computer Science*, 593–602. Springer.

Chakravarthy, S.; Krishnaprasad, V.; Anwar, E.; and Kim, S. K. 1994. Composite events for active databases: Semantics, contexts and detection. In Bocca, J. B.; Jarke, M.; and Zaniolo, C., eds., *20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile proceedings*, 606–617. Los Altos, CA 94022, USA: Morgan Kaufmann Publishers.

Denton, W. 2003. How to make a faceted classification and put it on the web. http://www.miskatonic.org/library/facet-web-howto.html.

Forgy, C. L. 1982. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19:17–37.

Gatziu, S., and Dittrich, K. R. 1994. Detecting composite events in active database systems using petrinets. In *Proc. Fourth International Workshop on Active Database Systems Research Issues in Data Engineering*, 2–9.

Gehani, N. H.; Jagadish, H. V.; and Shmueli, O. 1992. Event specification in an active object-oriented database. *SIGMOD Rec.* 21(2):81–90.

Gehani, N. H.; Jagadish, H. V.; and Shmueli, O. 1993. Compose: A system for composite specification and detection. In *Advanced Database Systems*, 3–15. London, UK: Springer-Verlag.

Geppert, A.; Berndtsson, M.; Lieuwen, D.; and Roncancio, C. 1998. Performance evaluation of object-oriented active database systems using the beast benchmark. *Theor. Pract. Object Syst.* 4(3):135–149.

Kwasnik, B. H. 1999. The role of classification in knowledge representation and discovery. *Library Trends* 48(1):22–47.

Luckham, D. C., and Schulte, R. 2008. Event processing glossary. Online Resource. `http://complexevents.com/?p=361`. Updated: July 2008. Last visited: November 2008.

Paschke, A.; Kozlenkov, A.; and Boley, H. 2007. A homogenous reaction rules language for complex event processing. In *International Workshop on Event Drive Architecture for Complex Event Process*.

Russell, A. 2006. Comet: Low latency data for the browser. Online Article. `http://alex.dojotoolkit.org/?p=545`. Last visited: July 2008.

Schmidt, K.-U., and Stojanovic, L. 2008. From business rules to application rules in rich internet applications. In *To be published in BIS*, Lecture Notes in Computer Science. Springer.

Schmidt, K.-U.; Stojanovic, L.; Stojanovic, N.; and Thomas, S. 2007. On enriching ajax with semantics: The web personalization use case. In *Proceedings of the European Semantic Web Conference, ESWC2007*, volume 4519 of *Lecture Notes in Computer Science*. Springer-Verlag.

Schmidt, K.-U.; Dörflinger, J.; Rahmani, T.; Sahbi, M.; Stojanovic, L.; and Thomas, S. M. 2008. An user interface adaptation architecture for rich internet applications. In Bechhofer, S.; Hauswirth, M.; Hoffmann, J.; and Koubarakis, M., eds., *ESWC*, volume 5021 of *Lecture Notes in Computer Science*, 736–750. Springer.