Scalable Computing: Practice and Experience
Volume 9, Number 4, pp. 329–340. http://www.scpe.org
ISSN 1895-1767

FROM BUSINESS RULES TO APPLICATION RULES IN RICH INTERNET APPLICATIONS

KAY-UWE SCHMIDT, ROLAND STÜHMER*, AND LJILJANA STOJANOVIC†

Abstract. The increase of digital bandwidth and computing power of personal computers as well as the rise of the Web 2.0 came along with a new web programming paradigm: Rich Internet Applications. On the other hand, powerful server-side business rules engines appeared over the last years and let enterprises describe their business policies declaratively as business rules. This paper addresses the problem of how to combine the business rules approach with the new programming paradigm of Rich Internet Applications. We present a novel approach that reuses business rules for deriving declarative presentation and visualization logic. In this paper we introduce a rule-driven architecture capable of executing rules directly on the client by implementing the Rete algorithm. We propose to use declarative rules as platform independent language describing the application and presentation logic. By means of AJAX we exemplarily show how to use client-side executable rules for adapting the user interface of Rich Internet Applications. We call our approach ARRIA: Adaptive Reactive Rich Internet Applications. In order to show the usability of our approach we explain our approach based on an example taken from the financing sector.

Key words: rich internet application, declarative user interface, rule engine, event condition action rules, AJAX

1. Introduction. Today's business world is characterized by globalization and rapidly changing markets. Thus in recent years business processes do not change yearly but monthly, the product lifecycle has shrunk from months to weeks in some industries and the process execution time has decreased from weeks to minutes as a result of the technological progress over the last few years. On the other side, the life cycle of IT applications stayed constant over time [23]. Business rules already proved their potential of bridging the gap between dynamic business processes and static IT applications. By declaratively describing the policies and practices of an enterprise the business rules approach offers the flexibility needed by modern enterprises.

At the same time with the dawn of the Web 2.0, a new technology for web applications appeared: AJAX [15]. Because of the Web 2.0 and AJAX, Rich Internet Applications (RIAs) emerged from their shadow existence in the World Wide Web. AJAX, in contrast to Adobe Flex (`http://www.adobe.com/products/flex`), now enables RIAs running in browsers without the need for any additional plug-in. Several Web 2.0 applications use AJAX heavily in order to provide a desktop-like behavior to the user. Now the time seems right for RIAs, because of the broad bandwidth of today's Internet connections, as well as the availability of powerful and cheap personal computers. Besides AJAX, other prominent members of the RIA enabling technologies are: Adobe Flex, Microsoft Silverlight (`http://www.microsoft.com/silverlight`), OpenLaszlo (`http://www.openlaszlo.org`), to mention just a few.

Given those two trends observable in today's IT landscape, traditional ways of programming web applications no longer meet the demands of modern Rich Internet applications. So, the strict distinction between declarative business logic and hard coded presentation logic does no longer hold. As web citizens are accustomed to highly responsive Web 2.0 applications like Gmail (`http://mail.google.com`), web applications based on business rules also have to provide the same responsiveness in order to stay competitive.

In this paper we propose a novel, declarative architecture for RIAs. We coined our proposed system architecture ARRIA which stands for Adaptive Reactive Rich Internet Application. In our architecture all business rules, affecting the UI and not demanding intensive back-end processing, are transferred into a client-readable format at design time. We call these rules in the following application rules. At run-time the application rules are executed directly on the client by a client-side rule engine. That enables a RIA to react straight to user interactions. The event patterns triggering the rules are found by a complex event processing unit. After identifying appropriate events, the application rules, in the form of event condition action (ECA) rules, are executed directly on the client. As a proof-of-concept and in order to evaluate the idea of ARRIAs we prototypically realized a rule-driven RIA using AJAX as client-side technology.

The paper is structured as follows: In Section 2 we present an example in order to motivate our work. The following Section 3 describes the historical development of rule-driven systems. In Section 4 we analyze the semantic and syntactic requirements for a client-side executable rule language. We present in Section 5 our JSON rules approach, an implementation of these requirements. Based on our motivating example we show in

*SAP AG, Research, Vincenz-Prießnitz-Straße 1, 76131 Karlsruhe, {Kay-Uwe.Schmidt, Roland.Stuehmer}@sap.com
†FZI Forschungszentrum Informatik, Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, Stojanovic@fzi.de

Section 6 how to derive application rules from business rules. Additional, in this section we show an exemplary JSON rule for manipulating objects. The architecture of our ARRIA approach is detailed in Section 7. The subsequent Section 8 elaborates on the implementation details. Section 9 gives an overview of related research in the filed of rule-driven RIAs, and, finally, the paper closes with Section 10, conclusions and prospects for future work.

2. Motivating example. For motivating our work we chose an example from the financing sector. The example illustrating our approach is an online application for a loan. The use case is as follows: A person wants to apply for a loan from a bank. S/he visits the web portal of that bank in order to fill in the online loan application. Figure 2.1 shows the form. The web site offers four input possibilities: first, the name of the applicant; second, the amount of the requested loan; third the income of the applicant, and, finally, the kind of employment. The two buttons below the form submit or cancel the loan application.

The IT department of the bank decided to implement the online loan application as RIA in order to take advantage of the advanced visualization techniques. The RIA shall give immediate feedback to the borrower signaling the probability of acceptance. Therefore, a traffic light was additionally introduced on the web page. The lights indicate the status of the application for a loan. A *red* light signals a low or zero probability that the loan will be granted. *Yellow* means that a clerk has to decide whether or not the loan application will be accepted. Finally a *green* light indicates that, based on the input data, the loan will be granted in all probability. The traffic light shall change as the user fills in the online form without explicitly asking the server. That leads to a desktop-like behavior of the web application.

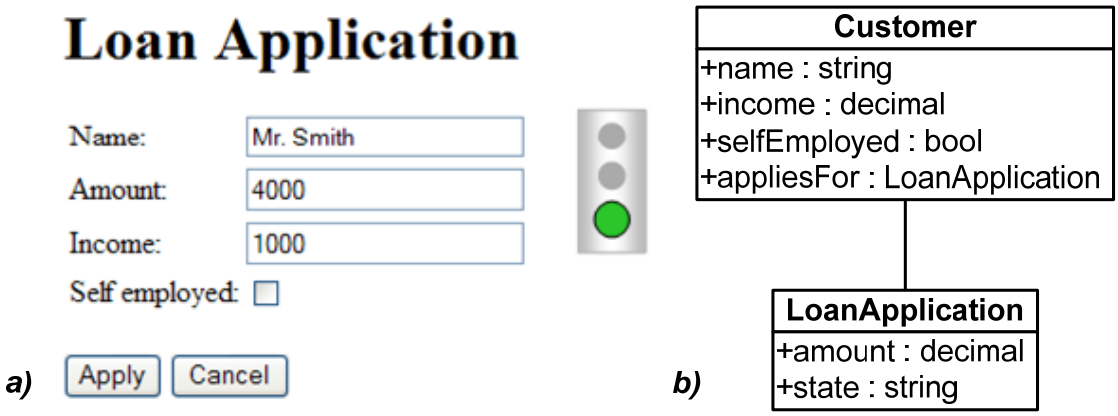

FIG. 2.1. *Motivating example taken from the finance sector*

The business logic of the application for a loan is well understood and written down as business rules, since they are subject to frequent changes. The RIA, and, especially, the manipulation of the traffic light, can reuse and can be built upon these business rules. The rules shown in Example 1 declaratively represent the business logic behind a loan application. For the sake of simplicity we abstract from the amount of the loan. The rules are written in the *IF/THEN* syntax because of its simplicity and its commonness of use.

Example 1 Business Rules.

```
IF C.income {\textgreater}= 1000 AND NOT C.selfEmployed THEN
    L.state = ''accepted''
IF C.income {\textgreater}= 1000 AND C.selfEmployed THEN
    L.state = ''to be checked''
IF C.income {\textless} 1000 THEN L.state = ''rejected''
```

Figure 2.1 b) depicts the UML class diagram of the business objects (BOs). BOs are objects that encapsulate real world data and business behavior associated with the entities that they represent [20]. They are also called objects in a domain model. A domain model represents the set of domain objects and their relationships. The two BOs engaged in our example are *Customer* and *LoanApplication*. They are connected by the relation

appliesFor which links one customer to one or many loan applications. The attributes of the customer class store customer specific attributes like name, income and employment status, whereas the attributes of the loan class hold loan specific data like the amount or the approval status of the application. A loan application can have the following statuses: *accepted*, *to be checked* or *rejected*.

The business rules depicted in Figure 2.1 a) define the business logic of when to grand a loan application. C is a placeholder for *Customer* objects and L for *LoanApplication* objects. The first rule states that if the borrower's income is grater then or equal to 1000 Euro and s/he is not self employed the loan will be granted in all probability. The second business rule states that if the income is greater then or equal to 1000 and s/he is self employed a clerk has to judge manually whether the loan will be granted or not. If the income is less then 1000 the loan will not be granted at all. In our example, all business rules are atomic. That means they are independent of each other and pairwise disjunct.

3. The evaluation of rule-driven web applications. Legacy rule-driven web applications are based on the web page paradigm as depicted in the left graphic in Figure 3.1. The web page paradigm states that every web page in a series of pages is downloaded separately. User data are collected in forms on the client and are sent to the server by user request. On the server side a production engine processes the input data and executes actions manipulating business objects. Based on the modified business objects a new web page is created and sent back to the client. Business rules in the back-end declaratively describe the business logic of the web application.

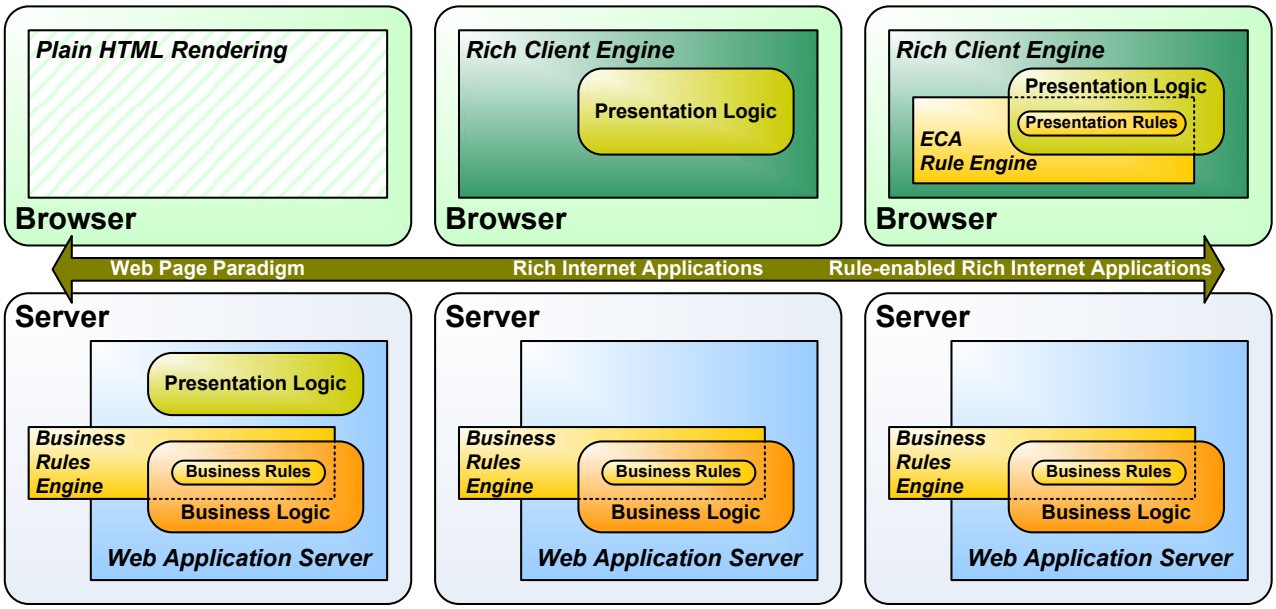

Fig. 3.1. *Evolution of rule-driven web applications*

Rich Internet Applications (RIAs) break the web page paradigm by introducing rich client-side functionality and asynchronous communication facilities. The middlemost graphic in Figure 3.1 depicts the evolution of RIAs from common rule-driven web applications. Up to date browsers provide a rich client engine capable of executing dynamic presentation logics. Together with the business logic, the production system stays on the server side but can be requested asynchronously. That is, business rules can be evaluated without being explicitly triggered by a user request.

Turning RIAs based on server-side rules into client-side rule-driven RIAs, that benefit from the best of the two worlds, is not trivial. Switching from the request/response communication of web applications relying on the web page paradigm to the asynchronous communication of RIAs goes only half way. Although asynchronous communication with the web server allows a RIA to reload only altered data rather then the page as a whole, as well as to pre-load chunks of data that might be good candidates for displaying next, the desired desktop-like responsive behavior is not achieved. This is because business rules and especially business rules concerned with the presentation layer are still evaluated on the server-side. Every user interaction, from pressing a button to

hovering the mouse over an artifact on the web site, must be processed on the server in order to let business rules fire appropriate actions as reaction to a user input. Also the advantage of the declarative character of rules is getting lost by only applying the rule paradigm to business logic and not to presentation logic. Presentation logic is also a good candidate for declarative modeling because it remains unchanged even for different platforms.

4. Requirements for a Client-Side Application Rule Language. For managing the proposed client-side rule engine, an appropriate rule language is an indispensable prerequisite. The language must consider requirements specific to Rich Internet Applications.

The semantics of our ECA rule language is constituted by the semantics of the events, conditions and actions by themselves. The semantics of each constituent can be separately defined, for example by reduction to their respective underlying languages. But there is more to it than that. On top of the composed semantics the overall semantics of the language as a whole must be clarified: the relationships between events, conditions and actions.

The so-called coupling modes from early research on ECA rules in the HiPAC project [12] (p.129-143) point out several relationships between events and conditions. However, in all cases a condition is evaluated after an event has occurred. No mode is defined requiring conditions to be fulfilled during the entire occurrence of an event. More recent works, e.g. in [3], suggest a revised semantics for ECA rules. It is stated there that the complete condition of a rule has to be satisfied during the whole detection time of the composite event, i. e. from the beginning of the occurrence of the first constituent event up to the end of the occurrence of its last constituent event. This understanding of ECA rules conforms to the notion of interval-based semantics established for complex events. Interval-based semantics views an event as having a duration, instead of viewing it as an instant at detection time. The duration lasts from the start of the first constituent event to the end of the last constituent event. Therefore, an accompanying condition should span the entire interval of the event duration. The downsides of not using interval-based semantics are pointed out by Galton and Augusto [14] and Berstel [3] for conditions. For events this includes unexpected results from transitivity of multiple sequence operators, and for conditions it includes possible matches with events, violating temporal axioms like matching system context in the future.

Furthermore, the language must expose all user-adjustable features of the event detection, the condition matching and the different kinds of actions. A good complex event detector relies on three things: An easy to use rule language, a rich set of event detection operators and an efficient algorithm to evaluate these operators. Furthermore, the event detection algorithm has to be an active instead of a passive query-based one. We want to stress this a little bit more. Events happen asynchronously and are generally not predicable by nature. Therefore, we insist on a forward-chaining algorithm that pushes actively new events in an appropriate data structure that proactively detects complex events. We are convinced that such a solution outperforms query-based pull strategies for instance proposed by Paschke et al [25].

Conditions are formulae over the state of an application. When a given formula is fulfilled, the system is in a state where the rule author wants some action to be executed. Traditional rule systems only execute condition action rules. The systems are called production systems. Two examples are OPS5 [4] and CLIPS [10]. To evaluate most types of ECA rules, a separate condition matcher is required in addition to the event detection. This can best be observed from the fact that condition action rules lack the triggering event specification, therefore another way must be provided to find and run any applicable rule. Furthermore, any applicable rule should be found at the time when its condition is fully satisfied. This means that changes to the state of the application should immediately be reflected in the activation of rules. No query-driven semantics should be used for rule activation because it would restrict the capacity to act to only certain intervals at which queries are issued. Instead of a query-driven (top-down) approach, a data-driven approach must be employed. A data-driven approach fulfils conditional predicates in a bottom-up way, also called forward-chaining. The advantage of forward-chaining evaluation is that for each change of state affecting a condition, the partial match is saved until it can be further completed to form a complete match in the end. Complete matches are reported immediately when they come into existence.

There are several requirements for the action part of rules. First of all the rule engine should allow for the highest possible flexibility, this means that arbitrary JavaScript actions must be allowed. Apart from the imperative approach using JavaScript, a declarative approach should be supported as it is offered by traditional production systems like OPS5 or CLIPS. In these systems the actions can alter the system state by only specifying modifications to objects. Such modifications include adding and deleting objects, as well as modification

of business objects. As a third type of rule action it might be useful to explicitly feed a new event back into the system. Other rules would be able to react to such an event, just like an event from any of the other event sources.

Also the non-functional requirement of user-friendliness targets several aspects. First of all the language should be extensible. This includes permitting the future use of JavaScript features which are not known today. Also, this includes the possibility of adding further operators for the event and condition part. In addition to extensibility, some measures of reusability should be provided. For example, complex event expressions which are repeated in several rules should be made reusable at design-time. The user should have the possibility of creating a set of named event expressions. These predefined expressions can be incorporated into further event expressions of different rules. Methods of reuse should also be provided for condition expressions and possibly actions. For the latter it might be possible to offer a library of predefined actions. User interface patterns [26] might help in finding a meaningful selection of such actions to be provided for the rule author. User-friendliness should also cover the run-time of the rule framework. One important requirement arises when a rule author wants to add and remove rules while the rule engine is running. Both the event detection and the condition matching algorithms must be able to alter their data structures in a coherent manner when rules are added or deleted from detection.

Lastly, on account of browser-friendliness there are also some non-functional requirements for a rule language. As far as the possible acceptance of a new rule language goes, it can be very important that the language closely fits the environment in which it is to be used. To accomplish this, the language should be lightweight, easy to deploy in a RIA and AJAX environment and should honor JavaScript programming practices, where possible.

Luckham writes in his book [22] on event processing, that an event language must be expressive enough, must be notationally simple, semantically precise and must have an efficient pattern matcher. He says this about event languages, but the preceding analysis has shown that Luckham's requirements hold true for the condition part, just as they do for the event part.

5. JSON Rules. We implemented the above analyzed requirements in a rule language named JSON rules. It is a language for defining client-side executable reaction rules. Reaction rules are triples of events, conditions and actions. From a user's point of view the rule language is the interface to programming and adapting ARRIAs. For the rule language the JavaScript-friendly JavaScript Object Notation (JSON) format is chosen. JSON is published as a Request for Comments (RFC) [9]. Like XML it provides a structured representation of data with deep nesting. Unlike XML it is readily usable in JavaScript because JSON syntax is the subset of JavaScript otherwise used to denote objects literals and array literals in the programming language. Although, JSON is JavaScript there is a thin parsing layer involved to provide security from introducing executable code. Other than that, JSON uses a very lean syntax compared to XML. Tags do not need to be named if, for example, they are just used to provide structure like nesting. JSON can be used to maintain nested data; therefore, our rule language can be formulated in JSON as an abstract syntax tree. A similar approach is taken by many modern XML-based languages, like RuleML and its ECA rule standard, Reaction RuleML [25]. Using an abstract syntax tree to transport the language relieves the client-side application of parsing any expressions. Instead the nesting of expressions can be easily determined by descending the supplied tree. Also, no aspects of concrete syntax must be retained when abstract syntax is used.

The complete grammar of our declarative client-side JSON ECA rule language is designed in (extended) Backus-Naur Form (BNF). We designed and tested the rule language grammar with the parser generator tool ANTLR (ANother Tool for Language Recognition, `http://www.antlr.org/`). The grammar describes a so-called rule file. The rule file is the granularity at which rules are transported, e.g. downloaded into the rule framework. A rule file may contain more than one ECA rule in a rule set. Meta data for the rule set are also part of the rule file and a library of reusable event expressions. The language is a specialization of JSON. The syntax of JSON can describe strings, numbers, the Boolean literals *true* and *false* as well as objects and arrays. Objects are enclosed in curly braces. They contain a comma separated list of attributes. An attribute is a string followed by a colon and the value. The value might in turn be any JSON expression. Arrays are enclosed in square brackets, containing a comma separated list of expressions. The proposed language restricts tree-expressions from JSON in way that only certain objects with certain attributes may be used and nested. The language is therefore a subset of JSON. An example JSON rule is given in the next chapter.

6. Deriving application rules from business rules. The starting point for every RIA is the business logic. The business logic declaratively encoded into business rules coarsely defines the presentation logic of the user interface for RIAs. But business rules are usually high-level and are not related to any user interface issues. On the other hand, application rules presenting the presentation logic have to control, on a fine grained level, complex user interfaces. Therefore, the first step in creating application rule sets is the analysis of the business rules and their related business objects. Based on this analysis, the user interface and the presentation logic in the form of declarative application rules can be designed.

The application rule in Example 2 is directly derived from the first business rule of the Example 1. It manipulates all JavaScript *LoanApplication* objects associated with a dedicated *Customer* object, whenever any property of the *Customer* object has changed. A web designer merely has to listen to *PropertyChangedEvents* of the *LoanApplication* object referenced by the *$LoanApp* variable and, if an event has been fired, to adjust the traffic light accordingly. On the other hand, it would also be possible to change the traffic light directly within the rule body by injecting JavaScript code directly into the rule's action part. The printout depicted in Example 2 shows the entire rule set in our case consisting of a single application rule. In line 01 the name of the rule set is defined. From line 02 to 16 a condition action rule is defined. Line 02 states the rule name and line 03 the description of the rule. From line 04 to 13 the condition is formulated. The condition consists of two parts, the first relates to the customer (line 04–08) and the second to the loan application (line 09–13). Line 12 joins all objects of *Customer* meeting the constraints defined in the lines 06–08 with all objects of *LoanApplication* that are not already accepted. In our example the RIA contains only one object of *Customer* and one object of *LoanApplication*. When all constraints are satisfied the action in line 14 is fired.

In line 13 the example JSON rule contains an extra constraint field checking whether `state` is unequal to "accepted". This constraint ensures that the rule is not invoked several times by the execution algorithm. On each change to the rule system runs all rules which have a matched condition. Therefore, a rule fires several times as long as its condition still matches the objects. Since our example rule would always set the loan application to *accepted* regardless of whether this has been done before, the rule would loop endlessly. The solution is to alter the rule in a way so that its condition is invalidated after the rule is run for the first time. Because the rule modifies an attribute which is not part of the condition, we correct this by adding the extra constraint to the rule. The stronger condition ensures that the rule does not match objects which were matched before.

Example 2 JSON Application Rule.

```
01 {"meta": {"ruleSet": ‘‘Loan Application Example"},
02 ‘‘rules": [{"meta": {"rule": ‘‘GrantLoans",
03 ‘‘description": ‘‘Grant loan!"},
04 ‘‘condition": [{"class": ‘‘Customer",
05 ‘‘fields": [
06   {"field": ‘‘income", ‘‘comparator": ‘‘>=", ‘‘literal": 1000},
07   {"field": ‘‘selfEmployed", ‘‘comparator": ‘‘==", ‘‘literal": false},
08   {"field": ‘‘appliesFor", ‘‘vardef": ‘‘$LoanAppID"}]},
09                          {"class": ‘‘LoanApplication",
10 ‘‘vardef": ‘‘$LoanApp",
11 ‘‘fields": [
12   {"field": ‘‘id", ‘‘comparator": ‘‘==", ‘‘variable": ‘‘$LoanAppID"},
13   {"field": ‘‘state", ‘‘comparator": ‘‘!=", ‘‘literal": ‘‘accepted"}]}],
14 ‘‘action": [{"type": ‘‘MODIFY",
15 ‘‘name": ‘‘$LoanApp",
16 ‘‘modify": ‘‘this.state = ’accepted’;"}]}]}
```

7. Architecture. First we highlight the design of the CEP engine followed by the design of the rule engine. For the design of an efficient complex event detector several alternative algorithms were proposed in the past. They differ in their detection approach, using either automata [18], Petri-nets [17] or a graph-based approach [8]. They also differ in their effects on the semantics of events they detect, and differ in general versatility.

SnoopIB [1] is chosen from the available approaches as a basis for the event detection in this thesis. Along with that, Snoop's operators are adopted with some extensions and with according extensions of the detection algorithm. A reason for choosing Snoop over the other detection methods is that the graph-based approach of

SNOOP allows the detection of overlapping complex events. This rules out automaton-based event detection as complex events of a given complex type may occur simultaneously. This means several complex incidents of the same type happen at the same time, in an overlapping fashion. Automaton-based algorithms are not capable of detecting more than one instance of the same complex event at the same time. This is an inherent drawback of how automata are used for event detection. As elaborated in Gehani at all [18, 19] automata are constructed from regular expressions specifying event patterns. Transitions model accepted events in a given state. An initial state is created with transitions for initiator events, the initial constituent events. The transitions lead to further states, and so on, up to one or more accepting states, where the complex event is detected. The complex event is then defined as the sequence of transitions which were taken from an initiator to a terminator event. When an automaton must accept overlapping complex events, the following happens. A suitable initiator changes the state of the automaton away from the initial state by using one of the transitions. The automaton will then be in a state which accepts constituent events to continue completion of the first complex event. There might be no transitions accepting initiators for further complex events, until the automaton is reset after completely detecting the first. Although there might be other transitions labeled with the initiator event type, these events will be incorporated in the first complex event as intermediate constituents. Other complex events are only started at the initial state. In summary this means that overlapping complex events are ignored, because once an automaton is in the process of detecting a complex event, it is usually not in its initial state anymore, to start detecting a second complex event at the same time. Algorithms based on Petri nets and on graphs do not share this deficiency. An important drawback of the Petri net-based approach, however, is that Petri nets do not support user-defined selection of tokens when a transition is fired. This means it cannot be predetermined by the user, which constituent events, e.g. tokens, are used when creating a complex event. Therefore, SAMOS [17] does not provide configurable event selection policies in its Petri net-based approach. Coloured Petri nets are introduced in Jensen [21]. They allow tokens to be individually distinguished at the transitions might accomplish event selection based on individual attributes. However, SAMOS uses colours only to model event parameters and to propagate these parameters through the Petri net. This concludes the major reasons for choosing the graph-based approach over automata or Petri nets. Automata cannot detect concurrent complex events and Petri nets do not offer a clear strategy for event selection.

The choice of detection semantics is the next important decision which has to be made on behalf of the event detection. The detection semantics are concerned with whether complex events are represented by an interval or only by a point in time. The preceding analysis for this work showed that a detection-based (point in time) semantics delivers unexpected results for certain operators, e.g. the sequence operator. Snoop revised its semantics towards an interval-based view of events, called SnoopIB [1]. The same holds for other event detection system like Reaction RuleML [25].

According to the requirements we decided to use Rete [13] as forward-chaining discrimination network for the evaluation of the conditions parts of a rule. The Rete algorithm has several similarities with the previously describes detection graph for events. Both are forward-chaining pattern matching algorithms and both must be able to add and remove nodes at runtime, etc. However, there are some important differences. Firstly, it must be noted that they serve different purposes. In terms of semantics of rules [5], the event graph is concerned with transient, temporal data, i. e. events. The Rete network, on the other hand, is concerned with persistent data, representing the system state, i. e. business objects. The two types of data are to be separated in order to avoid making unnecessary events persistent, and thereby imposing a storage burden on an application.

Figure 7.1 depicts the client-side components of the run-time architecture. The server-side components are skipped for the sake of simplicity. The software components of the run-time architecture carry out the application logic encoded in the declarative application rules. The application rules are transferred to the client together with the content data in response to the first initial user request. In the first preprocessing step the CEP Unit responsible for detecting complex events is initialized and, in a second step, the appropriate event handlers are set. As complex events are not issued directly by user interface widgets the CEP Unit has to register for each atomic event contained in complex events.

When the user interacts with the portal, he/she fills in forms, navigates through the site, and goes back, searches for terms and so on. All those interactions trigger events like mouse movements in the appropriate controls. The CEP Unit handles all atomic events to which it has subscribed in advance (step three) by its SnoopIB implementation. Based on the directives of the event detection algebra, it tries to identify complex patterns from the event stream. After detecting a complex event, the associated rules are evaluated by the client-side rule engine. This is step four in Figure 7.1 In step five the condition parts of the rules are evaluated,

if there are any, using the Rete algorithm. If the event and condition part of rule is matched during the evaluation phase, it is fired immediately.

The execution of a rule can have manifold actions which are marked as 6a to c. In step 6a a rule manipulates the status of the application. The status of the application is maintained in working memory. In a nutshell, the working memory consists of an arbitrary amount of local object variables. Further a change to the working memory can trigger additional rules that are not explicitly bound to any complex event pattern. These rules are conventional production or condition action rules (CA rules). A rule can also manipulate the user interface directly as depicted in step 6b. By this means, application rules can respond to user interactions immediately without an explicit server request. These rules are the guarantors of a responsive user interface. Any user interface manipulation can issue additional atomic events that might be recognized by the CEP Unit as parts of complex events. New rules can be triggered. So the rule execution in step 6b can trigger additional rules over the event detection mechanism. The last possible action of a rule execution is depicted in step 6c: The invocation of the Asynchronous Communication Controller (ACC). The ACC is responsible for loading new rule sets, for pre-fetching content data as well as for synchronizing with the BO's on the server-side. As a direct byproduct of pre-fetching data and synchronizing with the server, the ACC can alter the user interface.

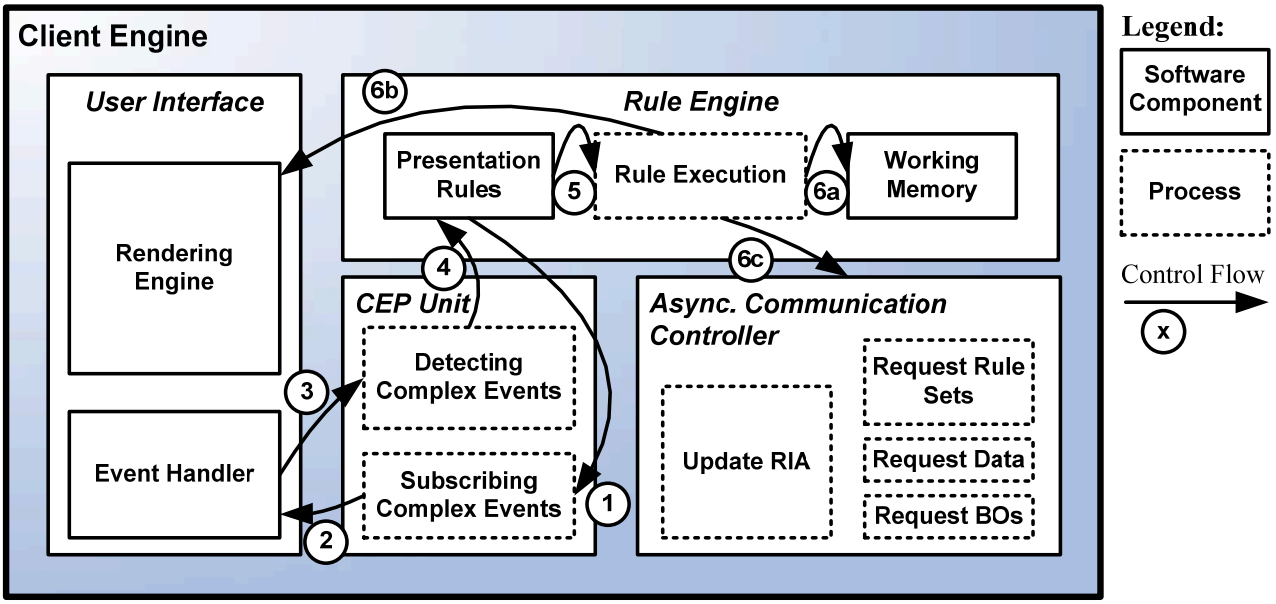

Fig. 7.1. *Run-time architecture*

8. Implementation of the run-time architecture. We implemented our event detection as well as our rule engine in JavaScript using a slightly modified SnoopIB and object-based Rete algorithm. The event graph is a network of nodes which represent event expressions. There are special nodes types for every event type. Incoming edges of a node originate in child nodes which represent sub-expressions. Simple event nodes have no incoming edges. Outgoing edges connect a node to its parent which makes further use of detected events. Detected events are propagated upwards in the network, starting with simple events which are fed into the graph at the simple event nodes. The propagation ends at top nodes which have no further parents. In these nodes events are extracted from the graph and are handed on to some action, which in the process discards the event. Event nodes may have more than one parent. This occurs when an event expression is used in several places of a pattern. The reused expression is then manifested only once in the event graph but outgoing edges are linked to all nodes where the expression is reused. All parent nodes are informed equally of detected events.

We implemented the following event operators in our JSON rule language. The logical operators from Snoop that we implemented are: `Or`, `And`, `Any`, as well as `Not`. Operators `And` and `Or` are binary operators in the sense that they involve two operands. The `Any` operator is a generalized form of the preceding ones. It accepts an arbitrary list of parameters and a parameter m, which specifies the number of events that must be detected to match the Any pattern.

Additional, we implemented Snoop's temporal operators: `Seq`, `A`, `A*`, `P`, `P*`, as well as `Plus`. The operator `Seq` is the sequence of two events in time. Operators `A` and `A*` are ternary operators, detecting occurrences of one event type when they happen within an interval formed by the two other event types. `A*` is a variant which collects all events and occurs only once at the end of the interval with all the collected constituents. `P` and `P*` are ternary operators as well, they also accept two events starting and ending an interval, but the third parameter is a time expression after which the events occur periodically during the given interval. A function may be specified to collect event parameters for each periodic occurrence. `P*` is the cumulative variant which occurs only once, containing all collected constituents. `P` stands for periodic because of its metronome characteristics. `A` stands for aperiodic because the detected constituents occur at irregular times. The `Plus` operator accepts an event type and a time expression. The `Plus` event occurs after the specified event type has occurred and the specified time has passed.

The operators mentioned so far are the complete set from Snoop. Content-based checks are added to them in order to fulfill the requirement for filtering by event parameters. Content-based checks do not provide structure as the previously described operators do. Content based checks filter streams of events, resulting in streams which contain only events matching a constraint check. Such checks are concerned with the parameters of events. The appropriate event operators are called *guard* by David Luckham or *mask* by the authors of Ode. We use the term mask. The event mask is designed as an operator with one event input and a Boolean function to be applied to the input. The value returned from the function decides about whether the input is accepted or discarded. An incoming event is accepted if the function returns `true`. When specifying a mask expression, the function itself may be selected from a set of predefined mask types. Moreover, the event masks in this work are extensible in the way that the function may optionally be an arbitrary user-defined implementation.

The Rete network is constructed from the top downwards, contrary to the event graph. This is because working memory elements (WMEs) enter the Rete network at a single, top node. As with the event graph, equal nodes must be shared. Equality is likewise determined by the function of a node combined with its input, meaning its predecessor nodes. Constructing the Rete network from the rule specification is done as follows. Each class pattern is first converted into a series of consecutive alpha nodes. There are different types of alpha nodes forming sub-classes of Node.Alpha, cf. Figure 8.1. These alpha nodes for example perform checks on the class of an object, the existence of an attribute of an object, or comparisons with the values of attributes, etc. On adding it to the network, each alpha node is linked to its predecessor, checking whether an equal node is already among the successor nodes and sharing it, if so. After the single object checks are completely represented in the network, an alpha memory is added in the end to store the output. To create the successive beta network, joins are gathered from the rule specification. Every free variable occurring in more than one object pattern is invoking a join. Joins are then ordered in pair wise joins by variable and by input memory. Beta nodes are then created with the necessary join predicates and attached to the matching alpha memories. A join predicate or Test is a JavaScript function. It is selected from a hash map of predefined comparator functions which are selected by the comparator specification in each rule. Comparator functions include wrappers for the built-in comparators from JavaScript like <, >, <=, >=, ==, != and like ===, !== which do not perform type coercions like their two-letter counterparts do. Also the JavaScript special operator *typeof* is available, which allows checks for the types of objects and primitives. Adding more functions to the hash map here provides simple extensibility for the rule framework. The comparator functions are two-parameter functions with Boolean result because they are used as join predicates. The functions are stored in the Test objects in join nodes. A join node has a beta memory as one input and an alpha memory as another. The beta memory supplies tokens which are lists of objects satisfying preceding joins. The alpha memory supplies plain objects (in the form of WMEs) which must match the other objects in the token according to the join predicates. After finishing all joins in the beta memory, a production node is added to the network. Such a node is a beta memory containing finished tokens representing a complete join. Each such token resembles a fully matched pattern and therefore a rule action is triggered from the production node.

9. Related work. Rule-driven Rich Internet Applications seems to be a new and novel approach, as we could not find related work on this topic. Nevertheless, there exists already a reasonable amount of work addressing subtopics of our approach. Carughi at al [6] describe RIAs as reactive systems where the user interface produces events. They use complex event processing in conjunction with server push technologies, but not for triggering application logic formulated in declarative application rules, that can be executed directly on the client. In their work complex events trigger some kind of server-side logic. They also do not address how complex events can be detected on the client-side.

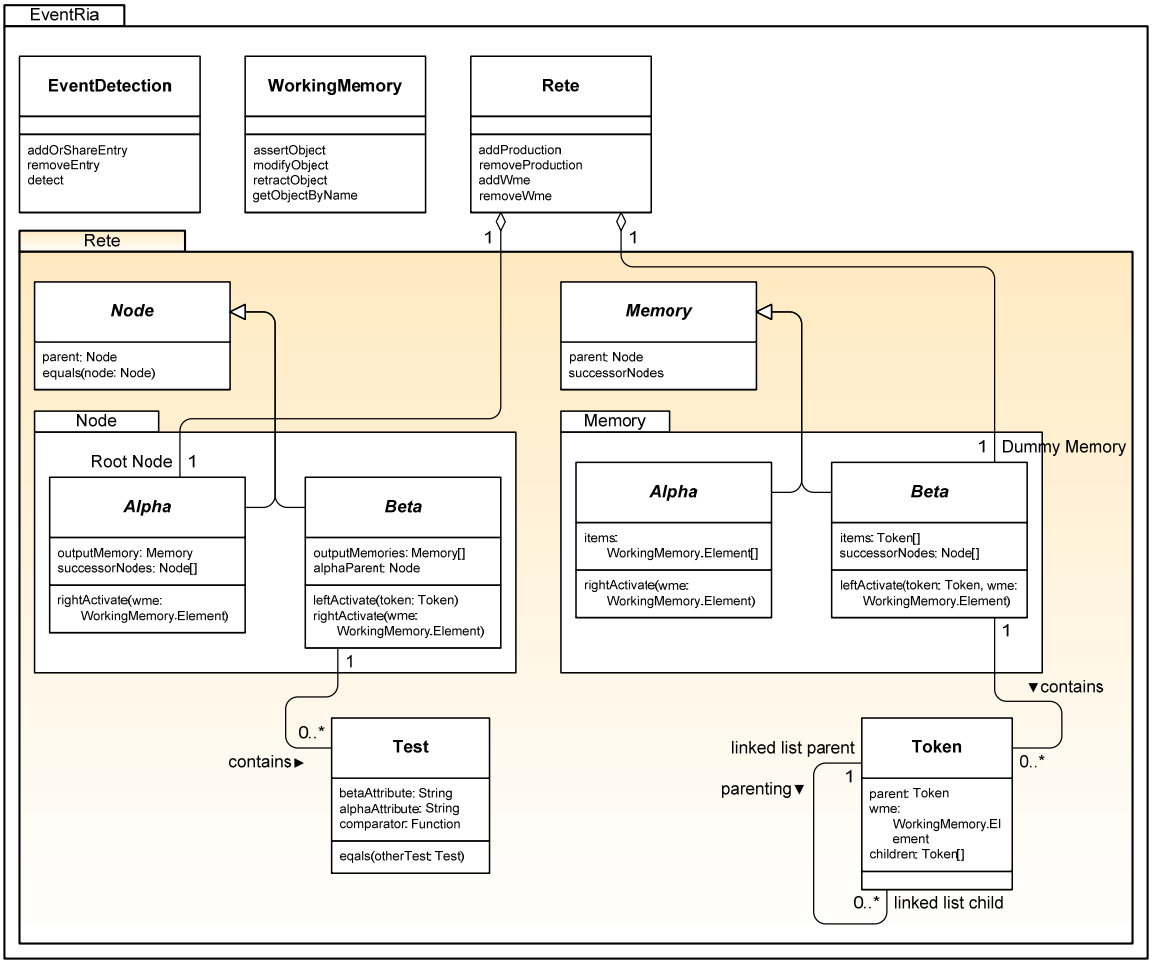

FIG. 8.1. *Rete Network (Class Diagram). This diagram shows the classes comprising the Rete network. The Rete class contains an alpha node in the role of the Root Node. Also, the dummy beta memory is connected to Rete. The rest of the network is reachable through objects of these two classes. Tokens are implemented as a linked list, so token objects are parenting token objects.*

The principles of complex event processing for reactive databases are well understood since the mid-1990s. Chakravarthy et al [8] outline an expressive event specification language for reactive database systems. They also provide algorithms for the detection of composite events and an architecture for an event detector along with its implementation. Our work in the field of complex event processing relies greatly on their work and the work done by Chakravarthy and Mishra [8], Papamarkos et al [24] and Alferes and Tagni [2]. Recently some effort was undertaken to broaden RuleML (`http://www.ruleml.org/`) to a event specification language. As a result Reaction RuleML (`http://ibis.in.tum.de/research/ReactionRuleML/`) [25] incorporates nicely different kinds of production, action, reaction, complex event processing and event logic rules into the native RuleML syntax but fails to support OWL ontologies.

In the web engineering paper of Garrigós et al, [16] AWAC is presented, a prototype CAWE tool for the automatic generation of adaptive web applications based on the A-OOH methodology. The authors define the Personalization Rules Modeling Language (PRML) an ECA language tailored the personalization needs of web applications. Our rule language follows a different approach as it has to deal with complex events on the client-side. PRML does not support complex event processing and is not a general purpose ECA language supporting more then personalization, in contradiction to our JSON rules.

The ECA-Web language suggested by Daniel at al [11] is an enhanced XML-based event condition action language for the specification of active rules, conceived to manage adaptiveness in web applications. Our JSON-Rules are different to that approach as we, as stated in the name, relay on JSON as exchange and execution

format. Moreover, we incorporated an event algebra for specifying complex events based on Snoop. Besides that, the whole adaptation approach is quite different as we support real-time adaptation directly on the client compared to the server-side adaptation and rule execution approach of ECA-Web.

10. Conclusions and future work. In this paper we presented a novel approach of using declarative application rules as a new programming model for RIAs. We call this amalgam of event processing, rule engine and RIA: ARRIA – Adaptive Reactive Rich Internet Application. By providing event detection we enable the web designer to define the behavior of the web application based on the order the user issues interaction events in time, that is based on order of his/her actions. The declarative application logic can be easily changed by rewriting the rules. The ECA rules can be executed without additional coding by arbitrary target systems like AJAX, Silverlight or Flex. We developed a light-weight ECA rule language tailored to the needs of RIAs. Furthermore, we implemented an enhanced event detection engine based on the SnoopIB algorithm. For matching the conditions of ECA rules we decided to implement a light-weight version of the Rete algorithm. As a proof of concept we implemented our motivating example using JSON rules. The ARRIA framework consisting of event detection and rule evaluation was implemented in JavaScript. As RIAs are not only AJAX applications we currently implement our framework in Silverlight. Moreover, we will evaluate the performance of the ARRIA framework and we will implement other use cases where our architecture will show its full potential.

REFERENCES

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

[2] J. J. Alferes and G. E. Tagni, *Implementation of a complex event engine for the web.*, in SCW, IEEE Computer Society, 2006, pp. 65–72.

[3] B. Berstel, *Extending the rete algorithm for event management*, in Proc. Ninth International Symposium on Temporal Representation and Reasoning TIME 2002, Washington, DC, USA, 7–9 July 2002, IEEE Computer Society, pp. 49–51.

[4] L. Brownston, R. Farrell, E. Kant, and N. Martin, *Programming expert systems in OPS5: an introduction to rule-based programming*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985.

[5] F. Bry and M. Eckert, *Twelve theses on reactive rules for the web*, in EDBT Workshops, 2006, pp. 842–854.

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

[7] S. Casteleyn, F. Daniel, P. Dolog, M. Matera, G.-J. Houben, and O. D. Troyer, eds., *Proceedings of the 2nd International Workshop on Adaptation and Evolution in Web Systems Engineering AEWSE'07, Como, Italy, July 19, 2007*, vol. 267 of CEUR Workshop Proceedings, CEUR-WS.org, 2007.

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

[9] D. Crockford, *Rfc4627: Javascript object notation*, tech. report, IETF, 2006.

[10] C. Culbert, G. Riley, and B. Donnell, *Clips reference manual volume 1, basic programming guide, clips version 6.0*, Software Technology Branch, Lyndon B. Johnson Space Center, NASA, (1993).

[11] F. Daniel, M. Matera, A. Morandi, M. Mortari, and G. Pozzi, *Active rules for runtime adaptivity management*, in Casteleyn et al. [7].

[12] U. Dayal, A. P. Buchmann, and D. R. McCarthy, *Rules are objects too: A knowledge model for an active, object-oriented databasesystem*, in Lecture notes in computer science on Advances in object-oriented database systems, New York, NY, USA, 1988, Springer-Verlag New York, Inc., pp. 129–143.

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

[14] A. Galton and J. C. Augusto, *Two approaches to event definition*, in DEXA '02: Proceedings of the 13th International Conference on Database and Expert Systems Applications, London, UK, 2002, Springer-Verlag, pp. 547–556.

[15] J. J. Garrett, *Ajax: A new approach to web applications*, `http://www.adaptivepath.com/publications/essays/archives/000385.php` (2005).

[16] I. Garrigós, C. Cruz, and J. Gómez, *A prototype tool for the automatic generation of adaptive websites*, in Casteleyn et al. [7].

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

[18] N. H. Gehani, H. V. Jagadish, and O. Shmueli, *Composite event specification in active databases: Model & implementation*, in VLDB '92: Proceedings of the 18th International Conference on Very Large Data Bases, San Francisco, CA, USA, 1992, Morgan Kaufmann Publishers Inc., pp. 327–338.

[19] ———, *Compose: A system for composite specification and detection*, in Advanced Database Systems, London, UK, 1993, Springer-Verlag, pp. 3–15.

[20] R. Heidasch, *Get ready for the next generation of sap business applications based on the enterprise service-oriented architecture (enterprise soa)*, SAP Professional Journal, 9 (July/August 2007), pp. 103–128.
[21] K. Jensen, *Coloured Petri Nets: Basic Concepts, Analysis Methods, and Practical Use*, Springer, 1992.
[22] D. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
[23] N. MacDonald, *Strategies for business growth*, in Gartner Symposium ITXPO, 2002.
[24] G. Papamarkos, A. Poulovassilis, and P. T. Wood, *Event-condition-action rule languages for the semantic web*, in SWDB, 2003, pp. 309–327.
[25] A. Paschke, A. Kozlenkov, and H. Boley, *A homogenous reaction rules language for complex event processing*, in International Workshop on Event Drive Architecture for Complex Event Process, 2007.
[26] J. Tidwell, *Designing interfaces*, O'Reilly, 1. ed. ed., 2006.

Edited by: Dominik Flejter, Tomasz Kaczmarek, Marek Kowalkiewicz
Received: December 1st, 2007
Accepted: January 15th, 2008
Extended version received: August 8th, 2008