A Lightweight Architecture of an ECA Rule Engine for Web Browsers

Emilian Pascalau[1] and Adrian Giurca[2]

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

Abstract. There is a large literature concerning rule engines (forward chaining or backward chaining). During the last thirty years there were various proposals such as RETE, TREAT and the derived Gator algorithm. Significantly, RETE was embedded into various expert systems such as Clips and its successor Jess, and Drools including in a number of commercial rule engines and was extended various times including with support for ECA rules. However, none of them is able to directly process DOM Events. The goal of this paper is to present the architecture of a forward chaining Event-Condition-Action (ECA) rule engine capable to handle Document-Object-Model Events. This architecture is instantiated into a JavaScript-based rule engine working with JSON rules.

1 Motivation

There is a large literature concerning rule engines (forward chaining or backward chaining). During the last thirty years there were various proposals such as RETE [6], TREAT [15] and the Gator algorithm [13] which is derived from the other two. Significantly, RETE was embedded into various expert systems such as Clips and its successor Jess[7], and Drools [18]. RETE [6] was extended various times including with support for ECA rules [5].

However, none of them is able to directly process DOM Events. The goal of this paper is to present the architecture of a forward chaining ECA rule engine capable to handle Document-Object-Model Events. This architecture is instantiated into a JavaScript-based rule engine working with JSON rules [10].

The main goals this design should address are:

- to move the reasoning process to the client-side resulting in reduced network traffic and faster response;
- to handle complex business workflows;
- information can be fetched and displayed in anticipation of the user response;
- pages can be incrementally updated in response to the user input, including the usage of cached data;
- to offer support for intelligent user interfaces;
- enable users to collaborate and share information on the WWW through real-time communication channels (rule sharing and interchange);

Complex event processing (CEP), is a methodology of processing events taking into consideration processing multiple events with the goal of identifying the meaningful events within a specific time-frame or event cloud. CEP employs techniques such as *detection of complex patterns*, *event correlation*, *event abstraction*, *event hierarchies*, and *relationships between events* such as causality, membership, and timing, and event-driven processes. A number of projects were developed in the last ten years on these issues. However, there is one event ontology which offers large opportunities to be exploited in the context of actual technologies such as Asynchronous JavaScript and XML (AJAX) [8] allowing the development of intelligent Rich Internet Applications (RIAs) i.e. web applications that typically run in a web browser, and do not require software installation ([1]) - The Document Object Model Events (DOM Events). This event ontology[1] provides a large amount of events types designed with two main goals: (1) the design of an event system allowing registration of event listeners and describing event flow through a tree structure (the DOM), and (2) defining standard modules of events for user interface control and notifications of document mutation, including defined contextual information for each of these event modules.

This ontology is already implemented into browsers giving extremely powerful capabilities to RIAs which use it. Nowadays, several Web 2.0 applications use heavily AJAX in order to provide desktop-like behavior to the user. The number of RIAs is increasing because of the broad bandwidth of today's Internet connections, as well as the availability of powerful and cheap personal computers. However, traditional ways of programming Internet applications no longer meet the demands of intelligent (rule-enabled) RIAs. For example a highly responsive Web 2.0 application such as Google Mail, can be much easily personalized/customized using rules towards a declarative description of its behavior.

Implementing intelligent RIAs require reasoning possibilities inside the browser. In addition, using Event-Condition-Action (ECA) Rules to represent knowledge unveils the opportunity to design and run rule-based applications in the browser.

2 The Architecture of an ECA Rule Engine for Web Browsers

2.1 The Components View

As depicted in Figure 1, the complete system comprises the *Event Manager*, *Rule Repository*, *Inference Engine* and *Working Memory*.

The main design goal of this architecture was to comply with the principles of Software as a Service (SaaS) architectures [4]. Therefore, the main capabilities considered in this design were:

- *Distributed Architecture* - all these components can act in different network locations.

[1] `http://www.w3.org/TR/DOM-Level-3-Events/`

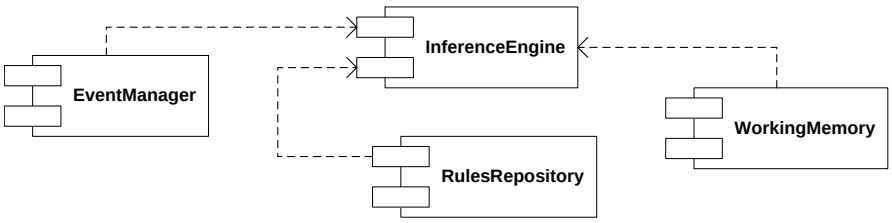

Fig. 1. Components View

– *Event-driven architecture* - We emphasize that both human agents and software agents interact with this architecture by creating events i.e. the reasoning is event driven. Moreover, the architecture instantiation gets translated into a full event driven engine.

This architecture is a live system i.e. an event-based system that is reactive and proactive. It is reactive because it reacts based on the events it receives. It is proactive because by itself generates events, that can be consumed also by other entities being part of the whole system.

2.2 The Working Memory

In the database community the main goal of designing ECA engines was to provide generic functionality independent from the actual languages and semantics of event detection, queries, and actions (see for example, [3] and [19]). However, two main issues make the difference: (a) in the case of an ECA architecture the Working Memory besides the usual standard facts it contains also event-facts and (b) the distinction between facts and event-facts is that the last ones are immediately consumed while traditionally facts are kept until specific deleting actions are performed.

2.3 The Event Manager

During the last years there is an intense work either on defining design patterns for complex event processing [17] or theoretical work on how Event-Condition-Action rules can be combined with event algebras for specification of the event part [2].

Our goal is to provide a light *Event Manager* capable to process faster simple events without duration. Particularly, this architecture must handle DOM Level 3 Events. However, the extension points in the *Event Manager* make possible future extensions for complex events processing if we will be able to provide motivating use cases.

Basically the *Event Manager* (depicted in Figure 2) has an event vocabulary and listen for events. Its main activity is to create an event queue to be processed by the inference engine.

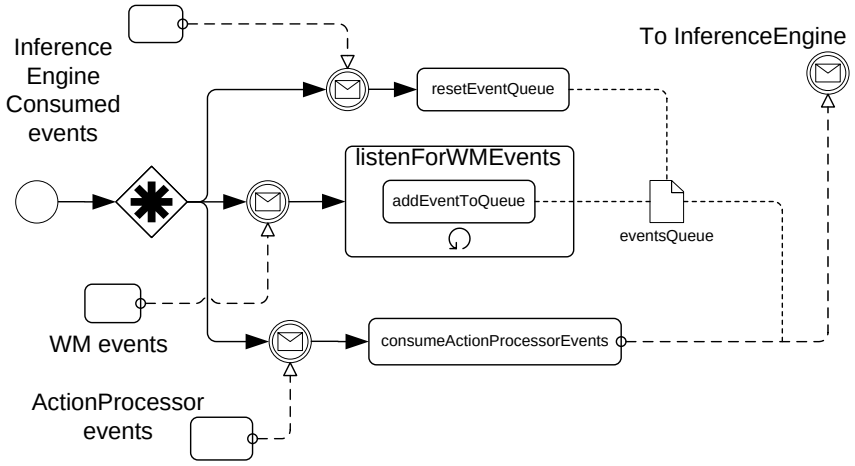

Fig. 2. EventManager

The *Event Manager* keeps on catching Working Memory (WM) event-facts and stores them in an event queue. In addition, it listens for two internal *Action Processor* messages:

- *busy* - The *Event Manager* keeps on catching WM events and storing them in the working queue of events.
- *idle* - The *Action Processor* informs that it is not working right now. The *Event Manager* pro-actively takes control and send its own message to the *Inference Engine* with the actual working queue of events. Each time an event is caught by the *Event Manager* it tries to find out about the state of the *Action Processor*. If there is no new message from the *Action Processor* it keeps going on based on its actual knowledge of the *Action Processor* state. It changes its knowledge when it receives a new message from the *Action Processor*.

Finally, the manager handles the inference engine consumed events. Our model looks for mandatory handling of engine consumed events as the default mechanism to achieve the event consumption. Therefore if there are events which are not processed/consumed by the inference engine they are kept by the manager on its lifetime or until they are consumed by rules.

2.4 The Inference Engine

Our goal were not to use RETE and its variants (although influences exist) but to build a lightweight engine. Our goals were not to embed strong efficient execution algorithms rather to offer a simple, extensible and fast rule execution engine. All these design goals were coming from our main goal: *running rules in the Web browser*.

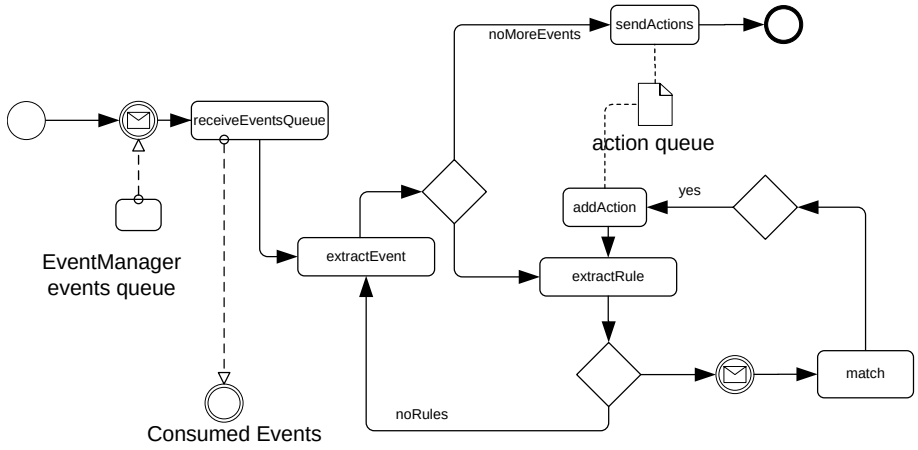

Fig. 3. InferenceEngine

The basic activities inside the *Inference Engine* (see the Figure 3) are to *consume events* (from the events queue delivered by the *Event Manager*) *match rule conditions* (match) and *deliver action queue* to the *Action Processor* (`sendActions`). Despite other architectures where the actions are consumed inside the inference engine, we decided for a separate component since the *Action Processor* is not just a blind action executor but is able to perform various consistency checks after it has received its queue of actions. The rules intended to be handled by this architecture are JSON Rules [10] (see 1 for a small rule example) which provide priorities for handling rule order execution. Our engine does not provide any conflict resolution mechanism i.e. does not handle any specificity, recency or refractoriness principles, but it can be extended to support such mechanisms. Finally the engine has no formal semantics such as other expert systems paradigms [14]. The syntax of a JSON rule is similar to JSON notation.

Example 1 (JSON Rule example).

```
{"id": "rule101",
  "appliesTo": ["http://mail.yahoo.com/"],
  "eventExpression": {"type": "click",
                      "target": "$X"
                     },
  "condition": [
                "$X:HTMLAnchorElement($hrefVal:href)",
                "new RegExp(/showMessage\?fid=Inbox/).test($hrefVal)"
              ],
  "actions":["append($X.textContent)"]
}
```

2.5 The Rules Repository

As we already know, the purpose of business rules repositories is to support the business rule information needs of all the rule owners in a business rules-based

approach in the initial development of systems and their lifetime enhancement. Specifically, the purpose of a business rules repository is to provide: (a) Support for the rule-related requirements of all business, (b) Query and reporting capability for impact analysis, traceability and business rule reuse including web-based publication and (c) Security for the integrity of business rule and rule-related information.

Parts of our previous work (see for example, [16]) introduced the architecture of such a registry. Basically, inside this architecture, the *Rules Repository* is responsible to handle loading and deploying of rule sets.

3 JSON Rules - Architecture Instantiation

We introduce the instantiation of our architecture in the JSON Rules context. Recall from [10] that JSON Rules where introduced and defined to tackle a particular environment which is the Web Browser. While the first part of this work addresses the architectural issues from the Platform-Independent Model (PIM) [12], [9] perspective, this part addresses it from the Platform-specific Model (PSM) perspective.

According to the reference architecture for Web Browsers introduced in [11] the system introduced here finds itself as part of the *Rendering Engine*. In the general perspective the JSON Rules engine will come as part of the accessed resource.

In the case of the Mozilla [2] browser's architecture [11] the system might be either part of the *Rendering Engine* or part of the UI Toolkit (XPFE[3] - Mozilla's cross-platform front end) if the system is packed as a browser add-on. The second approach gives greater flexibility since the UI of Mozilla browsers is XML based and as such uses an extended version of the rendering engine used to display the content of a specified resource. Based on this the JSON Rules engine introduced here seams quite feasible to be used to change also the UI and behavior of the browser itself.

The general components view depicted in Figure 1 gets instantiated in the JSON Rules context as depicted in Figure 4.

Depicted in Figure 4 are the main packages of the JSON Rules engine. While the `engine` and `repository` packages are self explanatory to some extent `lang` package contains all the JSON Rules language entities. The `utils` package contains entities dealing with different aspects such as: JSON parsing, or object introspection and so on. The `io` package provides the necessary entities managing IO operations. The `engine` package contains the following sub-packages: `eventmanager`, `actionprocessor`, and `matcher`. The general flow of the whole system is described in the Figure 5 (initially introduced in [10]).

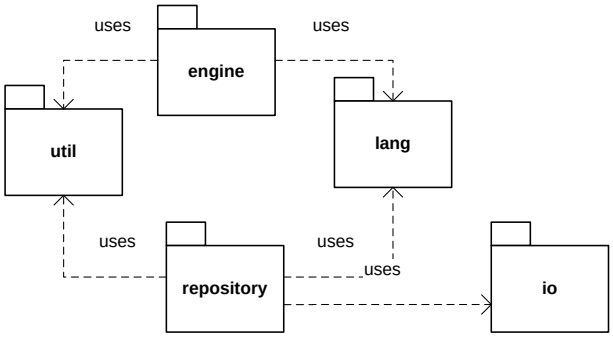

Fig. 4. Rule Engine - Packages

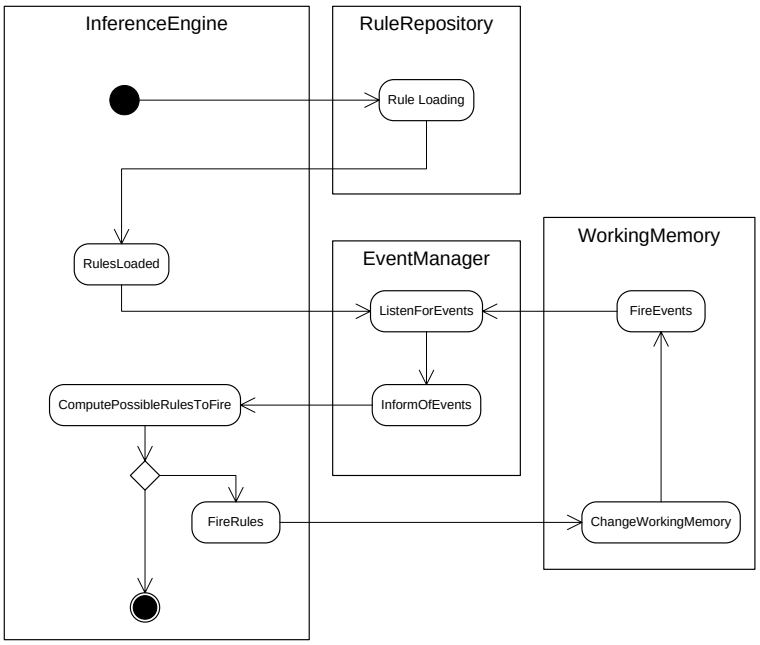

Fig. 5. The Rule Engine State Diagram

3.1 Main System

The `MainSystem` (Figure 6) is the interface through which the inference engine is accessed. As seen in the Figure 6 the only mandatory input is an Uniform Resource Identifier (URI) pointing towards a rule repository, from where rules will be loaded.

[2] `http://www.mozilla.com/`
[3] `http://www.mozilla.org/xpfe/`

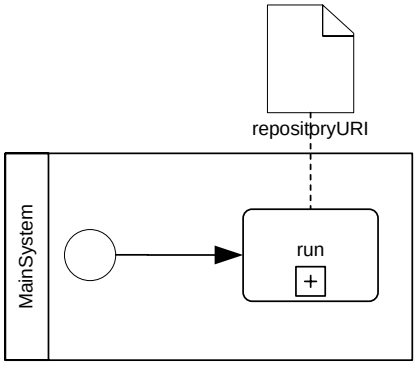

Fig. 6. MainSystem

Although not concretely depicted here through this interface a user could also specify other settings such as different event type for which the event manager should listen for.

Having specified a repository location the `MainSystem` performs the `run` activity.

The lifetime of the rule engine is in the scope of the lifetime of the current DOM inside the browser. Using the engine is simple. Firstly one must load the engine e.g. this

```
<script type="text/javascript"
  src="http://www.domain.com/jsonRulesEngine_Version.js">
</script>.
```

Secondly one must create an instance of the engine, for example:

```
var jsonRulesEngine=new org.jsonrules.JSONRulesMainSystem();
```

Having the engine instantiated, it is now possible to run it by calling `run()` with the URI of location of the repository as input parameter:

```
jsonRulesEngine.run("http://www.domain.com/rulesRepo.txt");
```

In a basic application the main steps that happen are:

- When an event is raised, the `EventManager` catches that event. Then the `EventManager` checks the `ActionProcessor`'s state.
- If the `ActionProcessor` is running, then `EventManager` stores the event in the queue of events that the `InferenceEngine` must later on process.
- However if the `ActionProcessor` is idle then the `EventManager` sends a message to the `InferenceEngine` containing the queue of events that must be processed. The `InferenceEngine` responds back to the `EventManager`, and informs it that it has received/consumed the queue such that the `EventManager` can reset its own queue.

– Events are processed one by one. For each event rules triggered by that event will be matched against the `WorkingMemory`. The action of each executable rule is added to the list of executable actions (to be processed by the `ActionProcessor`) according with possible priority of rules.
– The list of executable actions it is send to the `ActionProcessor`, to execute them.

3.2 Working Memory

As already introduced in [10] the `WorkingMemory` consists of the loaded DOM for the current active resource. Recall that by resource we mean the content which a browser loads in the form of a DOM representation from a specified URI. `WorkingMemory` facts are based on the DOM content. Moreover, in the context of our architecture, `WorkingMemory is` driven by events and contains event-facts. This type of behavior is imposed by the event-based nature of the DOM.

3.3 Event Manager

In addition to DOM Level 3 Events, the DOM specification provides the necessary interfaces through which an user-defined event can be created. However, in general, DOM events are simple events even though users could create their own events. There is also the possibility to use and define complex events by means of user defined APIs such as Yahoo YUI[4], Dojo toolkit[5] etc. To deal with such user defined APIs the `EventManager` uses the concept of adapter. An adapter can be written for each API and in this way events defined using those APIs could also be tackled by the `EventManager`.

Another significant aspect of the browser based instantiation is that the whole flow is by nature sequential. Actual browsers' JavaScript engines are sequential, and because of this, so is the whole engine introduced here. However in the eventuality of a browser with capabilities to run parallel JavaScript tasks then the general architecture could be instantiated following the ability to run parallel tasks.

3.4 Inference Engine

Figure 7 depicts the interaction between the `MainSystem` and the `InferenceEngine`. The `InferenceEngine` receives a `page` object form the `MainSystem`. Its subcomponents (`EventManager`, `ActionProcessor`, `Matcher`) are also instantiated and in this manner the system becomes alive by listening and throwing events.

3.5 Rule Repository

While a more detailed perspective on rule repositories has been already introduced in [16] here we use a simplified version of that. Rules defined in the repos-

[4] `http://developer.yahoo.com/yui/`
[5] `http://www.dojotoolkit.org/`

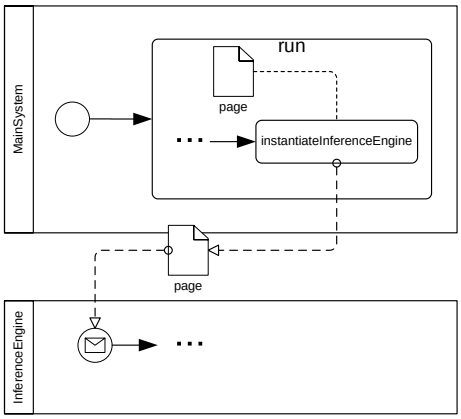

Fig. 7. System-InferenceEngine

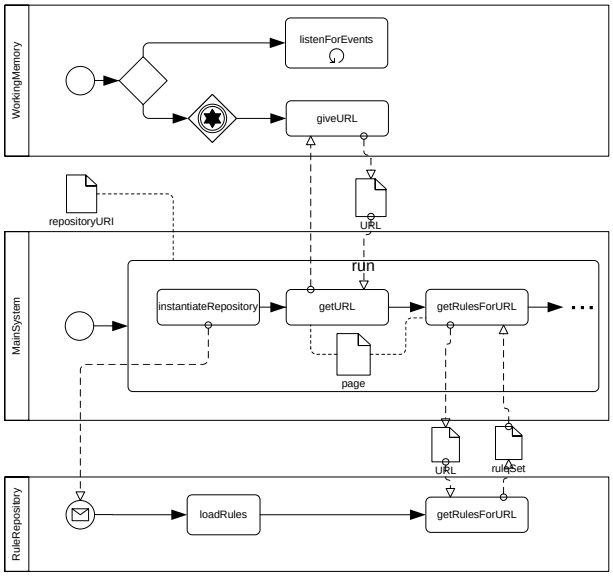

Fig. 8. RuleRepository

itory refer to a specific URI. This means that a specific rule can be used in the context of a specific resource. Rules referring to the same URI are grouped in rule sets.

Figure 8 depicts the interaction between the `MainSystem` and the `RuleRepository`. Basically, the `MainSystem` triggers a `RuleRepository` instance. The repository loads the rules from the `repositoryURI` specified location. Readers may notice that the repository might contain rules that do not refer to the current active resource. As such the `MainSystem` requests the URI of the current resource from the `WorkingMemory`. Based on that URI it requests from the repository the rule set referring to the current resource. Finally, based on this information (i.e. the URI of the current resource and the rule set associated) the `MainSystem` creates a `Page` object which will be used by the `InferenceEngine`.

4 Conclusion

This paper describes the general architecture of an ECA rule-based and forward chaining engine for web browsers. The design of such an engine derives from the goal to perform intelligent RIAs. The instantiation of the architecture results in a JavaScript-based ECA rule engine capable to load and execute ECA rule sets in the browser. This way we achieve a main goal: Implementing intelligent RIAs require reasoning possibilities inside the browser. The next steps related to this research are: (1) to investigate the capabilities of this engine to handle rule-based mashups on the Web and (2) to analyze scalability of the engine against the main browsers.

References

1. Jeremy Allaire. Macromedia Flash MXA next-generation rich client. `http://www.adobe.com/devnet/flash/whitepapers/richclient.pdf`, March 2002.
2. Erik Behrends, Oliver Fritzen, Wolfgang May, and Franz Schenk. Embedding Event Algebras and Process for ECA Rules for the Semantic Web. *Fundamenta Informaticae*, 82(3):237–263, 2008.
3. Erik Behrends, Oliver Fritzen, Wolfgang May, and Daniel Schubert. An ECA Engine for Deploying Heterogeneous Component Languages in the Semantic Web. In *Current Trends in Database Technology - EDBT Worshops*, pages 887–898, 2006.
4. Keith Bennett, Paul Layzell, David Budgen, Pearl Brereton, Linda Macaulay, and Malcolm Munro. Service-Based Software: The Future for Flexible Software. In *Proceedings of the Seventh Asia-Pacific Software Engineering Conference (APSEC2000)*, pages 214 – 221. IEEE Computer Society, 2000. `http://www.bds.ie/Pdf/ServiceOriented1.pdf`.
5. Bruno Berstel. Extending the RETE Algorithm for Event Management. In *TIME*, pages 49–51, 2002.
6. Charles Forgy. Rete – A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.
7. E. Friedman-Hill. Jess The Rule Engine for the Java Platform. `http://www.jessrules.com/jess/docs/Jess71p2.pdf`, November 2008.

8. Jesse James Garrett. Ajax: A new approach to web applications. http://www.adaptivepath.com/ideas/essays/archives/000385.php, February 2005.
9. Dragan Gasevic, Dragan Djuric, and Vladan Devedzic. *Model Driven Architecture and Ontology Development*. Springer Verlag, 2006.
10. A. Giurca and E. Pascalau. JSON Rules. In *Proceedings of the Proceedings of 4th Knowledge Engineering and Software Engineering, KESE 2008*, volume 425, pages 7–18. CEUR Workshop Proceedings, 2008.
11. Alan Grosskurth and Michael W. Godfrey. A Reference architecture for web browsers. In *Proceedings of the 21st IEEE international conference on software maintenance (ICSM'05)*, page 661664. IEEE Computer Society, 2005. http://grosskurth.ca/papers/browser-archevol-20060619.pdf.
12. Object Management Group. MDA Guide Version 1.0.1. http://www.omg.org/docs/omg/03-06-01.pdf, 2003.
13. E. Hanson and M. Hasan. Gator: An optimized discrimination network for active database rule condition testing. Technical report, 1993.
14. Antoni Ligeza. *Logical Foundations for Rule-Based Systems*, volume 11 of *Studies in Computational Intelligence*. Springer Verlag, 2nd edition edition, 2006.
15. D. Miranker. Treat: A better match algorithm for AI production systems. In *Proceedings of the AAAI'87 Conference*, 1987.
16. Emilian Pascalau and Adrian Giurca. Towards enabling SaaS for Business Rules. In *Business Process, Services Computing and Intelligent Service*, pages 207–222, 2009. http://bpt.hpi.uni-potsdam.de/pub/Public/EmilianPascalau/ism2009.pdf.
17. Adrian Paschke. Design Patterns for Complex Event Processing. *CoRR*, abs/0806.1100, 2008.
18. Mark Proctor, Michael Neale, Michael Frandsen, Sam Griffith Jr., Edson Tirelli, Fernando Meyer, and Kris Verlaenen. Drools 4.0.7. http://downloads.jboss.com/drools/docs/4.0.7.19894.GA/html_single/index.html.
19. Marco Seiriö and Mikael Berndtsson. Design and Implementation of an ECA Rule Markup Language. In *RuleML*, pages 98–112, 2005.