Supporting ECA rule markup in ruleCore

Mikael Berndtsson[1] and Marco Seiriö[2]

[1] University of Skövde, Sweden
mikael.berndtsson@his.se
http://www.his.se/berk
[2] Analog Software, Sweden
marco@analog.se
http://www.rulecore.com

1 Introduction

Event Condition Action (ECA) rules were first proposed in the late 1980s and extensively explored during the 1990s within the active database community for monitoring state changes in database systems [11, 10, 14]. Briefly, ECA rules have the following semantics: when an event occurs, evaluate a condition, and if the condition is satisfied then execute an action. Recently, the concept of ECA rules have been transferred to the Web community for supporting ECA rules for XML data, see [2] for an overview.

We see a great need for an ECA rule markup language, in terms of having a suitable format for exchanging ECA rules between different applications and platforms. The need for an ECA rule markup language has also been identified elsewhere, e.g. [1]:

> " ... ECA rules themselves must be represented as data in the (Semantic) Web. This need calls for a (XML) Markup Language of ECA Rules."

Existing work on ECA rule markup languages is still very much in the initial phase, for example, the RuleML [4] standardization initiative has no markup for events, only for conditions and actions. In addition related work, e.g., [3, 5], on ECA rules for XML usually has an XML programming style for specifying ECA rules, rather than specifying ECA rules in a markup language.

We approach the above need for an ECA rule markup language and the lack of markup for events in existing literature by presenting the design and implementation of the ECA rule engine ruleCore[3] [12] and the ruleCore Markup Language (rCML)[4]. The rCML Language has a clear separation between specification of the three different parts (event, condition, action). In addition, it also supports specification of an extensive set of composite events.

[3] ruleCore is a registered trademark of MS Analog Software kb
[4] rCML is a trademark of MS Analog Software kb

2 RuleCore

RuleCore [12] is implemented in Python, Qt, XML, and it supports ECA rules and event monitoring in heterogeneous environments. For example, a broker system can be used to integrate heterogeneous systems, and ruleCore can be attached to such a broker system and react to events that are sent through the broker.

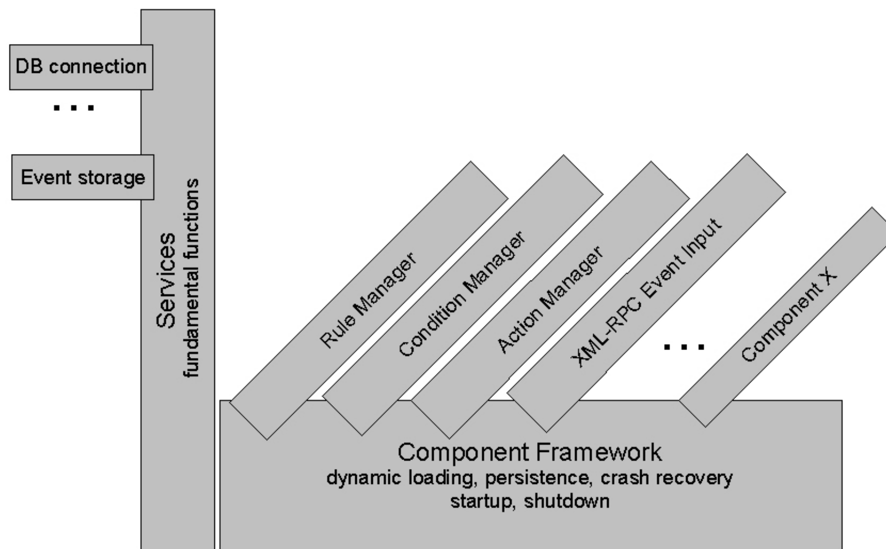

Fig. 1. ruleCore architecture

The ruleCore engine is built around a concept of loosely coupled components and is internally event driven. Components communicate indirectly using events and the publish/subscribe event passing model. The functionality of the ECA rules are provided by a number of components working in concert, where each component provides the functionality in a well defined small area. As the components are not aware of the recipient of the event they publish, it is easy to reconfigure the engine to experiment with other models besides the more well known ECA model. For example, one could insert an additional processing step between any of the event, condition or action steps. All internal and external events are stored in a relational database (PostgreSQL). Storing the event occurrences in a database implies that traditional database tools can be used for off-line analysis, visualization, simulation and reporting.

At the core of ruleCore lies a component framework, see Figure 1. The framework provides services for loading, initializing, starting and stopping components. It also handles persistence for the components and manages automatic crash recovery for the engine. The crash recovery mechanism is fully automatic and restores the last known state of the engine at startup in case the engine was not shut down properly. The recovery mechanism uses the transaction management features of the PostgreSQL database to roll forward transactions to keep the internal state of the engine consistent at all times. The temporal features of the engine are fully integrated into the recovery process and thus all time dependencies are managed in a best effort manner even in case of engine failure or down time.

The components that provide the functionality can broadly be divided into three groups.

- *Input components* are responsible for implementing support for a specific transport protocol and accepting events through it. Currently support exists for receiving events with XML-RPC, TCP/IP Sockets, SOAP, IBM WebSphere MQ, and TIBCO Rendezvous.
- *Rule components* provide the functionality of the rules. The rule manager, condition manager and action manager components work together using event passing to implement the ECA rule execution model.
- *Support components* provide functionality that is directly or indirectly used by other components. In this group we find components for event routing, event flow management, persistent state management and management of the configuration of the engine.

3 The ruleCore Markup Language (rCML)

3.1 Design Principles

Each rule, event, condition, or action in rCML has a unique name. This design decision is reusing previous design solutions from the active database community. In essence, it means that rules, events, conditions, and actions should be treated as first class objects. Some advantages are that, e.g., events can be related to other objects and also have attributes, and they can be added, deleted and updated as any other object. If for example, events are not treated as first class objects, the event information needs to duplicated in each ECA rule that is triggered by the event. This can quickly lead to maintenance problems.

3.2 ECA Rules

ECA rules are specified inside the <rules> element and each individual ECA rule is described with a <rule> element. A <rule> element has the following subelements:

- The <description> element contains a description of the rule. This subelement is only for user convenience and it is not used by the engine.

– The <event-ref> element contains a reference to the composite event that trigger the rule. The main target of applications for rCML are applications that react on composite events. However, a simple basic event can also act as a triggering event for a rule by constructing a composite event with only one sub event.
– The <condition-ref> contains a reference to a condition definition element <condition-def> that specifies a condition that is evaluated when the rule is triggered by its event.
– The <action-ref> contains a reference to a <action-def> element that is executed if the rule condition is evaluated to true.
– The <minus-action-ref> contains a reference to a <action-def> element that is executed if the triggering event can never be detected.
– The <instance-limit> element is used to limit the number of the rule instance for each type of rule. Possible values are:
  1. None
  2. An integer specifying the maximum number of rule instances.

Below is an example of an ECA rule in rCML:

```
<rules>
    <rule parameter=’append’ create=’single’ name=’Rule1’>
        <description>A description of this rule</description>
        <event-ref enabled=’yes’>E1</event-ref>
        <condition-ref enabled=’yes’>Condition12</condition-ref>
        <action-ref enabled=’yes’>Action1</action-ref>
        <minus-action-ref enabled=’yes’>Action2</minus-action-ref>
        <instance-limit>None</instance-limit>
    </rule>
</rules>
```

The <event-ref>, <condition-ref> and <action-ref> and <minus-action-ref> all contain an attribute called *enabled* with the possible values of *yes* or *no*. Thus a rule whose rule condition should always be evaluates to true is specified as <condition-ref enabled=’no’></condition-ref>.

3.3 Event Types

Two different event types are supported in rCML: basic events and composite events. A simple composite event can be defined by using two basic events. However, more complex composite events are defined by building composite events out of other composite events.

The following event operators are supported:

– Conjunction. Similar event operators are supported in Snoop [7], Ode [9], and SAMOS [8].
– Disjunction. Similar event operators are supported in Snoop [7], Ode [9], and SAMOS [8].

- Sequence. The sequence operator is perhaps most useful when the sub-events are basic events. Similar event operators are supported in Snoop [7], Ode [9], and SAMOS [8].
- Prior sequence. The prior sequence operator behaves like the sequence operator when all of its sub events are basic events. However, when the sub events are composite events the semantics of the composite event detection are as follow. The terminating event in a sub-composite event must occur before the terminating event in the following sub-composite event occurs. The semantics of the prior sequence operator in rCML is similar to the semantics of the prior event operator as defined in Ode [9].
- Relative sequence. The relative sequence operator behaves like the prior sequence operator and sequence operator when all of its sub events are basic events. The relative sequence operator requires that the terminating event in a sub-composite event is detected before the detection of the initiating event in the following sub-composite event. The semantics of the relative sequence operator in rCML is similar to the semantics of the relative event operator as defined in Ode [9].
- Any. A similar event operator is found in Snoop [7]. The any event operator will detect when any m events out of the n specified sub events have occurred, where $m \leq n$. The order of detection of the sub events is not important. Thus, the semantics of the <any> element is similar to the <and> element, but with the difference that the user can choose that only a limited number of the sub events need to be detected.
- Between. The between event operator uses one initiating event and one terminating event to detect the composite event. Any number of events can occur between the initiating event and the terminating event. All the events that occur between the initiating event and the terminating event can be stored for condition evaluation. The between event operator is usable when the initiating and terminating events of a composite event are known but not how many events that will occur in between them.
- Not. The classical semantics of the NOT operator when specifying composite events for ECA rules are that an event E is not detected during an interval specified by two events. For example, a composite event NOT E3 (E1,E2) is detected if event E3 is not detected between the detection of E1 and E2. Previous systems [6, 8] have restricted the use of the NOT operator to: (i) a conjunction [6], i.e., event E3 should not occur between (E1 and E2), or ii) a time interval [8], i.e., event E3 should not occur between 18:00 and 20:00. The approach taken in rCML generalize the usage of the NOT operator to any type of event interval. Thus, the NOT operator extends previous usage of the NOT operator for specifying composite events for ECA rules.
- Count. The count event operator is used to count how many times its only sub event is detected within an interval. The interval is configured in such a way that the count operator knows when it should start and stop counting event occurrences. Thus, a <count> element is either in an open state (counting) or in a closed state (not counting).

- Timeport. The time event operator supports specification of absolute, relative, and periodic time events. Similar events have been proposed by the active database community.
- State gate. The state gate operator can be used to detect whether an object is in a particular state, for example, between 12:00 and 13:00 the object is in the "LUNCH" state.

4 The Future

The rule engine is currently being extended to support dynamic ECA rule management, i.e., adding ECA rules on the fly. This is a feature that is mandatory, if we are to support ECA rules that integrate parts from different ECA systems. For additional details about ruleCore and rCML, please see [12, 13].

References

1. J. J. Alferes, R. Amador, and W. May. A general language for Evolution and Reactivity in the Semantic Web. In *Proceedings of the 3rd Workshop on Principles and Practice of Semantic Web Reasoning*, 2005.
2. J. J. Alferes, J. Bailey, M. Berndtsson, F. Bry, J. Dietrich, A. Kozlenkov, W. May, P. L. Patranjan, A. Pinto, M. Schroeder, and G. Wagner. State-of-the-art on Evolution and Reactivity. Technical Report REWERSE deliverable I5-D1, 2004.
3. J. Bailey, A. Poulovassilis, and P. T. Wood. An Event Condition Action Language for XML. In *Proceedings of WWW'2002*, pages 486–495, 2002.
4. H. Boley, B. Grosof, M. Sintek, S. Tabet, and G. Wagner. RuleML Design. RuleML Initiative, http://www.ruleml.org/, 2002.
5. F. Bry and P.-L. Patranjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. In *Proceedings of the 20th Annual ACM Symposium on Applied Computing SAC'2005*, 2005.
6. S. Chakravarthy, E. Anwar, L. Maugis, and D. Mishra. Design of Sentinel: An Object-Oriented DBMS with Event-Based Rules. *Information and Software Technology*, 36(9):559–568, 1994.
7. S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. K. Kim. Composite Events for Active Databases: Semantics Contexts and Detection. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 606–617, September 1994.
8. K. R. Dittrich, H. Fritschi, S. Gatziu, A. Geppert, and A. Vaduva. SAMOS in hindsight: experiences in building an active object-oriented DBMS. *Information Systems*, 28(5):369–392, July 2003.
9. N. Gehani, H. V. Jagadish, and O. Smueli. Event specification in an active object-oriented database. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 81–90, 1992.
10. N. W. Paton, editor. *Active Rules in Database Systems*. Monographs in Computer Science. Springer, 1999. ISBN 0-387-98529-8.
11. N. W. Paton and O. Diaz. Active Database Systems. *ACM Computing Surveys*, 31(1):63–103, 1999.
12. ruleCore. The ruleCore home page: http://www.rulecore.com/.

13. M. Seiriö and M. Berndtsson. Design and Implementation of an ECA Rule Markup Language. In *Proceedings of the 4th International Conference on Rules and Rule Markup Languages for the Semantic Web (RuleML-2005)*, volume 3791 of *Lecture Notes in Computer Science*, pages 98–112. Springer, 2005. ISBN 3-540-29922-X.
14. J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996. ISBN 1-55860-304-2.