RDL: A Rule Definition Language for Specifying the Behavior of Distributed Systems

Edwin Moulier-Santiago and Jaime Yeckle-Sánchez
Advisor: Javier Arroyo-Figueroa

Electrical and Computer Engineering Department
University of Puerto Rico, Mayagüez Campus
Mayagüez, Puerto Rico 00681-5000
emoulier@yahoo.com

Abstract

This paper presents RDL, a Rule Definition Language to specify the behavior of distributed systems. RDL allows to specify such behavior in terms of how distributed components react to the occurrence of events. The computational model and syntax of RDL are presented. The use of RDL in preliminary tests shows that it takes 1/6 of the number of lines in a Java program to capture the same semantics.

1. Introduction

A distributed system (DS) consists of a set of processes or services executing in dissimilar platforms that communicate via message passing. Architectures like CORBA and RMI provide mechanisms to develop DS, but lack of enough abstractions to specify the behavior of the system. Instead, the behavior is defined by means of methods or functions that contain some algorithm whose behavior is hidden from the clients. The only way to know about behavior is examining the program code or documentation. The understanding of the behavior turns complicated, while the amount of code increases; especially when more than one person develops the application. One alternative to this problem is to specify the behavior of the system in terms of how is components react due the occurrence of events within. Events are occurrences of interest that contain relevant information about changes in components of the system. Such events trigger behavioral changes in one or more components of the entire system. Event-Condition-Action-AlternativeAction (ECEA A) rules [Arroyo99] can be used to specify such behavior.

This paper presents RDL, a Rule Definition Language to specify the behavior of DS in terms of how distributed components react to the occurrence of events in a system.

In RDL, a rule can be viewed as an algorithm that is executed due to the occurrence of events. The rule specifies the actions or alternative actions to be taken if the necessary events occur. How and when a rule is evaluated are issues of the computational model of the RDL, which will be presented in this paper.

The rest of this paper is organized as follows. In the next section, a comparison with related works is made. in Section 3, we present the computational model of RDL. Section 4 presents an example of the use of RDL to specify the behavior of a distributed system. Our conclusions are presented in Section 5.

2. Related Works

In Bates' work [Bates95], an event description language (EDL) is used to define simple and composite events, and behavioral models of different scenarios of a distributed system. Contrary to RDL, it does not have an object-oriented model.

Mansouri and Sloman [Mansouri96] present a language called GEM that is used to define events and rules for event handling. Many of different types of composite events are supported by a set of predefined operators. Temporal constraints are also supported. Contrary to RD, events and rules are not treated uniformly as objects.

3. Computational Model

The computational model of RDL has the following elements: (i) an event model, which describes what are events and their types; (ii) a rule model, which describes the role and structure of rules; and (iii) a behavioral model, which describes how rules react to the occurrence of events. In the following subsections, we shall describe these elements.

3.1 Event Model

In RDL, all events are treated as objects. Events of the same type are defined by an *event type* and are represented by an *event class*. The class Event is the base class of all the event classes and defines the common structures of all events type. A Java specification of the class Event is presented in Figure 1. This class defines attributes such as *eid*, which represents a unique identifier for each event object; the *daytime* attribute, which defines the time, relative to the day, when the event is produced; *ttl, which* defines how much time the event object will be alive; and *producer*, which identifies the distributed object that produced the event. It also defines methods such as: *t(), which* returns the value of the attribute *daytime* mentioned above; *ts(),m* which returns the time the event have in the event set; *getProducerClassName()*, which returns the name of the distributed object that produce the event,; *getProducer(), which* returns the distributed object that produce the event; and *isDead()* , which returns true if the event has past its own *ttl* value.

```
public class Event
{
 / ***** Event attributes ******/
 // event identifier
 public String eid;
 // when event was produced
 public Timevalue daytime;
 // time-to-live in set
 public TimeValue ttl;
 // who produced the event
 public DistributedObject producer;

 /****** Event Methods ******/
 // return the time the event was
produced
 public Timevalue t() { … }
 // return the amount of time in the set
 public TimeValue ts() { … }
 // get name of class of event  producer
 public String getProducerClassName() {
… }
 // get the producer object of the event
 public DistributedObject
getProducer(){…}
// return true if event is dead
 public boolean isDead() { … }
}
```

Figure 1.
Java definition of the class Event

In this event model, only simple events are supported. The definition of composite event expressions is part of the rule model of RDL described in the next section.

Figure 2 shows the specification of an event type class named *GageLevelReport.* In addition to the attributed inherited from the *Event* class, this class defines attributes like *loc* that identify the location of the event produced and *level* that represents a measure of some type of measure instrument like water gages. Events of this class will be used to illustrate some examples of rule evaluation.

```
// Class GageLevelReport definition
package aaa;

public class GageLevelReport extends
Event
{
   double level; //gage water level
   int loc; //gage location}
}
```

Figure 2.
Example of an event type specification

3.2 Rule Model

A rule can be viewed as an algorithm that is executed due to the occurrence of one or more events. When such algorithm is executed, it is said that the rule is *evaluated*. A rule is evaluated when it is *triggered* by the occurrence of one or more events. A rule can be in one of two possible states: *active* or *inactive.* Active rules can be triggered, while inactive ones cannot be triggered. The number of times a rule is triggered is defined by the behavioral model, which will be presented in the following section.

A priority number can be assigned to each rule to specify the order of triggering with respect to the others.
Rules are triggered with the same priority can be triggered in parallel; rules with lower priority have to wait for the end of the evaluation of rules with higher priority before they can be triggered.

The *event pattern* of a rule defines the types of events whose occurrence may trigger the rule. When there exists a set of events that satisfies the event pattern, the rule is triggered. The semantics of an event pattern are explained by the behavioral model, which is described in the next section.

When evaluated, a set of *actions* are performed in an execution path. An action can be an invocation of a method on a distributed object or the posting of a new

event. Three possible execution paths can be defined for each rule. A *conditional path* is defined when a condition is given, which must be satisfied in order to execution a given set of actions. An *alternative conditional path* is defined when a set of actions is specified and are to be executed if a given condition is not satis fied. An *unconditional path* is defined if a set of actions are specified to be performed unconditionally. Conditional and alternative conditional paths take precedence over unconditional paths.

In RDL, rules can be grouped into packages. A package in RDL is similar to a Java package, used to group together a set of logically related elements; in RDL, packages are used to group together a set of logically related events and rules.

3.3 Behavioral Model

The behavioral model of RDL defines how rules are triggered and evaluated upon the occurrence of events, and how many times they are triggered with respect to the number of events.

An event is said to be *alive* if it has been created (posted) and its time-to-live (ttl) has not been reached. The time-to-live of an event can be specified for each event; otherwise a default value is given to this attribute. Events that are alive are kept into an *event set;* events that are dead (not alive) are removed from the set as soon as they reach their time-to-live.

A *rule evaluation cycle* occurs when all the rules that are active have been considered for triggering. The *rule evaluation cycle interval* is a system-defined time between evaluation cycles. Its value depends on the type of applications to be supported by the distributed system. For real-time systems, the interval might be in the order of milliseconds.

The order of triggering is determined by each rule's priority, as explained in the previous section. A rule is triggered if there exists a set of events that satisfy its event pattern. An event pattern can be formally defined by an n-tuple $< X_1, X_2, \ldots, X_n >$, where each X_i is an event type expression, and n is the number of event types specified in the pattern. An event type expression can be one of the following: (i) the name of an event type, or E_i; (ii) a negation operator (!) preceding the event type, or $!(E_i)$; or (iii) an event set constructor operation, or $\{E_i [c] \}$, where c is a Boolean condition that must be satisfied by an event of type Ei to belong to the set.

An *events combination* is defined by an n-tuple $<e_1,e_2, \ldots, e_n>$, where each e_i is an event in the event set and is of the same type of the corresponding E_i in the pattern. A special event is the *null event* (∅) which can be of any type.

An event combination *satisfies* an event pattern if, for each e_i, one of the following two conditions is met: (i) $X_i = E_i$ or $X_i = \{ E_i \}$; or (ii) $e_i = \varnothing$ and $X_i = !(E_i)$.

A rule is said to have *consumed* an event combination if any action has been taken after being triggered by the combination.

With these definitions done, we are ready to define the number of times a rule is triggered in each evaluation cycle. In each evaluation cycle, each rule is evaluated once for each event combination satisfied by its event pattern, if it has not consumed the event combination.

4. Rule Syntax

The components of the rule language syntax, presented in Figure 3, are the following:

The **package_specification** clause is used to specify the package to which the rule belongs. A package clause can be followed by many rule specifications, and it assumed that all the following rules belong to the same package.

The **rule_id** clause is the unique identifier of the rule (unique within the package).

The **priority_no** clause is used for the execution paradigm of rules, such that semantics are captured correctly by executing rules in the appropriate order. Can be zero (the default) or a positive number. The lower the number, the highest the priority.

The **trigger_events** clause is an expression composed y a single event (simple event expression) or set of events (composite event expression) that will trigger the execution of the rule. A simple event triggers the rule upon its occurrence. Composite events trigger the rule depending on the relationships specified among events (e.g., one event along with others, one event but not others, time-related). Complex event patterns are specified by operators, such as "&&" (and), "||" (or), "!" (not, or absence of an event of a given type) and ">>" (causality, an event occurs after the other). A special type of operator is the set constructor ({}) which creates a subset of all the events of a given type that satisfy a condition.

The **usage_specification** clause is used by rules that need to make use of services of an existing DS component. The usage clause of a rule specifies which

services are to be used in evaluating a condition or performing an action.

```
[ package <package_specification>]
rule <rule_id>
[ priority <priority_no> ]
on <trigger_events>
[ use <usage_specification> ]
[ if <condition>
then <actions>
[ else <alternative_actions>] ]
[do <actions>]
```

Figure 3. Syntax of the rule language

The **condition** clause specified in a rule, must be satisfied ("true") to execute the rule upon the occurrence of events. Rule conditions may include time relationships among events (e.g., one event before another, one event 5 minutes after the other).
The **action** clause consist of a list of operations to be performed if the trigger events occur and the condition is satisfied.

The list of actions or alternative actions can be either an invocation of a service, an invocation of a method of an event or the posting of a new event (with the **post** operator).

The **alternative_actions** clause consist of a list of operations can be specified to be performed when the rule is triggered but the rule condition is not satisfied.

In an experiment realized, a comparison between RDL and Java was performed. The results of the experiment shows that the implementation of a set of algorithms using Java produce almost 6 times the code produced using RDL to implement the same set of algorithms.

5. Examples

Figure 4 shows three different levels of complexity for rule definition. The first statement is the *package* declaration that state that the subsequent rules belongs to this package. In ***rule eip1***, the rule is triggered when an event of type GageLevelReport occurs after another event of the same type. If this occur, the *if* condition is evaluated and if the time difference between the two events is equal or less 15 minutes, the level difference between the two events is greater or equal than 0.75 inches, and the events come from the same location; an event of the class type *EventInProcess* is posted, with its location ("loc") with the same value as the location of the gages.

The ***rule eip2*** use the event-set constructor {}, to define a set of events. In this example, two sets, S1 and S2, are created. A set is constructed of events that satisfy the condition placed between square brackets. The set S1 has events of the type GageLevelReport that has been in the set in the last 15 minutes, whose level is between 12 and 15 inches, and whose location is the same as in the event "glr1". The set S2 contains all the events that have occurred in the last 15 minutes and are in the same location as event "glr1". Upon such occurrences, the "if" clause checks if more than 50% of the events in S2 are in event S1; if this condition is true, then a new EventInProcess event for the same location is posted.

In the ***rule*** aeenotify, a rule is defined in a different package called "aee". This rule uses the "use" clause to use the AEENotification distributed service. Upon the occurrence of an EventInProcess event called "eip", if the location of the event is CARRAIZO, then the "notify" method of the AEENotification service is

```
package aaa;

rule eip1
on GageLevelReport glr1 >>
   GageLevelReport glr2
if (glr2.t( ) -  glr1.t( ) <= 15:00) &&
    (glr2.level - glr1.level  >= 0.75)
&&
     (glr2.loc == glr1.loc)
then
   post EventInProcess {loc=glr1.loc};
end;

rule eip2
on  GageLevelReport glr1   &&
{GageLevelReport [ (ts() <= 15:00) &&
                   (level >= 12.0) &&
                   (level <= 15.0) &&
                   (loc == glr1.loc] } S1
          &&
 {GageLevelReport [ (ts() <= 15:00) &&
                    (loc == glr.loc } S2
if   (S1.size() / S2.size()  > 0.5)
then
  post EventInProcess {loc=glr1.loc};
end;

package aee;
import aaa;
rule aeenotify
use AEENotification aeen
on EventInProcess eip
if eip.loc=CARRAIZO
do
  aeen.notify("CARRAIZO alert");
end;
```

Figure 4.
Examples of rule definitions in RDL

invoked.

An experiment was made to compare a set of rules in RDL with an equivalent Java implementation. The results of the experiment show that the Java implementation required almost 6 times the number of lines of code required in RDL.

5. Conclusions

In this paper, we have presented RDL, a Rule Definition Language to specify the behavior of distributed systems. RDL has the capability of constructing complex statements with a high-level of abstraction. The results preliminary tests show that RDL requires less lines of code to express the same semantics as an equivalent Java implementation.

6. References

[Arroyo99] Arroyo-Figueroa, J.A., Borges, J.A., Rodriguez, N.J., "ERF: An Event-Rule Framework for Supporting Heterogeneous Distributed Systems", A proposal submitted to the National Science Foundation, 1999. Department of Electrical and Computer Engineering, University of Puerto Rico, Mayagüez Campus.

[Bates95] Bates, P. "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior", ACM Transactions on Computer Systems, vol. 13, no. 1, 1995, pp. 1 – 31.

[Mansouri96] Mansouri, S. and Sloman, M., "A Configurable Event Service for Distributed Systems", Proc. of the 3rd Int'l. Conference on Configurable Distributed Systems, Annapolis, MD, 1996 pp. 210-217

[Moulier01] Moulier, Edwin, Arroyo, J., "**A Rule-Based Intelligent Event Service (RUBIES)**", Proceedings of the Computing Research Conference 2001, University of Puerto Rico, Mayagüez Campus, March 2001.