UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.dit.unitn.it

A JAVA IMPLEMENTATION OF COORDINATION RULES AS ECA RULES

K M Senthil Kumar

June 2003

Technical Report # DIT-03-037

A Java implementation of Coordination Rules as ECA Rules

K M SENTHIL KUMAR

DIT – Department of Information and Communication Technology

University of Trento, 38050 Povo, Trento, Italy

senthil@dit.unitn.it

Abstract: This paper gives an insight in to the design and implementation of the coordination rules as ECA rules. The language specifications of the ECA rules were designed and the corresponding implementation of the same using JAVA as been partially done. The paper also hints about the future work in this area which deals with embedding this code in JXTA, thus enabling to form a P2P layer with JXTA as the back bone.

1 Introduction

Peer-to-peer (P2P) computing consists of an open-ended network of distributed computational peers, where each peer can exchange data and services with a set of other peers called acquaintances. Peers are fully autonomous in choosing their acquaintances. Moreover we assume that there is no global control in the form of global registry, global services, or global resource management, nor a global schema or data repository. P2P offers an evolving architecture where peers come and go, choose whom they deal with, and enjoy some traditional distributed service with less start-up cost. Since the data residing in different databases may have semantic inter – dependencies, we allow peers to specify coordination formulas that explains how the data in one peer must relate to data in acquaintances. Coordination formulas may also act as soft constraints or guide the propagation of updates. In addition peers need an acquaintance initialization protocol where two peers exchange views of their respective databases and agree on levels of coordination between them. The level of coordination should be dynamic, in the sense that acquaintances may start with little coordination, strengthen it over time with more coordination formulas and eventually abandon it when tasks and interests change.

In such a dynamic setting we cannot assume the existence of a global schema for all databases in a P2P network, or even those of all acquainted databases. Moreover peers should be able to establish and evolve acquaintances, preferably with little human intervention. The Local Relational Model [1] (LRM) was introduced as a data model specifically designed for P2P applications. LRM assumes that the set of all data in a peer to peer network consists of local (relational) databases each with a set of acquaintances, which define the P2P network topology. For each acquaintance link, domain relations define translations rules between data items and coordination formulas define systematic dependencies between the two databases. Coordination formulas can be implemented by coordination rules.

The main goal of this paper is to design the semantics of the coordination rules and to implement them using JAVA .This work forms the part of the C2C project [2].This work will be appended to the work done on Query proceesing in peer to peer network [3].This paper had been constructed entirely from the thesis work done by Vasiliki Kantere [4].

The structure of the paper is organized as follows. Section 2 gives a brief introduction to active databases. Section 3 gives the fundamentals of ECA rules. Section 4 defines the semantics of the ECA rule language. Section 5 defines the execution semantics of the rules. Section 6 talks about the underlying architecture and the code structure. Section 7 reports about the conclusions and future work

2 Active Databases

Traditional database management are passive in the sense that commands are executed by the database (e.g., query, update, delete) as and when requested by the user or application program. However some situations cannot be effectively modelled by this pattern. As an example, consider a railway database where data are stored about trains, timetables, seats, fares and so on, which is accessed by different terminals. In some circumstances it may be beneficial to add additional coaches to specific trains if the number of spare seats a month in advance is below a threshold value. Two options are available to the administrator of a passive database system who is seeking to support this requirement. One is to add the additional monitoring functionality to all booking programs so that the preceding situation is checked each time a seat is sold. However, this approach leads to the semantics of the monitoring task being distributed, replicated, and hidden among different application programs. The second approach relies on a polling mechanism that periodically checks the number of seats available. Unlike the first approach, here the semantics of the application is represented in a single place, but the difficulty stems from ascertaining the most appropriate polling frequency. If too high there is a cost penalty. If too low, the reaction may be too late (e.g. the coach is added, but only after several customers have been turned away).

Active databases support the preceding application by moving the reactive behaviour from the application (or polling mechanism) into the DBMS. Active databases are thus able to monitor and react to specific circumstances of relevance to an application. The reactive mechanism is both centralized and handled in a timely manner.

Active databases, as opposed to ‘passive’ ones, are able to recognize specific situations in the database where they react automatically, without an explicit external request. In passive databases two kinds of integrity constraints are supported: *key constraints*, which restrict some data so that their values are unique and *referential constraints*, which require that data references should reference existing data items. These constraints are checked immediately after a database operation and, in case of violation, they roll back the transaction where the operation occurs. However, often there is a necessity to support other kinds of constraints. Such constraints maybe complicated and usually depend on the type of data stored. Active databases offer a mechanism for the specification and monitoring of such constraints. An active DBMS provides a mechanism for the declaration of rules, often referred to as the *knowledge model* or the *rule language*, and a mechanism for the execution of the rules, often referred to as the *execution model* or *rule execution semantics* [6].

A common approach for the knowledge model uses rules that have up to three components: an event, a condition, and an action. Most active database systems support rules with all three of the component described; such a rule is known as an event-condition-action or *ECA rule*.

3 ECA Rules

Generally rules are comprised of three parts: and *event*, a *condition* and an *action*. The event is the happening that causes the rule to fire. Events can be simple or composite. Composite events are formed from simple ones with the help of an event language. The latter comprises operators with which events (simple and composite) can be combined. The condition of the rule checks the state of the databases at the time of when the rule event happens. A condition is usually expressed as a predicate defined in terms of the condition clause of the database query or part thereof. In this case, the result of the query condition determines if the condition holds. Finally, a condition can be a user defined Boolean function. The action of the rule is the task that is executed when the rule event occurs and the rule condition holds. The action can involve database operations, transactions, or user-defined functions.

4 Rule Language

The basic database notations that are used in the rule language are

Db_Action = Any SQL code (create, delete, update, insert, abort or commit)

Relation_Name$_J$ = Name of the relation in a database J

SOP = {P_1, P_2, P_3…..P_N} where P_1, P_2, P_3…..P_N are the name of attributes or the attribute values of the specific relation

4.1 Rule

The rules are of two types: rules that consists of all three parts (the condition part is optional), known as ECA rules(Event-Condition-Action) rules; the second type are rules with only a condition and an action part, known rules(Condition- Action) or *production* rules.[5]

ECA Rules	Production Rules
Where Event (E_1, E_2, E_3 ……E_N)	*If* Condition
If Condition (C_1, C_2 ….C_K)	*Then* Action
Then Action (A_1, A2, A_3……..A_M)	

The major difference between the two types is that in the first a rule is triggered because of the occurrence of a specific event, whereas in the second a rule is triggered when the database reaches a specific state for which it is periodically checked. Thus, the triggering of the rule in the second case depends solely on the state of the database, rather than the occurrence of external events. Another difference is that several ECA rules with different conditions can be declared to have the same event part, whereas for a CA rules a specific condition can trigger only one rule.

4.2 Event

An event is something that happens at a point in time. Specifying an event therefore involves providing a description of the happening that is to be monitored. The nature of the description and the way in which the event can be detected largely depends on the source or generator of the event. Possible alternative for the sources are:

- Structural operations, in which case the event is raised by an operation on piece of structure
- Behaviour invocation, in which case the event is raised by the execution of some user- defined operation
- Transaction, in which case the event is raised by transaction commands
- Abstract or user-defined, in which case a programming mechanism is used that allows an application program to signal the occurrence of an event explicitly

- Exception, in which case the event is raised as a result of some exception being produced
- Clock, in which case the event is raised at some point in time
- External, in which case the event is raised by a happening outside the database.

Generally, an event expression is either simple or composite. Composite event expressions are formed by applying the operators of event algebra on simple or composite events. So generally an event is of the form:

Event = SE | CE (Simple event or Composite event)

- Simple, in which case the event is raised by a single low-level occurrence that belongs to one of the categories described in source
- Composite, in which case the event is raised by some combination of simple or composite events using a range of operators that constitute the event algebra

We define the following set of parameters:

Definition:

T = (Sec, Min, Hrs, Day, Weekday, Month, Year)

where the allowable sets of values for each component of the T tuple are:

Sec € {0, 1, 2…59}
Min€ {0, 1, 2…59}
Hrs € {0, 1, 2…23}
Day € {1, 2 …31}
Weekday € {Mon, Tue, Wed, Thu, Fri, Sat, Sun}
Month € {1, 2...12}
Year € N

Generally, a simple event SE starts at a time point T_S and ends at a time point T_E. Assume that the event happens instantaneously (i.e.) not much of time is consumed by the occurrence of the event.

T_S – Start of the Event

T_E – End time of the Event $\quad\quad T_S \approx T_E$

4.2.1 Periodic and Non periodic Events

The time factor of the events can be expressed in the following way:

t = (T1, T2, Boolean)

If the value of Boolean is true then it is a periodic Event and if it is false, it is not.

t = (T1, T2, True)

The event occurs at T1 and at T1 + N* T2 where N € Z_+

t = (T1, - , False) It is not a periodic event

The event occurs at T1

If the Boolean value is false and both T1 and T2 are mentioned then it defines a time interval

t = (T1, T2, False)

Denotes a time interval starting at T1 and ending at T2: t € [T1, T2]

Hence a simple event can be denoted as

Simple Event (SE) ➔

SE = (T) (Event descriptor) where t = (T1, T2, Boolean)

Event Descriptor = (Qid, Db_Action, Relation_ Name_J, SOP)

Where Qid is the Query Id

Hence a composite event can be denoted as

Composite Event (CE) ➔

A composite event CE is a combination of simple events. This combination is derived by applying the set of operators of event algebra on simple or other composite events.

The general form of a composite event is:

$CE = SE_1 (op_1) SE_2 (op_2)\ldots\ldots\ldots SE_i(op_i)$

Where op_i represents the operators of the event algebra.

The operators can be either unary or binary.

Some of the operators defined are:

Table 1: Operators of the event algebra

Operator	Type	Function	Syntax
$\cap_t$	Binary	Logical AND	$<Event1>\cap_t<Event2>$
U_t	Binary	Logical OR	$<Event1>U_t<Event2>$
$!_t$	Unary	Logical NOT	$!_t <Event1>$
$*_t$	Unary	Zero or more occurrences	$*_t <Event1>$
$+_t$	Unary	One or more Occurrences	$+_t <Event1>$
$_m\#_t$	Unary	Maximum number of occurrences	$_m\#_t <Event1>$
$_m\&_t$	Unary	Minimum number of occurrence	$_m\&_t <Event1>$

where t – Time interval

4.3 Condition

The condition of an ECA rule is a Boolean expression using the operators of the Boolean algebra i.e. AND, OR and NOT.

Condition = $(\text{Cond})_{\text{CONTEXT}}$

Where Cond is any normal Boolean expression .A simple condition is declared by defining the two operands and the operator. If the operands are strings, only equality and inequality operands are allowed. The operands of a simple condition are either constant are variables.

Where CONTEXT is an identifier € $\{DB_T, BIND_E, DB_E, DB_C\}$

4.4 Action

The range of tasks that can be performed by an action is specified as its options. Actions may update the structure of the database or rule set, perform some behaviour invocation within the database or an external call, inform the user or the system administrator of some situation, abort a transaction or take some alternative course of action

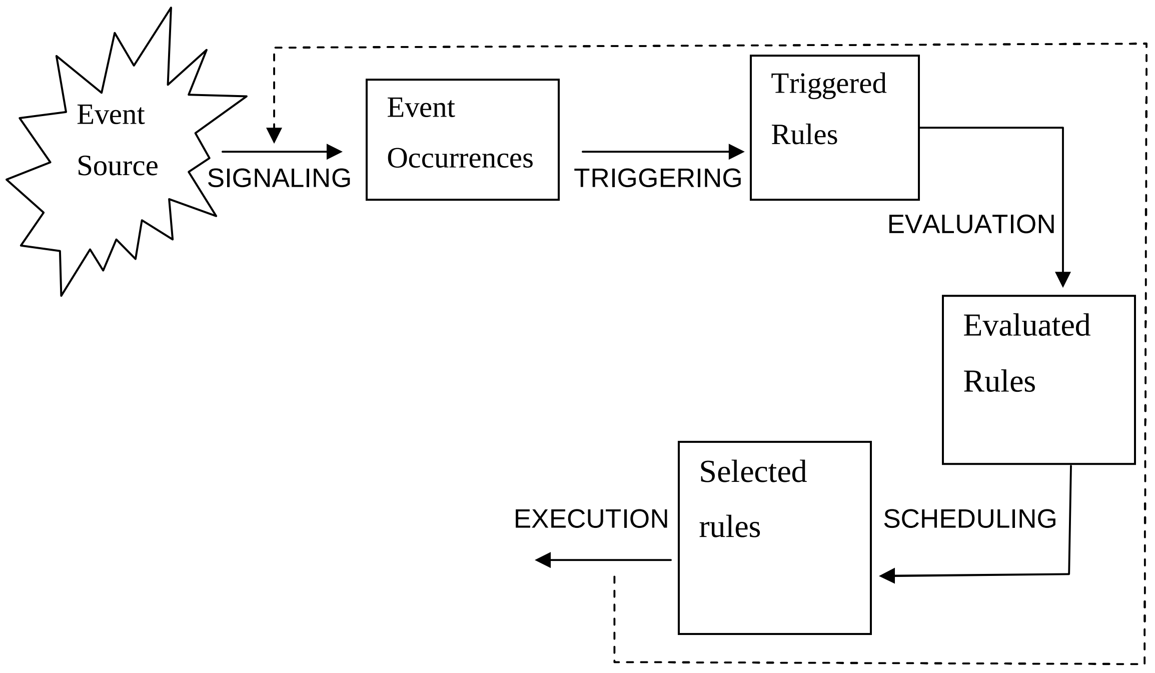
Hence action can be best defined as:

Action = Db_Action | SE | CE

5 Execution Semantics

The execution model specifies how a set of rules is treated at runtime. Although the execution model of a rule system is closely related to aspects of the underlying DBMS, there are a number of phases in rule evaluation, illustrated in Figure 1 , that transcend considerations that relate to specific software environments

Figure 1: Principle steps that take during the rule execution



1. The *signalling* phase refers to the appearance of an event occurrence caused by an event source
2. The *triggering* phase takes the events produced thus far, and triggers the corresponding rules. The association of a rule with its events occurrence forms a rule instantiation.
3. The *evaluation* phase evaluated the condition of the triggered rules. The rule conflict set is formed from all rule instantiations whose conditions are stratified.
4. The *scheduling* phase indicates how the rule set is processed
5. The *execution* phase carries out the actions of the chosen rule instantiations During action execution other events can in turn be signalled that may produce cascaded rule firing

The execution of the rule is in general defined as:

Exec (Rule) = Seq {Exec [Event (E_1, E2……..E_N)]

Exec [Condition (C_1, C_2…......C_K)]

Exec [Action (A_1, A_2………A_M)]}

Where C_1, C_3….C_K are parameters instantiated by events E_1 E_2….E_N

5.1 Rule

The execution model goes beyond the rule language and describes how the rules are evaluated at runtime. There are several issues that concern rule execution, as well as inter-relations among these issues.

5.1.1 Rule Granularity

Refers to the issue of determining how often the system runs the rule evaluation procedure and can be defined at three levels.

Granularity € {Continuous, Triggered by an Event, Triggered by any Transaction}

- Continuous: The system checks extremely frequently for the triggering of the rule. However this issue is relevant only for CA rules and ECA rules with a periodic event. For other ECA rules there is no point in checking for rules when no appropriate event has occurred
- Triggered By event: The system checks for the triggering of rule at the time of database operations.
- Triggered by Transaction :The system checks fro the triggering of rule at the end of each transaction

Exec (Rule) = Cont.Exec (Rule, T, P) | TrigByEvent.Exed (Rule, E, P)|

TrigByTrans.Exec (Rule, Tra, P) where P is the priority

Where T is the time, P is the priority, and E is the Event, Tra is the transaction

5.1.2 Rule Evaluation

Rule evaluation is either instance or set oriented. Instance oriented allows a one-to-one correspondence between the rules and events. In this a rule is triggered for each instance of an event. Set oriented allows a many to one correspondence between events and rule. In this a rule is triggered for a set of events. Not both of these techniques can be applied in all cases. It is not possible to instantiate a rule for a set of instances when the rule granularity is a simple database operation, this means that the system responds by triggering a rule after each simple database operation that matches the event part of the rule. For a set oriented execution, the net effect of the set of event occurrences is usually considered in order to trigger a rule. For example, if a tuple in a relational database is inserted and then deleted, the net effect is that there is no event occurrence at all. If a tuple is inserted and then updated, the net effect is an insertion of the updated tuple.

Table 1: Execution Semantics of an ECA Rule

Granularity ➔	Continuous	Trig by Event	Trig By Transaction
Instance	Cont.Exec(Rule ,T, P)	TrigByEvent.Exed(Rule , E, P)	TrigByTrans.Exec(Rule , Tra, P)
Set	Cont.Exec(Rule ,T ,{E1, E2 …EN}, P)	TrigByEvent.Exed(Rule , {E1, E2 …EN}, P)	TrigByTrans.Exec(Rule , Tra , {E1, E2 …EN}, P)

where {E1, E2 …EN}--- Set of Events.

Usually when the system checks for rules that should be triggered more than one rule is eligible for firing. This can happen because an event occurrence matches the event part of the several rules .When a conflict arises as to which rule to fire, the simplest solution is to choose a rule randomly. The second approach is to prioritize the set of rules that are triggered by the same event. Rule priorities can be assigned during rule creation time or at runtime.

5.2 Event

When detecting composite events, there may be several event occurrences (of the same event type) that could be used to form a composite event. Suppose that we have to evaluate the composite event CE = (E1<E2) (where '<' stands for followed by) and the event instances $E1_1 < E1_2 < E2_1$. The four possible *consumption policies* [5] used to evaluate the CE:

- Recent: The policy considers only the most recent occurrences of events. Thus the instance of CE produced is: $CE_1= (E1_2 < E2_1)$.
- Chronicle: The policy considers only the earliest occurrences. Thus the instance of CE produced is: $CE_1= (E1_1 < E2_1)$.
- Cumulative: The policy accumulates all the instances of the simple events that concern the event of the rule until the instance of the latter can be formed. Thus two instances of CE would be produced: $CE_1= (E1_1 < E2_1)$. and $CE_2 = (E1_2< E2_1)$
- Continuous: This policy starts the composition of a new composite event instance whenever a simple event instance. In this case one instance of CE would occur that would contain the parameters of both $E1_1$, $E1_2$ as well as of $E2_1$.

The *role* of the event indicates whether event must always be given for active rules, or whether the explicit naming of the event is unnecessary. If the role is *optional*, then when no event is specified condition-action rules are supported, which have significantly different functionality and implementations from event-condition-action (ECA) rules. If the role is *none* then events cannot be specified, and all rules are condition-action rules. If the role is *mandatory* then only ECA rules are supported.

5.3 Condition

The role of a condition indicates whether it must be given. In ECA-rules, the condition is generally optional. When no condition is given for an ECA rule or where the role is none, an event-action rule results. IN systems in which both the event and the condition are optional, it is always the case that at least one is given.

Role € {Mandatory, Optional, None}

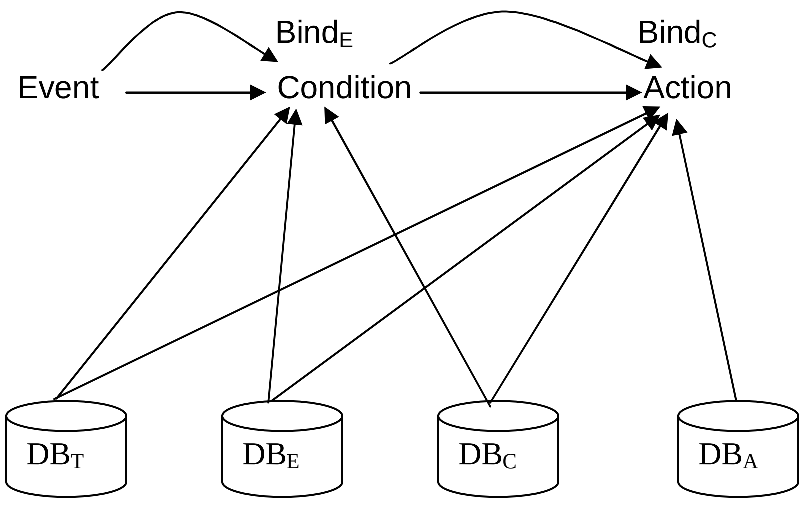
The context indicates the setting in which the condition is evaluated. The different components of the rule are not evaluated in isolation from the database or from each other, and further more they may not be evaluated in quick succession. As a result the processing of the single rule can potentially be associated with at least four different database states:

Context € {DB_T, $Bind_E$, DB_E, DB_C}

DB_T: database at the start of the current transaction
DB_E: database when the event took place
DB_C: database when the condition is evaluated
$Bind_E$: Binding associated with the event.

Active rule systems may support facilities within the condition of a rule that allow it to access zero or more of the states DB_T, DB_E, and DB_C and may also provide access to bindings associated with the event. The availability of information to different components of a rule is illustrated in Figure 2

Figure 2: The context within which a rule is processed



5.4 Action

The context of the action is similar to that of the condition, and indicates the information that is available to the action as illustrated by Figure 2. It is sometimes possible for information to be passed from the condition of a rule to its action as DB$_E$ or Bind$_C$

Action € {DB$_T$, Bind$_E$, Bind$_C$, DB$_E$, DB$_C$, DB$_A$}

DB$_T$: database at the start of the current transaction

DB$_E$: database when the event took place

DB$_C$: database when the condition is evaluated

DB$_A$: database when the action is evaluated

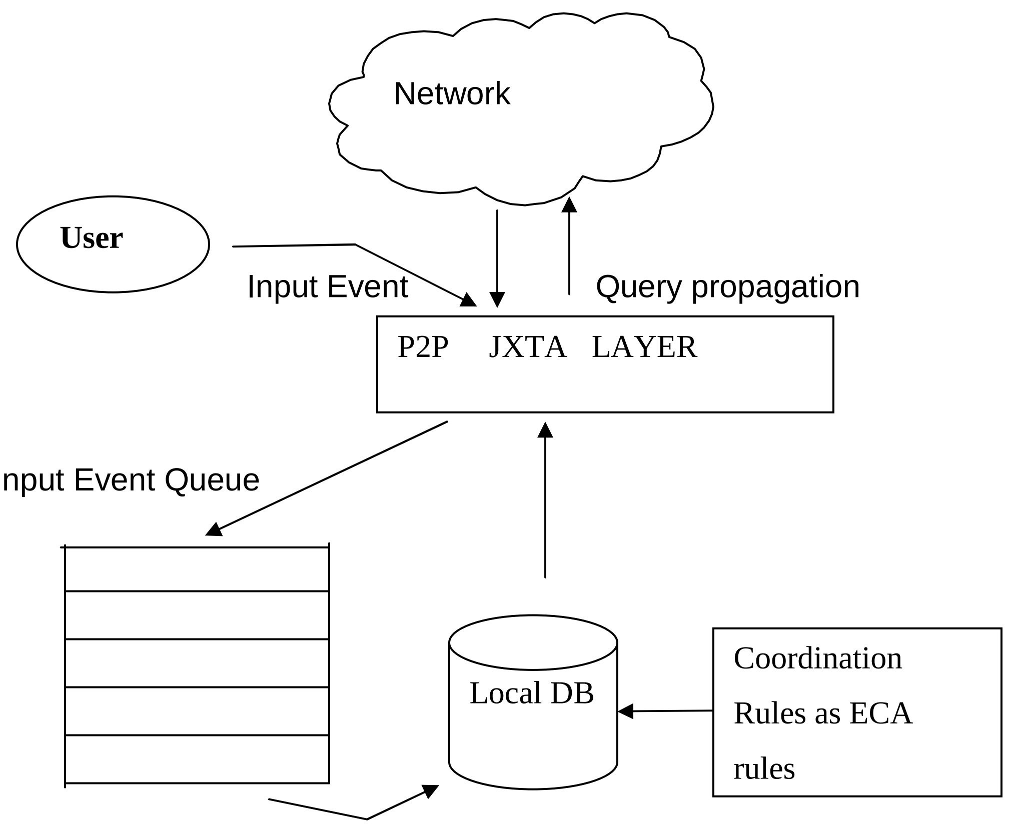
Bind$_E$: Binding associated with the event

Bind$_C$: Binding associated with the condition

6 Architecture

The P2P Layer constructed using JXTA take care of all the communication overheads required for establishing peers, and carrying out any future transmission of messages of all kinds between the peers. JXTA designs a set of protocols designated mainly for transporting and addressing space support on P2P networks. It also provides mechanism for peers and other basic resources discovery, gives well developed tools for metadata representation, communication links establishment, establishing acquaintances and so on. New results are being published in relation to the design of algorithms related to establishing and abolishing acquaintances in a Peer to Peer database network [7]. In other words, JXTA gives an instrument suite for P2P applications development of arbitrary nature.

Figure 3: Architecture



There is an input Event Queue in which any events that is pertaining to the database at that node is added. These events can be either initiated by the user at that node or the event could have come from the network. The P2P JXTA layer decides whether the query coming from the network need to be evaluated at the particular database or not. So the input event queue contains the events that have to occur on that database. When these events are executed by the DBMS at the database, the rules that get triggered by the execution of the events are evaluated. The actions that are to be executed are in turn sent to the above P2P JXTA layer. The layer again finds where exactly these actions have to be executed and propagates them to their respective destinations. If the actions are to be performed on the same database then they are again placed in the input event queue.

6.1 Procedure for Rule Execution

The code structure follows the below mentioned algorithm for the execution of the ECA rules. Initially all the events are queued up in the event queue and all the ECA rules are also stored for the database. Then the first event from the queue is taken and the corresponding ECA rule which will be triggered on its execution is found out. Once the corresponding rule or rules are found, the execution of the rule takes place by evaluating the condition first. Then if the condition evaluates to true then the corresponding actions are sent to the P2P JXTA layer.

```
Exec(Rule, Granularity, Evaluation, Priority)
{
   exec= true
   match1=false
   while (exec)
     {
        if (notempty(input _eventqueue))
            {
              event1 =  choose_event(input_event queue)
              while(not matched  or end_of_ECA rules)
```

```
                  match1= match(event1 , ECA rule)
                if(match1)
                  {
                    if(rule_conditon)
                     do_rule_action()
                     exec=false;
                  }
          }
     }
}
```

1. Create rules and insert them into the rule queue
2. Create events and insert them into the event queue
3. Take the first event and go through all the rules and see whether rule.event = event
4. Once the event is matched the entire rule is obtained from the Queue
5. Then the rule condition is evaluated
6. Once the condition evaluates to true the rule action is executed
7. Reduce the event queue size by one by removing the first even
8. Go to step three or exit

The codes are given in Appendix. The program has been written in JAVA as in future this code can be easily embedded on a JXTA platform

7 Conclusions

The semantics of the rule language for ECA rules had been initially proposed in the thesis report by Vasiliki Kantare. The language constitutes one of the first attempts to define ECA rule languages where event arise in multiple databases, and conditions and actions need to be evaluated with respect to several databases. The work was also one of the first to focus on peer-to –peer computing, where coordination rules among databases are defined dynamically at run time by end users, instead of them being defined at design time by database engineers. The semantics of the language defined, needed further refinement and alternative implementations of the rule language was necessary to minimize communication overheads between the peers.

The work that I have done is further refinement of the semantics of the language. The redefining of the language of ECA rules is mostly suited for coding it in Java and later embedding the code in JXTA. JXTA acts as the backbone for the development of the P2P layer.

Regarding the implementation of the ECA rules, the code that I have written now stores the rules and events and finds the corresponding rules matching the events that have occurred.

This is a very preliminary work. This work can be further extrapolated for complex events and can be improved by including the evaluation of the condition.

Acknowledgements

I would like to thank Prof Fausto Giunchiglia for introducing me to ECA rules and guiding me through out the project. I would also thank Ilya Zaihrayeu for helping me with the code.

References

[1] P A BERNSTEIN, F.GIUNCHIGLIA, A KEMENTSIETSIDIS, J. MYLOPOLOUS, L SERAFINI, I ZAIHRAYEU. Data Management for Peer to Peer computing: A vision WebDB02 –Fifth International Workshop on the Web and Databases, 2002.

[2] FAUSTO GIUNCHIGLIA. www.dit.unitn.it/~fausto – C2C Project

[3] I. ZAIHRAYEU, Query answering in Peer to Peer Database Networks, Technical Report

[4] VASILIKI KANTERE, A rule mechanism for Peer to Peer Data Management, Master Thesis report, 2002

[5] CHAKRAVARTHY, S., KRISHNAPRASAD, V., ANWAR, E., KIM, S.K. 1994 Composite events for Active Databases: Semantics, contexts and detection. In Proceeding of the Twentieth International Conference on Very Large Databases, J. Bocca, M. Jarke, and C .Zanialo, Eds., Morgan-Kaufmann, San Mateo, Ca, 606-617

[6] N. PATON, O. DIAZ, Active Database Systems, ACM Computing Surveys, Vol 31, No.1 , March 1999

[7] VASILIKI KANTERE, ILUJU KIRINGA,J. MYLOPOLOUS, Cordinating Peer Databases Using ECA Rules, Department of Computer Science, University of Toronto, School of Information Technology and Engineering, University of Ottawa

[8] FAUSTO GIUNCHIGLIA. “Contextual reasoning”. Epistemologia, special issue on “I Linguaggi e le macchine”.

Vol. XVI, pages 45-364, Tilgher-Genova, Italy, 1993.

[9] FAUSTO GIUNCHIGLIA, CHIARA GHIDINI “Local Models Semantics, or Contextual Reasoning = Locality + Compatibility”. KR’98 –Sixth International Conference on Principles of Knowledge Representation and Reasoning. Morgan-Kauffman, 1998. Long version: Ghidini, C., and Giunchiglia, F. “Local Models Semantics, or Contextual Reasoning = Locality + Compatibility”. Artificial Intelligence. 127(3):221-259, 2001.

[10] ANNA PERINI, ANGELO SUSI, FAUSTO GIUNCHIGLIA. “Designing Coordination among Human and Software Agents”. SEKE'02 –Fourteenth International Conference on Software Engineering and Knowledge Engineering, 2002.

Appendix

Event.java

```
package project;

class Event {

 String Db_Action;
 String Rel_Name;
 String Time;

 void Set(String Dbname, String Rname, String t)
    {
       Db_Action = Dbname;
       Rel_Name = Rname;
       Time = t;
    }//End of the function Set

  }//End of the Class Event
```

Execution.java

```
package project;

import java.util.Vector;
import java.io.*;

public class Execution {

  public static void main (String args [])
    throws IOException
    {
    BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

    String str [] = new String[100];
    String choice,name,relname,aname,arelname;
    String time,atime;

while(true){

    /*for(int i=0;i<=60;i++)
     System.out.println();*/

    System.out.println("Enter a choice \n");
    System.out.println("1 Enter an Event \n");
    System.out.println("2 Enter  a Rule \n");
    System.out.println("3 List the Events \n");
    System.out.println("4 List the Rules \n");
    System.out.println("5 Execute \n");
    System.out.println("6 Quit \n");
    System.out.println("ENTER UR CHOICE : \n");

    choice = br.readLine();

    if(choice.equals("1"))
      {
```

```
      System.out.println("\n Enter the name of the event :\n");
        name= br.readLine();
      System.out.println("\n Enter the name of the relation:\n");
        relname = br.readLine();
      System.out.println("\n Enter the time :\n");
        time = br.readLine();
      Event Newevent = new Event();
      Newevent.Set(name,relname,time);
      Queue.registerEvent(Newevent);
      }
    if(choice.equals("2"))
      {
      System.out.println("\n Enter the name of the event :\n");
        name= br.readLine();
      System.out.println("\n Enter the name of the relation in the 
event:\n");
        relname = br.readLine();
     /* System.out.println("\n Enter the time  of the event:\n");
        time = br.readLine();*/
      System.out.println("\n Enter the name of the action :\n");
        aname= br.readLine();
      System.out.println("\n Enter the name of the relation in the 
action:\n");
        arelname = br.readLine();
      System.out.println("\n Enter the time  of the action:\n");
        atime = br.readLine();

      Rule Newevent = new Rule();
      Newevent.Set(name,relname,aname,arelname,atime);
      Queue.registerRule(Newevent);
      }
    if(choice.equals("3"))
           Queue.GetAllEvent();

    if(choice.equals("4"))
              Queue.GetAllRule();

    if(choice.equals("5"))
```

```
     {
     System.out.println("Executing.............\n");
     Queue.Execevent();
     }
    if(choice.equals("6"))
    {
        System.out.println("OKFYNE");
         break;
     }
    }
    }
  }
```

Queue.java

```
package project;

import java.util.Vector;

public class Queue {

       public static Vector EventQ = new Vector(10,10);
       public static Vector RuleQ =  new Vector(10,10);
       static int countE=0, countR =0;
       public static int registerEvent(Event Newevent)
         {
          EventQ.addElement(Newevent);
          countE++;
          return EventQ.capacity();
          }
       public static int registerRule(Rule Newrule)
         {
          RuleQ.addElement(Newrule);
          countR++;
          return RuleQ.capacity();
          }
       public static void GetAllEvent()
        {
         Event getevent = new Event();
         for(int i=0; i<countE;i++)
          {
           getevent = (Event) EventQ.get(i);
           System.out.println("Event" + i +":");
           System.out.println("  " + getevent.Db_Action +"," +
getevent.Rel_Name + "," + getevent.Time);
           }
        }
       public static void GetAllRule()
        {
         Rule getrule = new Rule();
         for(int j=0; j<countR;j++)
          {
           getrule = (Rule) RuleQ.get(j);
           System.out.println("Rule" + j +":");
           System.out.println("  " + getrule.E_Db_Action +"," +
getrule.E_Rel_Name + "," + getrule.A_Db_Action + "," +
getrule.A_Rel_Name +"," + getrule.A_time);
           }
        }


       public static void Execevent()
        {
           Event Trigevent = new Event();
           Trigevent = (Event) EventQ.firstElement();
           for (int i =0; i <countR; i++)
            {
              Rule Checkrule = (Rule) RuleQ.get(i);
              if( (Trigevent.Db_Action.equals(Checkrule.E_Db_Action))
                  && (Trigevent.Rel_Name.equals(Checkrule.E_Rel_Name))
)
```

```
                 {
                 System.out.println("Rule found for the event \n");
                 System.out.println("Rule" + i +":");
                 System.out.println("  " + Checkrule.E_Db_Action +"," +
Checkrule.E_Rel_Name + "," + Checkrule.A_Db_Action + "," +
Checkrule.A_Rel_Name +"," + Checkrule.A_time);

                 if(  (ExecCondition(Checkrule)) )
                   ExecAction(Checkrule);
                 }//End of if loop

              }//End of for loop
              System.out.println("Removing the event from the EventQ
\n");
              EventQ.remove(0);   //remove the event as it is done with
              System.out.println("Reducing the size of EventQ by 1\n");
              EventQ.trimToSize();  //reduce the size of the vector by
1
              countE--;
         }//End of the method as Execevent

        public static boolean ExecCondition(Rule Execrule)
        {
        //do the necessary condition check
        System.out.println("Checking for condition.....\n");
        return true;

        }

        public static void ExecAction(Rule Execrule)
        {
        //do the necessary action
        System.out.println("Doing the necessary action \n");
        return;
       }

     }//End of class Queue
```

Rule.java

```
package project;

class Rule {

 String E_Db_Action;   /*Events are stored*/
 String E_Rel_Name;

 //The condition is also stored

 String A_Db_Action;  /*Actions Are Stored*/
 String A_Rel_Name;
 String A_time;

 void Set(String Edbname, String Ername,String Adbname, String Arname,
  String Atime)
  {
  E_Db_Action = Edbname;
  E_Rel_Name = Ername;

  A_Db_Action = Adbname;
  A_Rel_Name = Arname;
  A_time = Atime;

  }//End of the function set
}//End of Class Rule
```