Integrity Constraint Enforcement in a Multidatabase using Distributed Active Rules

L. G. Gomez and S. D. Urban
Arizona State University
Department of Computer Science and Engineering
Tempe AZ 85287-5406 U.S.A.
Tel.: (602) 965-2784 Fax: (602) 965-2751
E-mail: {Lgomez |S.urban}@asu.edu

Abstract

Active database rules have been identified as a useful technology for integrity constraint maintenance in centralized database systems. Maintaining constraints in a distributed environment such as that of a multidatabase system provides an even more challenging task for active rule technology. This paper presents the notion of distributed active rules for constraint processing in a distributed environment, together with an architecture for the use of such rules. The specification of distributed active rules is based on the statement of constraints that exist between heterogeneous database sources. The condition of a distributed rule must provide for an efficient means to check both local and remote constraint conditions. We present the structure of distributed active rules and provide an execution semantics for such rules. We also describe an architecture for communication between local and global rule processors. Finally, we discuss future research issues associated with the analysis of distributed constraints and the generation of distributed active rules.

Keywords

Integrity Maintenance, Multidatabases, Distributed Active Rules

1 INTRODUCTION

An important challenge in modern information systems is the integration of heterogeneous and autonomous data systems. The integration of such components is commonly known as a *multidatabase system* (Elmargamid *et al.* 1990). As part of that challenge, a significant problem to be addressed is consistency maintenance among distributed database components. In centralized and distributed systems, integrity constraints are often implemented directly in the application code. As a result, errors and omissions in the checking and main-

S. Jajodia et al. (eds.), *Integrity and Internal Control in Information Systems*

tenance of constraints can easily be introduced. Furthermore, if guaranteeing database consistency in a single database is a difficult problem, data communication issues, heterogeneity and autonomy make constraint maintenance in a multidatabase environment an even more complex task.

Decoupling applications from consistency maintenance is referred to as *knowledge independence* in (Baralis *et al.* 1994). To support knowledge independence, *active database* technology (Widom *et al.* 1996) can be used together with a constraint specification language to provide automated responses to constraint violating operations. Active database technology has primarily been investigated in the context of centralized systems, making use of *Event-Condition-Action* rules to provide reactive behavior. For example, when an event occurs that may potentially violate a constraint, a condition is evaluated to test for constraint satisfaction. If the constraint is violated, an action is triggered to repair the constraint violation.

This paper presents the concept of distributed active rules for constraint maintenance in a multidatabase environment, together with an architecture for the execution of such rules. The specification of distributed active rules is based on the statement of constraints that exist between heterogeneous database sources, where constraints can either be *private global constraints* or *public global constraints*. In the case of private global constraints, a local database expresses a constraint that must be maintained locally based on conditions that exist in remote databases. Public global constraints are constraints that must be maintained globally through cooperation among all databases involved in the constraint. Constraints in this environment are expressed using the Multidatabase Constraint Specification Language (MCSL), (Gomez *et al.* 1997), which is based on the ODMG 2.0 standard (Cattel 1994) for the expression of constraint query conditions.

Distributed active rules are developed based on the statement of MCSL constraints. The specification of distributed active rules must be concerned with checking both local and remote conditions for the purpose of detecting constraint violations. Furthermore, distributed rules stored at a local site can be triggered by events that occur at remote sites. In this paper, we describe the structure of distributed rules and the manner in which rule conditions are organized into local and remote conditions. We also describe the architecture of the environment, including rule processing components that must exist at each local database and the global rule processing components that must be constructed to support distributed execution of remote rule conditions.

The contribution of the work presented in this paper lies in the extensions that we have defined for transforming event-condition-action rules into rules that function over distributed data. The use of distributed active rules supports the definition of non-trivial constraints between heterogeneous database sources and provides a viable mechanism for communication between databases in the checking and maintenance of such constraints. Through the use of such rules, active database technology can therefore be extended into distributed

domains, where the databases involved in the multidatabase environment are autonomous and otherwise passive database systems.

The remainder of this paper is organized as follows. Related work on constraint specification languages and constraint maintenance in distributed environments is first presented in Section 2. In Section 3, we describe the multidatabase environment and the type of constraints supported. Section 4 describes the distributed active rule language. Rule execution semantics and the architecture of the distributed active multidatabase system are presented in Section 5. The paper concludes in Section 6 with future research directions.

2 RELATED WORK

Research on the use of active databases for constraint maintenance has received substantial attention in recent years (Widom *et al.* 1996). Most of this work has primarily been investigated in the context of centralized systems and does not consider rules and constraints in distributed or heterogeneous database systems. In this section, we address research related to integrity constraints and distributed active databases.

Constraint management in distributed environments initially focused only on tightly coupled systems (Simon *et al.* 1986). Later, new approaches were proposed for loosely coupled environments in which distributed transactions are not available. One approach to constraint maintenance in a multidatabase is based on the concept of data dependencies. A Data Dependency Descriptor model which includes consistency predicates and restoration procedures is proposed in (Rusinkiewicz *et al.* 1991). Another approach described in (Ceri *et al.* 1993) uses active rules and persistent queues to maintain the consistency of existence and value dependencies between relational databases.

Protocols are used in (Grefen 1994) for integrity constraint checking in federated databases. The basic protocol detects an update, raises an alarm when a violation is detected, and notifies the Constraint Manager. Protocols vary in terms of requirements of the underlying systems, level of asynchronous communication, flexibility and execution cost. Not all the protocols proposed are accurate, meaning that they can produce 'false alarms' (notification when a violation did not occur). Repairing actions in this approach are not addressed.

(Chawathe *et al.* 1996) suggest a formal approach for constraint management in loosely coupled distributed databases where locking and transaction primitives may not be available. Weaker notions of constraint maintenance are formalized and an event-based formal framework is introduced.

The constraint approach in (Grufman *et al.* 1997) describes the integration of a functional database and an active object system to enforce integrity across a multidatabase. The systems are integrated using a tightly-coupled approach where the global schema is maintained by the functional database. Constraints are limited to universally quantified variables over a simple conjunction of predicates.

Optimization techniques for distributed constraint checking to avoid remote database access have also been investigated. In (Barbara *et al.* 1992), the Demarcation protocol is used to maintain simple arithmetic constraints. The work in (Gupta 1993) suggests a method to generate local tests to test global integrity constraints.

Issues related to the use of active rules in a distributed environment have only recently been investigated. Rule decomposition, rule distribution, and correct evaluation of distributed rules in a distributed active database are analyzed in (Hsu *et al.* 1992). This research is performed in the context of relational databases where relations are partitioned horizontally and/or vertically and segments are distributed among sites. Rule queries are decomposed using algebraic manipulations based on principles of query optimization. Condition evaluation can be done in a distributed fashion, but rule processing in general is still centralized.

In (Ceri *et al.* 1992), a locking scheme and a rule-task executor that allow rules to reference data at multiple sites is described. To coordinate sites and support rule priorities, additional locking and communication protocols are proposed. One limitation of the approach is that tables cannot be replicated or fragmented across sites.

In (Pissinou *et al.* 1996), a reactive multidatabase architecture that permits the explicit specification, recognition and resolution of temporal changes to support interoperability of objects over time is proposed. Our architecture differs from (Pissinou *et al.* 1996) in that rule events, conditions and actions are decoupled and can be executed at different sites.

The difference between our work and the research presented above is that we address remote rule condition testing and remote action execution within an object-oriented approach to the expression of constraints and rules. Active Rules are structured to use optimization techniques that minimize communication with remote sites. Furthermore, we focus on non-trivial inter-object constraints that involve components stored in different databases.

3 THE MULTIDATABASE ENVIRONMENT

This section presents the general framework of the multidatabase environment that we are assuming for this research. Section 3.1 describes the architecture of the environment together with an Airline application that will be used as a running example in the rest of the paper. Section 3.2 then describes the types of constraints that are maintained through the use of distributed active rules.

3.1 The Multidatabase environment

We are assuming a loosely-coupled, federated database system in which there is no global schema. Each database in the federation may have a different

data model and different database capabilities. It is the responsibility of the multidatabase administrator to resolve any differences between database components. Although each database component is autonomous, some of the databases may require access to data from remote components of the federation to enforce application rules.

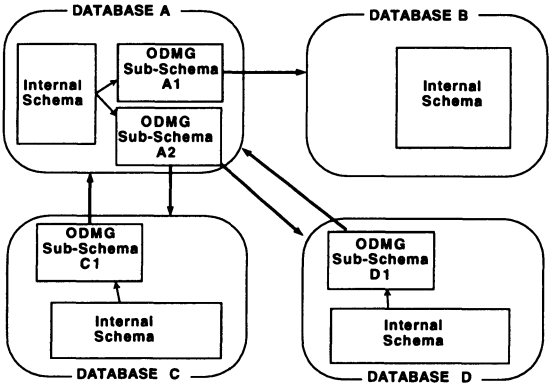

Figure 1 Multidatabase Architecture

To integrate information from different databases, each database provides an object-oriented sub-schema based on the ODMG model (Cattel 1994). Each database in the multidatabase environment can access information from the sub-schemas provided by other databases in the federation. To access data in a remote database, a local database will *import* an ODMG definition of the schema provided by the remote database. The imported schema, on the other hand is viewed as the *export* schema of the remote database. Also, a single database can export different sub-schemas to different databases. For example, Figure 1 illustrates a multidatabase composed of four databases. Database A *exports* sub-schema A1 to remote database B and sub-schema A2 to remote databases C and D. Database A also *imports* sub-schemas C1 from database C and D1 from database D. Not all databases need to export data. Sometimes a database only needs access to remote data such as in the case of database B.

To support our discussion of distributed constraints and rules, we introduce a small airline example. Assume that AirFun, LittleAir, and ControlAir are three independent enterprises that maintain their own database applications. These three companies need to coordinate and share information. *ControlAir* is a regulation agency that maintains global information about crew members, cities and airlines. ControlAir also maintains statistics about accidents, the flight history of pilots and other information that may be used for any airline. *AirFun* is an airline that provides passenger transportation and cargo services. This airline keeps information about flights, planes, and packages shipped. *LittleAir* is a small airline that only offers passenger transportation.

LittleAir does not have crew staff and has to contract the services of AirFun airline. The database of LittleAir only keeps information about flights. Flights from AirFun or LittleAir can be assigned to any crew member who is registered by the regulation agency.

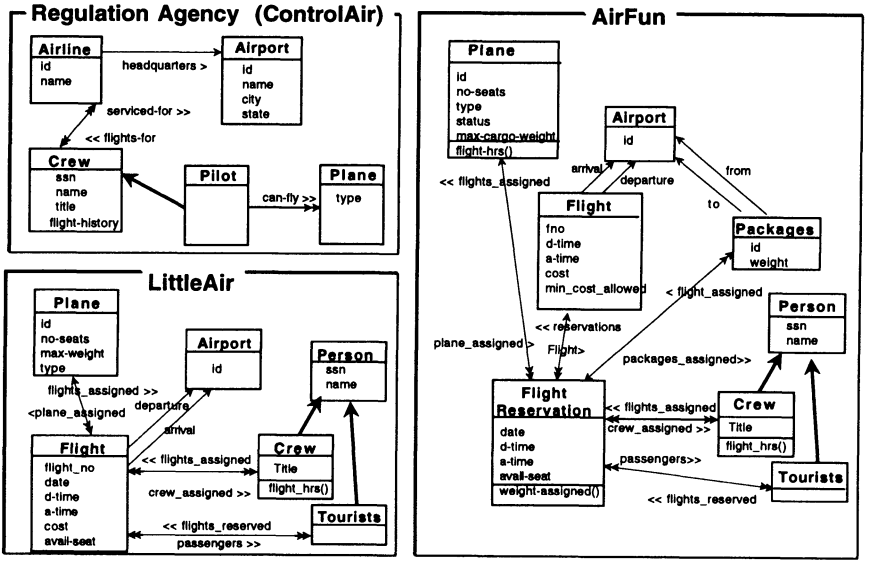

Figure 2 Exported schemas for Airline Example

The export schemas of AirFun, LittleAir and ControlAir are shown in Figure 2. In the graphical notation, each abstract object is represented inside a box. The upper part contains the *name* of the object, the middle part contains all *simple properties*, and the lower part contains all *derived attributes* and *methods*. Relationships are represented by labeled arrows between the abstract objects. Single-arrows and double-arrows represent single-valued and multi-valued properties, respectively. Bold unlabeled arrows represent ISA relationships.

3.2 Multidatabase Constraints

In the multidatabase environment introduced in the previous section, individual databases collaborate in the exchange of information. External control from other databases, however, is limited. In this environment, we identify two different forms of distributed constraints: Private Global constraints and Public Global constraints.

Private Global Constraints (PvGC's) define dependencies between data that is stored in more than one database. PvGC's are considered private because the remote database is not aware of the constraint. The remote database allows limited access to some of its information through an export schema. To define a PvGC's, it is possible to use any component of the internal database schema of the local database as well as any component of the remote schemas that are being imported to the local database. Since remote databases are not aware of the constraint, the local database cannot generate actions that may alter the remote databases when a PvGC violation is detected. However, the local database can be informed when a change in the remote database has occurred. The local database can then take some local action to fix the violation. Consider the following example of a PvGC:

Pilots in LittleAir can only fly on the planes for which they successfully completed the required training by ControlAir.

If a license to fly plane 'B777' is revoked for a pilot in the ControlAir database, a notification is generated to remote database LittleAir. If the constraint is violated, LittleAir does not have the authorization to abort the transaction in ControlAir. It can, however, generate a local action to eliminate the pilot from all the flights that have plane 'B777' assigned.

Public Global Constraints (PbGC's) are constraints that are associated with entities in more than one database. All participant databases in the federation agree with the definition of this kind of constraint. The specification of PbGC's is done in the context of the exported schemas. The internal schemas of each database are not available for the federation. Each of the individual databases involved in the specific constraint collaborates during the constraint maintenance process. In the local database in which the event is triggered, if the global constraint is violated a local action is executed to restore the violation. An example of a public global constraint is:

For safety reasons imposed by the regulation agency, pilots and flight attendants cannot fly more than 8 hours in the same day.

Private and Public global constraints are expressed in a high level declarative language called the Multidatabase Constraint Specification Language (MCSL). MCSL provides a syntax based on ODMG OQL that allows easier expression of complex constraints between databases. Consider the constraint example illustrated in Figure 3. This safety constraint imposed by the regulation agency indicates that crew members cannot fly more than 8 hours in the same day .

The *ForAll* section supports the declaration of object variables used in the expression of the constraint, indicating that the condition to be expressed as part of the constraint must be true for all Crew members c_i and all flights f_i assigned to the crew. The scope of variables in the forall section can be

```
PbG CONSTRAINT AirFun_valid_flight_hours
ForAll:
    c in AirFun::Crew, f in c.flights_assigned
Define:
    AirFun_hours := c.flight_hours(f.date),
    LittleAir_hours := element SELECT c2.flight_hours(f.date)
                       FROM Crew c2
                       WHERE c.ssn= c2.ssn
                       USING LittleAir
Condition:
    sum(AirFun_hours, LittleAir_hours) <= 8
end
```

Figure 3 Safety constraint example

any object defined in the import or local schema. The notation Airfun::Crew denotes that Crew is a class in the AirFun database.

The *Define* section allows the declaration of variables that represent values that result from the evaluation of local or remote queries. Such variables can be used in the specification of the constraint *Condition*. In the above example, AirFun_hours is the number of hours that c_i has in the local database on the date of flight f_i. Similarly, LittleAir_hours is the number of hours that c_i has in the remote database on the same date. The condition expresses that the sum of the hours should not exceed the value of 8.

To enforce MCSL constraints, we use distributed active rules. The following sections describe how distributed rules are used to check and maintain constraints.

4 DISTRIBUTED ACTIVE RULE DEFINITION

Given the basic assumptions about the multidatabase system and the constraint language, this section presents the details of the distributed active rule language. Section 4.1 describes the basic structure and general semantics of distributed rules. The use of distributed active rules is illustrated with examples in Section 4.2.

4.1 Rule structure

Existing rule languages are defined in the context of centralized database systems. An important aspect to consider in a distributed environment is the identification of the local and the remote components needed to evaluate the rule in an efficient manner. For example, in some cases it may be possible to validate global constraints by checking local data only (Gupta 1993). In other cases, it may be required to examine data at remote sites. Under some

circumstances, both local and remote conditions must be examined. Given these considerations, the basic structure of a distributed active rule is shown in Figure 4.

Rule <*rulename*>
  Event: E_1 or E_2 or ... or E_n
  [**Condition**:
    [exists {O_i in <*class_variable*> | <*class_variable*>.<*attribute*> } where]
      <*local_condition_name*>($p_1, ..., p_n$)=true OR
      (<*local_condition_name*>($p_1, ..., p_n$)=unknown AND
      <*remote_condition_name*>($p_1, ..., p_n$)=true)
  ]
  Action: <*action-list*>
  Priority:
    [before <*rule-list*>]
    [after <*rule-list*>]
End_Rule

Figure 4 Structure of the distributed active rule

In the **event** specification, each E_i has the form before | after <*event-name*>. The before and after options are used to indicate when the condition and action of the rule are executed with respect to the event. Specifically, the before directive indicates that the rule condition and action are evaluated before the execution of the event. Similarly, the after directive indicates that the condition and action are evaluated after the execution of the event. A rule can be triggered by events that occur at local or remote sites. Events at remote sites, however, can only be used together with the after directive.

In general, any method can be used in the event specification. However, the events of interest for constraint maintenance are only the low-level operations that change the state of the database. In the object-oriented model used as a framework, the low-level operations that can alter the state of the database are: 1) New_<*object_name*> and Delete_<*object_name*>, used to create and delete instances, 2) Modify_<*attr_name*> and Modify_<*sv_rel_name*>, used to change values of attributes or single-value relationships, and 3) Insert_<*mv_rel_name*> and Delete_<*mv_rel_name*>, used to create or delete multi-valued relationships.

As in traditional centralized environments, the **condition** is a query that determines if the constraint is violated. A condition evaluation that returns a value of *true*, indicates that there is a violation of the constraint. An action can then be executed to restore the consistency of the data. Since it may be necessary to evaluate the rule condition in the local database and in remote locations, the condition is composed of a local part and a remote part. The local condition is evaluated first and if it returns a value of true, there is no need to check the remote component. In this case, the action can be triggered.

If the local condition returns a value of *false*, the condition is satisfied and the action is not triggered. For distributed constraints, however, there may not be enough information in the local database to validate a constraint. If the local condition evaluation returns a value of *unknown*, then the remote condition must be tested. Testing the condition of a distributed active rule is complicated by the fact that the condition must be evaluated for all objects affected by the event. As a result, for one instance of a triggered rule, some objects affected by the constraint may require the checking of local conditions only while others may require the checking of local and remote conditions. A more detailed description of the rule execution semantics occurs in Section 5.

Another option in the rule specification is the omission of the condition clause. These types of rules are known as event-action rules, in which the condition is assumed to be true and the action is always executed when the event is triggered.

For constraint maintenance, there are two types of **actions** that can be executed: *abortive* and *corrective* actions. An abortive action can be executed in the local database when a local event introduces a violation of a public global constraint. Corrective actions are executed when a remote event causes a violation of a private global constraint in the local database that owns the private constraint. In this case, the local database does not have the authority to alter the state of the remote transaction that generated the event. Corrective actions can be executed, however, in the local database to restore the consistency of the data.

Finally, a rule **priority** can be defined using the before and after clauses in the rule definition. The rules specified in the before clause of rule R_i are executed before R_i and the rules specified in the after clause are executed after R_i. Note that we also assume immediate coupling modes between the event and the condition and between the condition and the action. We have not yet addressed the issues of deferred coupling modes for distributed rules.

4.2 Rule Examples

To illustrate the distributed active rule language introduced in the previous section, this section presents several rule examples that refer to the airline application of Section 3.1. Consider again the private global constraint from Section 3 that restricts the planes that can be assigned to a pilot. The specification of this constraint is shown in Figure 5.

During the rule specification process we need to identify the operations that can affect the constraint and the entities involved. For example, this constraint involves the entities LittleAir::Crew, LittleAir::Flight, LittleAir::Plane, ControlAir::Crew and ControlAir::Plane. Identifying the possible updates that can affect this constraint is not trivial because the information about the type of plane a pilot can fly is stored in the database ControlAir and the assignment

```
PvG CONSTRAINT LittleAir_Crew_Can_Fly
Description: Pilots can fly only in planes for which they completed required training
ForAll :
    c in LittleAir::Crew, f in c.flights_assigned
Define:
    plane_type_assigned := f.plane_assigned.type
Condition:
    IF c.title ='PILOT' and plane_type_assigned<> NULL THEN
    Exists SELECT p2.type
        FROM Crew c2, c2.canfly p2
        WHERE c.ssn = c2.ssn and p2.type= plane_type_assigned
        USING ControlAir
end
```

Figure 5 Can-fly constraint example

of flights is stored in LittleAir database. Furthermore, there is no direct relationship between a crew member and the type of plane he/she is assigned to fly in any of the airline databases. The plane type must be examined by traversing through the flights assigned to each crew member and the plane assigned to each flight.

Assigning a flight to a crew member, changing the plane assigned to a flight, modifying the type of a plane or removing a canfly relationship in ControlAir are a few examples of operations that can affect this constraint. For each operation, an active rule can be defined. The following examples illustrate two of those rules.

Example 1 The active rule in this example is used to verify the consistency of the database when a plane is assigned to a flight:

```
Rule Crew_Can_Fly
    Event: before LittleAir::Flight.modify_plane_assigned(Flight,Plane) |
           before LittleAir::Plane.insert_flights_assigned(Plane ,Flight)
    [Condition:
        exists C in Flight.crew_assigned where
            Local_Crew_can_Fly_is_invalid(C, Flight)=true OR
            (Local_Crew_can_Fly_is_invalid(C, Flight)=unknown AND
            Remote_Crew_can_Fly_is_invalid(C, Flight)=true)
    Action: Abort
End_Rule
```

The local and remote condition are evaluated for all crew members $c_i \in$ Flight.crew_assigned. If the condition is true for any c_i, the transaction is aborted. The details of the local and remote conditions are:

```
Rule_condition Local_Crew_canFly_is_invalid (c,f)
    Subcond 1:
        if c.title='PILOT' and f.plane_assigned <> NULL then
            Execute Next condition (Subcond 2)
        else return(false)
    end
    Subcond 2:
        plane_type_assigned := f.plane_assigned.type
        OQ_Crew_canFly(c, plane_type_assigned,found)
        if found =true then return(false)
        else return(unknown)
    end
EndRule


Rule_condition Remote_Crew_canFly_is_invalid (c,f)
    Subcond 1:
        RQ_Crew_canFly(c.ssn,plane_type_assigned,found)
        if found =true then return(false)
        else return(true)
    end
EndRule
```

Using the constraint specification, local and remote tests are developed. The local condition is expressed in terms of local objects only. The remote condition is defined in terms of the results obtained in the local condition combined with calls to remote databases. For each combination (c_i,*Flight*), the local condition is evaluated first. In the local rule condition, notice that there are two subconditions. The first subcondition is evaluating the local predicates identified in the constraint specification. The second subcondition contains OQ-calls that optimize the remote condition checking process with the introduction of additional tests that can avoid the need to evaluate the remote condition. The 'OQ' or *Optimizer Query* identifies the method call as a locally optimized test. In the example above, the OQ_Crew_canFly is checking if c_i has another flight in the local database with the same airplane type. If another flight is found, we can conclude that the plane type is valid and there is no need to test the remote condition. However, if no flight is found, the status of constraint satisfiability is unknown and we proceed to test the remote condition. The remote condition calls the remote query RQ_Crew_canFly to find in ControlAir if the combination c_i and *plane_type_assigned* is valid.

Example 2 In this example, we illustrate that two different options for event-action rules can be used to restore the consistency of the data when a license to fly a plane is revoked for a pilot in ControlAir database. Notice that in this example, LittleAir receives a remote event originated in ControlAir. Let

('John','B777') be the link removed in ControlAir. To eliminate the violation there are two options:

1. To remove all the flights with plane 'B777' that 'John' has in LittleAir.

   Rule Crew_Cannot_Fly_1
   Event: after ControlAir::Crew.delete_can-fly(C,Plane-type)
   Action: LittleAir::Crew.Delete_flights_assigned(C.ssn, Plane-type)
   End_Rule

   The action Delete_flights_assigned is implemented as the following update:

   Delete relationship flights_assigned (crew_assigned)
   from crew c2, c2.flights_assigned f
   where c2.ssn='John_ssn' and f.plane_assigned.type='B777'
   using LittleAir

2. To remove the plane 'B777' from all the flights that 'John' has in LittleAir.

   Rule Crew_Cannot_Fly_2
   Event: after ControlAir:Crew.delete_can-fly(C,Plane-type)
   Action: LittleAir::Flight.Delete_plane_assigned(C.ssn, Plane-type)
   End_Rule

   The action Delete_plane_assigned corresponds to the following update:
   Delete relationship plane_assigned (flights_assigned)
   from crew c2, c2.flights_assigned f
   where c2.ssn='John_ssn' and f.plane_assigned.type='B777'
   using LittleAir

5 RULE EXECUTION SEMANTICS

Given the presentation of distributed active rules in the previous section, this section describes the execution semantics for such rules. The distributed active database architecture that supports the execution of the distributed active rules is presented in Section 5.1. Section 5.2 describes the algorithm for processing distributed active rules.

5.1 Distributed Active Database Architecture

The architecture of the active component of the multidatabase environment is illustrated in Figure 6. We assume that each component can respond to read-only requests from remote components. In addition, local components can receive an event notification from a remote component and start the

execution of local transactions. Since the system does not support distributed transactions, local components are not allowed to send update requests to remote components. The database components can be passive databases. A layered approach is used to support the active functionality required for the detection of the events and the processing of the rules (Widom *et al.* 1996).

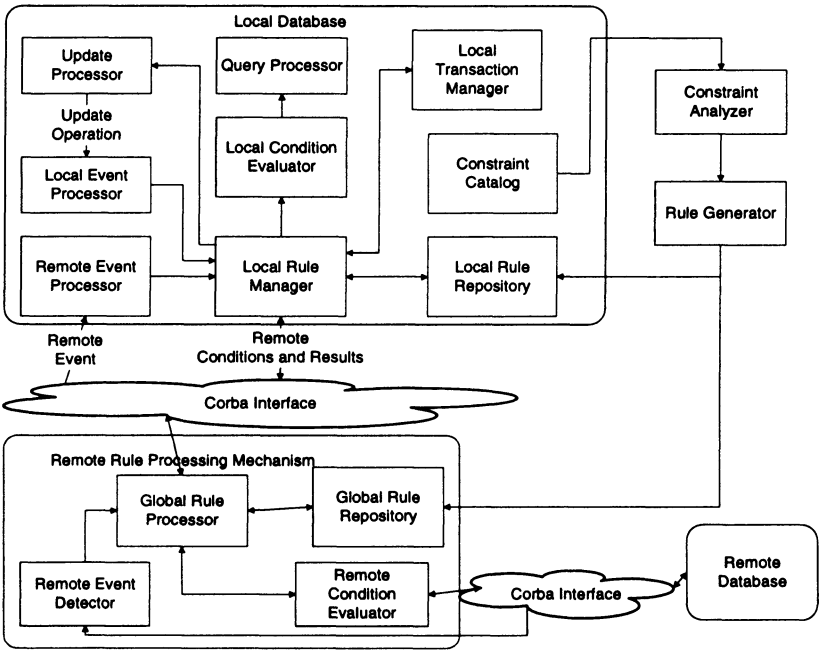

Figure 6 Architecture of the active multidatabase system

The rules used in the distributed active rule system are stored in the data repositories. The *Constraint Catalog* contains the specification of the constraint in MCSL. This specification is used to report violations to the user. The *Local Rule Repository* contains global and local rules used to maintain consistency of the local database. The *Global Rule Repository* contains information about the remote queries and the corresponding global rules.

As shown in Figure 6, every component database is assumed to have an *Update Processor* that executes update requests in the local database. If an update is specified as an event of an active rule, the rule is executed. If the local database is a traditional passive database, then the update processor must be extended to signal the local event processor when a change has occurred in the database. The *Query Processor* executes read-only requests in the local database. Since these requests do not change the state of the database, the query processor does not trigger any integrity constraint rule. However, the

query manager participates in the rule execution process by evaluating local conditions.

The *Local Rule Manager* controls the execution of local and distributed active rules. When an event is detected, the rule manager triggers the rules associated with the event and calls the *Local Condition Evaluator* to test conditions in the local database. If there is not enough information to verify the constraint locally, the rule manager invokes the remote rule processing mechanism.

The remote rule processing mechanism provides the communication interface between databases and controls the execution of global rules. The *Global Rule Processor* executes the remote calls needed to check constraint conditions from a remote database. The *Remote Condition Evaluator* is invoked by the global rule processor to evaluate read-only queries in a remote database. In a distributed environment, the same rule may be triggered by events in different databases. Therefore, it is necessary to provide an execution model that supports both concurrent and sequential rule execution. The Global rule processor allows concurrent execution of rules but also serializes rule execution when conflicts in the concurrent access of data are detected. Concurrency control and recovery for transactions operating in the local database is provided by the *Local Transaction Manager.*

The *Remote Event Processor* is invoked when an event is executed at a remote database. When the remote event occurs, the database in which it occurs must notify the *Remote Event Detector* of the remote rule processing mechanism. The remote event detector then signals the remote event processor at the local databases that are interested in the occurrence of the event. When the signal is received, an active rule is triggered. Typically, the actions executed within the rule are local corrective operations to satisfy a constraint that was violated by the remote database.

Interoperability of all database components in the environment is achieved through the use of CORBA technology (CORBA 1993). In particular, we have used ORBeline (ORBeline 1994) to develop the prototype for this research (Healy 1997). The use of the CORBA distributed object framework provides several advantages. First, CORBA provides the mechanisms by which objects transparently make requests and receive responses by simply invoking methods calls. Second, the low-level communication is hidden by CORBA and the modules are written independently of the communication. Finally, CORBA objects are not attached to a specific location. Therefore, it is easy to redistribute modules as the system matures.

5.2 Rule Processing

In this section we present the algorithm for processing of distributed active rules. To formally describe the rule processing algorithm, we first introduce several concepts and definitions.

Definition 1 Let K_i be a constraint. The *Constraint Class Set* of K_i denoted as $CCS(K_i) = \{C_1, C_2, ..., C_n\}$ is the set of all class names on which the constraint K_i has an effect.

Definition 2 Let E_j be an event in the database. The *Potentially Violated Set* of constraint K_i by the event E_j, denoted $PVS(K_i, E_j)$, is the set of object instances $\{P_1, P_2, ..., P_n\}$ that can violate the constraint K_i when E_j is executed. Each P_i is a tuple of the form $[O_1, O_2, ..., O_n]$, where each O_i is a member of a class in $CCS(K_i)$.

Definition 3 The *before-event* rule set $BR(E_j) = \{br_1, br_2, ..., br_n\}$ is the set of rules with the before directive in the specification of event E_j. Likewise, the *after-event* rule set $AR(E_j) = \{ar_1, ar_2, ..., ar_n\}$ is the set of rules with the after directive in the specification of event E_j.

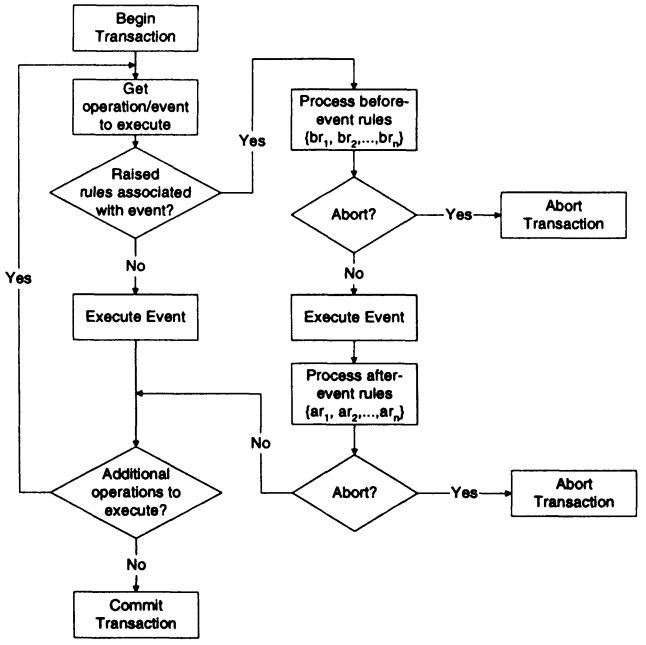

Figure 7 Transaction processing

The control flow for transaction processing at each local site is shown in Figure 7. When the event E_i is detected, the rule process starts for the before-event rules $\{br_1, br_2, ..., br_n\}$. If an abort directive occurs with such a rule, the transaction is aborted immediately. If a rule does not invoke an abort,

the event is executed. Finally, the set of after-event rules $\{ar_1, ar_2, ..., ar_n\}$ is processed in the same manner.

Each rule set is processed sequentially before and after the event using algorithm 1. In each rule set, rules are ordered based on priority. During the rule evaluation process, there are two queues that contain the object instances that are being tested for constraint violation. The *local condition queue* contains the object instances P_i that need to be evaluated in the local database. Similarly, the *remote condition queue* contains the object instances P_i that need to be evaluated in the remote database.

Algorithm 1. Let E_j be the update operation that triggered the rule, the processing of the before-event and after-event rule set is as follows:

While rule set not empty {
1. Select a Rule R_i from rule set
2. Identify potentially violated set $PVS(K_i, E_j)$ and append each $P_i \in PVS(K_i, U_j)$ to the local condition queue.
3. **While** local condition queue not empty
   3.1 Get tuple $[O_1, O_2, ..., O_n]$ from local-condition-queue
   3.2 result= evaluate-local-condition(*condition-name*, $O_1, O_2, ..., O_n$)
   3.3 if result =*unknown* then
   enqueue(remote-condition-queue, $[O_1, O_2, ..., O_n]$)
   3.4 if result= *true* then
   execute action
4. **While** remote condition queue not empty
   4.1 Get tuple $[O_1, O_2, ..., O_n]$ from remote-condition-queue
   4.2 result = evaluate-remote-condition(*condition-name*, $O_1, O_2, ..., O_n$)
   4.3 if result = *true* then
   execute action

}

The condition evaluation of each P_i involves local and remote condition checking. A result of *unknown* in the evaluation of the local condition for tuple P_i indicates that there is not enough information in the local database to test the constraint. Therefore, the condition checking is not complete and the P_i is appended to the remote condition queue for further processing. If the result of the condition is *false*, there is no constraint violation and there is no need to check the condition for P_i in the remote database. However, if the result of the condition is *true*, the constraint is violated and the rule action is executed immediately. If the rule action is an abort, the rule processing algorithm terminates. If the rule action is a corrective action then the corrective action is invoked as a subtransaction to the triggering event for each object that violates the constraint. After the local condition is tested for all P_i's in the

potentially violated set, the remote condition is evaluated for the P_i's in the remote condition queue.

Example 3 Assume that John, Mike and Peter are pilots. Eagle and Early-Bird are planes of type 'A320'. Sky and Sunrise are planes of type 'B777'. The information of the type of planes that the pilots can fly as well as the flight assignments is shown in tables 1 and 2 :

Crew	canfly
John	'B777', 'A320', 'A340'
Mike	'B777', 'A320'
Peter	'A320', 'A340'

Table 1 Crew Information

Flight	crew-assigned	plane-assigned
f100	John, Mike, Ann	
f101	John, Mike	Eagle
f102	Peter, Ann	Eagle
f103	John, Ann	Sky

Table 2 Flight Information

For simplicity we use the crew name instead of ssn to identify the crew members. Also we show directly the values of the attributes instead of object identifiers. Notice how the algorithm is executed when the following events trigger the distributed active rules:

1. Assign plane EarlyBird to flight f100.
   The rule crew_can_fly is triggered and the potentially violated set is identified. When the local condition is checked, all elements satisfied the local condition. John and Mike have another flight with type 'A320'. Since Early-bird is also of type 'A320' it is assumed that the constraint is not violated. Ann is not a pilot and therefore satisfies the constraint. As a result, there is no need to check the remote constraint.
   PVS= { [John, f100], [Mike, f100], [Ann, f100] }
   Local Condition Queue= { [John, f100], [Mike, f100], [Ann, f100] }
   Remote Condition Queue{ null }

2. Assign plane Sunrise to flight f100.
   The rule crew_can_fly is triggered and the potentially violated set is identified. When the local condition is checked, John and Ann satisfied the local condition because John has another flight with type 'B777' and Ann is not a pilot. The remote constraint is only verified for Mike and no violation of the constraint occurs.
   PVS= { [John, f100], [Mike, f100], [Ann, f100] }.
   Local Condition Queue= { [John, f100], [Mike, f100], [Ann, f100] }
   Remote Condition Queue{ [Mike, f100] }

3. Revoke license for plane type 'B777' to John.
   Assume that we are using the Crew_Cannot_Fly_1 rule presented in Example 4.2 The action will eliminate relationship crew_assigned {John} for flights {f100,f103}.

6 SUMMARY AND FUTURE WORK

This paper has presented the concept of distributed active rules for the checking and maintenance of constraints in a multidatabase environment. Distributed rules extend the traditional notion of event-condition-action rules with the specification of rule conditions that involve local and remote components. We presented an execution model for distributed active rules as well as a distributed rule processing architecture for the execution of such rules. Distributed active rules support the use of complex constraints between heterogeneous database systems by decoupling multidatabase constraint enforcement from application code and providing a more general mechanism for the enforcement of complex constraints between distributed data sources.

There are several directions for future research that we are currently investigating. As illustrated in this paper, the rule definition process is not a trivial task. We are investigating techniques for the automated analysis of MCSL constraints to assist in the generation of the distributed rules. An important aspect of this work is to develop techniques for optimizing rule conditions so that the need for remote condition testing is minimized by making use of local data whenever possible. Another important aspect of this work involves a thorough analysis of distributed constraints to identify the operations that can violate a constraint and the databases in which those operations can occur. Finally, although we have experimented with the implementation of the concepts presented in this paper, a full implementation of a distributed active rule processing environment is still under development so that we can better analyze architectural and execution issues of distributed active rules.

Acknowledgments
We would like to thank Sean Healy of the Multidatabase System group for his valuable comments and development work during the implementation of the prototype.

REFERENCES

Barbara, D. and Garcia-Molina, H.(1992) The Demarcation Protocol: A technique for maintaining linear arithmetic constraints in distributed database systems. In *Proceedings of the International Conference on Extending Database Technology.*

Baralis, E., Ceri, S., Paraboschi, S.(1994) Declarative Specification of Constraint Maintenance. In *Proceedings of the 13th International Conference on the Entity-Relationship Approach.*

Cattel, R.(1997) *The Object Database Standard 2.0*, (Ed. R. Cattell and K. Barry). Morgan Kaufmann.

Ceri, S. and Widom, J.(1992) Production rules in parallel and distributed database environments. In *Proceedings of the 18th International Conference on Very Large Databases.*

Ceri, S. and Widom, J.(1993) Managing Semantic Heterogeneity with Production Rules and Persistent Queues. In *Proceedings of the 19th International Conference on Very Large Databases* .

Chawathe, S., Garcia-Molina, H. and Widom J.(1996) A Toolkit for Constraint Management in Heterogeneous Information Systems. In *Proceedings of 12th International Conference on Data Engineering.*

Elmargamid, A. and Pu, C. (Eds.).(1990) Special issue on heterogeneous databases. *ACM Computing Surveys*, Vol. 22, No. 3.

Gomez, L. and Urban, S. (1997) MCSL: A Multidatabase Constraints Specification Language, Computer Science and Engineering Technical Report. Arizona State University.

Grefen, P.(1994) Integrity Constraint Checking in Federated Databases. *Memoranda Informatica*, Dept. of Computer Science. Univ. Of Twente, The Netherlands.

Grufman, S., Samson, F., Embury, S. Gray, P., and Rish, T.(1997) Distributing Semantic Constraints Between Heterogeneous Databases. *Proceedings of 13th International Conference on Data Engineering.*

Gupta, A.(1993) Local Verification of Global Integrity Constraints in Distributed Databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Mgmt of Data.*

Hanson, E. and Khosla, S.(1997) An Introduction to the TriggerMan Asynchronous Trigger Processor. In *3rd International Workshop on Rules in Databases.*

Healy, S.(1997) A CORBA Implementation of a Multidatabase Framework to Support Distributed Rule Processing. MCS Project Report.

Hsu, I., Singhal, M., and Liu, M.(1992) Distributed Rule Processing in Active Databases. In *Proceedings of 8th International Conference on Data Engineering.*

OMG (1993), The Common Object Request Broker: Architecture and Specification, *OMG Document: 93-12-43 Revision 1.2.*

ORBeline 2.0 User's Guide (1994), Post|Modern Computing Technologies.

Pissinou, N. and Vanapipat, K.(1996) Active Database Rules in Distributed Database Systems: A Dynamic Approach to Solving Structural and Semantic Conflicts in Distributed Database Systems. In *International Journal of Computer Systems, Science and Engineering.* Vol. 11, No. 1.

Rusinkiewicz, M., Sheth, A., and Karabatis, G.(1991) Specifying Interdatabase Dependencies in a multidatabase environment. In *COMPUTER* Vol. 24 No.12.

Simon, E. and Valduriez, P.(1986) Integrity Control in Distributed Database Systems. In *Proceedings 19 th Hawaii International Conference on System Sciences.*

Widom, J., Ceri, S., and Dayal, U.(1996) *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann.

7 BIOGRAPHY

Lorena G. Gomez received her B.S. and M.S. degrees in Computer Systems Engineering (1986) and Computer Science (1990) from ITESM in Monterrey, Mexico. She is currently a doctoral candidate in the Department of Computer Science and Engineering at Arizona State University. Her research interest include Active and Object-Oriented Database Systems, and Distributed Systems. Lorena is a member of the Association for Computing Machinery, the Upsilon Pi Epsilon Honor Society in Computer Science, and the Phi Kappa Phi Honor Society.

Susan D. Urban is an associate professor in the Department of Computer Science and Engineering at Arizona State University. She received the B.S., M.S. and Ph.D. degrees in computer science from the University of Southwestern Louisiana. Her research interests include semantic and object-oriented data modeling, integrity constraints, active database systems, engineering databases, and semantic heterogeneity in heterogeneous database environments. She has served on the editorial board of IEEE Transactions on Knowledge and Data Engineering. Urban has also been actively involved with the IEEE Computer Society International Conference on Data Engineering since 1984 and served as the program co-chair for the 1998 Data Engineering Conference. She has served on the progam and organizing committees of various other database conferences. Urban is a member of the Association for Computing Machinery, the IEEE Computer Society, and the Phi Kappa Phi Honor Society.