Abstract

Throughout many research and development projects for active rule systems, active rules are implemented with different syntax and semantics. It becomes one of the stumbling blocks to apply active database systems especially in networked heterogeneous multidatabase environments. Utilizing the recent development of CORBA and ODMG standards, an active rule system is developed for heterogeneous ODBMS. Active rules represented in an ECA type are managed by a rule base. When events included in application database programs are detected, triggered rules by the events are retrieved from the rule base over network and interpreted dynamically. To ensure a fast interpretation, the rules are stored in a bytecode format. Separation of rule-managing function will allow updating rules anytime without modifications in application programs or DBMS. The change in rules will be reflected instantly in application programs via dynamic interpretation. The active rule system described is applied for integrity maintenance of spatial objects. With experimental results, overheads of byte code interpretation and runtime retrieval of triggered rules through network are discussed.

Keywords: Active rules, Dynamic interpretation, Byte code, Heterogeneous ODBMS, Integrity, Spatial Database.

1. Introduction

1.1 Motivation

As an effort to provide the users with the knowledge management features that are orthogonal to database programming, database community has conducted many research and development projects to utilize active rules for such applications as integrity constraint maintenance, view maintenance, and workflow management [18]. However, because active
database systems have been designed and implemented with different syntax and semantics of active rules, it is hard to build applications (such as e-commerce [10] and IDE [11]) with heterogeneous multiple database systems in distributed environments. The main idea of this paper is to provide active rules for heterogeneous ODBMS by taking advantage of the recent development of CORBA[12] and ODMG[2] standards.

The active rule system described in this paper has been developed as a component of a larger middleware project called SDBC (Spatial Data Base Connectivity)[3], which give programmers a singe, object-oriented view of database and programming while relieving them from the burden of knowing the implementation-specific details of individual databases such as the physical location and the ODBMS process architecture. The key to such transparency is the extension of the ODMG object model [2] with a set of spatial data types and operators based on OpenGIS [13]. In addition, SDBC incorporates the global management function to ensure the consistency of SDBC transactions accessing multiple databases. By linking with SDBC modules, the active rule system is applied to definition and maintenance of the integrity of spatial data.

Commercial database systems provide only limited supports for data integrity checking because of the high computing cost. However, integrity maintenance becomes more important especially in distributed environments because the behavior of the whole system is not predictable unless the integrity of the shared data is maintained properly. When there is no applicable service for data integrity checking, the application programmer usually inserts his own code for checking constraints in his program. He has to do this whenever there is a database operation that need to check of constraints. As the number and complexity of integrity conditions increase, this will become unmanageable by the application programmers. It becomes almost impractical to demand every programmer to have the exact knowledge on the complex integrity constraints and to code them correctly.

The proper solution would be relieving the programmers of the responsibility of data integrity checking and letting a separate system to handle it. The integrity constraints can be collected by specialists who have the proper domain knowledge for the required constraints in detail and can be stored in a rule base as a set of rules. The active rule system, then, can monitor the database accessing activity of application programs and check whether the constraints are being satisfied when required. The integrity constraints can be expressed as ECA-rules (Event-Condition-Action rules).

Architectures for active database systems can be classified into layered, built-in, and compiled [18]. Because, in a built-in architecture, all active database components are included in the database system itself, the layered and compiled architectures are applicable for
heterogeneous DBMS’s. The compile approach has the advantage of fast running of precompiled triggered rules without the intervention of run-time event monitoring [20]. However, because all the triggering events must be detected in compile-time, the compile approach could not reflect runtime facts such as real time and memory allocation. The other drawback of the compile approach is that the compiled code should be recompiled when the corresponding rules are changed.

The active database system presented in this paper implements the layered architecture. Based on the uniform interface to heterogeneous ODBMS’s, triggering events are detected and active rules are executed. Because the active rules are considered as global to distributed databases, a rule base is managed and interacts with event handlers through network in order to provide triggered rules for detected events. Once the triggered active rules are sent to a event handler, they are interpreted on the data from corresponding DBMS. In order to minimize the overhead of run-time interpretation of active rules, we save the active rules in terms of Lua byte code [9]. Like Java, Lua supports a stack-based virtual machine. We select Lua because it is portable embedded language (written in C) and its size is small (about 130 Kbytes).

This paper is organized as follows. In the remaining of this section, we introduce the overall middleware environment for spatial ODBMS. Section 2 summarizes the related work. The syntax and semantics of active rules are described in section 3 with an example. In section 4, the overall structure and behavior of the dynamic rule interpretation system is described. Section 5 gives the results of some experiments and discusses the overheads of byte code interpretation and network communication. Section 6 concludes the paper and discusses some future work.

1.2 Layered Architecture of Active Rule System for Spatial ODBMS

The overall architecture of SDBC and active rule system is depicted in Figure 1. It employs the OMG Object Request Broker (ORB) for transparent network-level access. Right above this ORB layer lies the SDBC access layer, which implements the transparent access provider (TAP) library. An application program is linked to this TAP library and runs as a client process. The shaded ovals in the figure represent process boundaries.
Below the ORB layer lies the SDBC server layer, which implements the LDA (Local DBMS Adapter) and the SDBC Coordinator consisting of GDDM (Global Data Dictionary manager), GTM (Global Transaction Manager), and Active Rule Manager. Active rules are compiled into byte codes and saved in the Rule Base. LDA is linked with Lua byte code interpreters and executes triggered rules. Either an LDA or an LDA with an ODBMS library runs as a process depending on the process architecture of a specific ODBMS. The SDBC coordinator exists as another process.

Four functional modules of the SDBC server layer, LDA, GDDM, GTM, and Global Rule Manager are implemented as CORBA objects and registered in the ORB. LDAs hide ODBMS-specific implementation details. The interfaces of these objects, called SDBC server layer interface (SLI), are described by the OMG interface definition language (IDL), and the ORB directs a client request to an appropriate object. The ORB itself supports the method shipping.

SDBC assumes that the underlying ODBMS conforms to the ODMG standard and supports a set of spatial data types and operators defined by SDBC. In case the ODBMS does not provide these spatial data management functions as in most commercial implementations, the SDBX (Spatial DataBase eXtension) layer is provided. SDBX implements SDBC-defined spatial data types, operators, and spatial indexes on top of commercial ODBMSs.

Although an application programmer can access SLI directly through the ORB, this approach has the following limitations. First, an application programmer has to know how to deal with CORBA. Second, because IDL does not support function overloading, runtime polymorphism, parameterized types, and so on, an application programmer cannot take
advantage of the full expressive power of popular ODBMS programming languages like C++. Third, the performance of certain applications is bound to be limited only with the method shipping mechanism. The TAP in the SDBC access layer has been introduced to overcome these limitations.

Hiding the existence of ORB, the TAP shields an application programmer from the burden of knowing ORB programming. It solves the mismatch in the expressive power between SDBC C++ OML and OMG IDL. It provides facilities for managing the object cache in the client, and supports the object shipping mechanism. Applications that have frequent access to a set of objects can benefit from this mechanism.

2. Related Work

Researches have been centered around three aspects of active database systems: optimizing the computation of the condition part, defining rule languages, and efficient event monitoring. Studies on rule languages have been published in [1, 4, 5, 15]. [4] describes a rule language with flexible execution modes for active databases. Their language expresses not only simple triggering events but also composite triggering events and supports nested transaction model. [1] extends their previous rule language, O++, to express integrity constraints and triggers. Triggers can be used to monitor complex triggering events and to call appropriate procedures. The active rule defined in Section 3.1 provides a general syntax for conditional and action statements. Event types defined in our rule language are limited to object modifications (i.e., create, delete, and update). Temporal or application-defined events could be included in future extensions.

Efficient event monitoring techniques have been reported in [1, 4, 5, 7, 17]. [5] incorporates the monitoring system directly into the kernel of the database system to support a fast event detection. [7] uses Petri-net to detect events efficiently. [1, 17] suggests languages that class designers can use to define rules when designing the schema. In both languages, rules are defined inside the corresponding class definitions. In [1], the rules are fired when the constructor function or member functions of the corresponding class are called, and the rules' conditions are met. In [17], class designers can specify which member variables should be monitored to detect events. Here rules are checked for processing if the constructor function is activated, or if the monitored variables are modified.

[4] separates rule management from the application program. An external event detector monitors the application program for detecting events. When detected, an event signal is sent by the event detector to the rule manager. Not all events raise signals; only those event listed in the event-list inside the event detector are signaled. Which events to detect is determined by rule definitions. A rule is created through object manager. The object manager, when requested
to create a rule, asks the event detector to add the event of this rule to the event-list, notifies the
rule manager about this rule, and asks the condition evaluator to add this rule to its condition
graph. When an event signal is received, the rule manager checks which rules are to be fired
and turns over the corresponding rule-firing entity that contains information about these rules to
the condition evaluator. The condition evaluator executes the necessary query plans with the
help of the object manager to evaluate the conditions of these rules and returns selected rules.
Finally the rule manager, again with the help of the object manager, executes the action code of
the selected rules. Since the rule processing code is separated from the application program, [4]
provides better chance for optimizing rule conditions and rule management compared to [17].
However, the event detector has to monitor the application program constantly, and when an
event is detected, the corresponding rule has to be searched through in the potentially large rule
base. Our approach is similar to that of [4] in that it separates the rule base from the application
program and searches rule base whenever an event is detected. However, in our approach, the
database schema defined in ODL is preprocessed in order to provide event-handling codes.

The optimization techniques for rule conditions are published in [6, 14, 15, 16]. [15] compiles the rules beforehand and links the resulting object code with the triggering monitor. This improves the rule processing time considerably. However, each time a new rule is added or
an existing rule is modified, the rule module has to be re-compiled and re-linked with the
triggering monitor. [6, 16] suggests techniques based on discrimination networks. A
discrimination network accepts a database operation as input and outputs corresponding rules.
[14] suggests incremental evaluation methods. It observes that the same condition tends to
repeat in several rules and saves the result of the computation of one condition to reuse it later.
In this research, we concentrate on optimization of processing spatial objects. Utilizing the
spatial indexes (i.e., R-tree and R*-tree) supported by SDBC, the evaluation of spatial operators
(e.g., distance, intersect, overlap, and contain) could be optimized.

3. Active Rules

3.1 Syntax of Active Rules

The syntax of ECA rules consists of event, condition, and action as follows.

\[
\text{active_rule} = \text{DEFINE rule_mode RULE rule_id FOR } \{ \text{class_name_list}\} \text{\'} \text{;'} \\
\text{VAR var_list}\text{;'} [\text{ARRAY OF } | \text{LIST OF } | \text{SET OF}] \text{class_name } \text{\'} ; \\
\text{EVENT trigger_event} \text{\'} ; \\
\text{CONDITION cond_stmt } \text{\'} ; \\
\text{ACTION action_stmt} \text{\'} ; \\
\text{END_RULE } \text{\'} ; \\
\]

The rule_mode specifies whether a rule is regarding multiple object instances or not (i.e.,
set-oriented or instance-oriented). Active rules in this work are defined for specific class(es) and can be referred to as targeted rules. Targets for a rule are defined in the FOR clause. The event in active rules can be one of transition operations (i.e., create, update, and delete) of persistent objects.

\[
\text{Rule\_mode} = \text{“SET-ORIENTED”} | \text{“INSTANCE-ORIENTED”} \\
\text{trigger\_event} = \text{INSERT} \text{‘(‘} \text{instance ‘)}\text{’} \\
| \text{DELETE} \text{‘(‘} \text{instance ‘)}\text{’} \\
| \text{UPDATE} \text{‘(‘} \text{instance} \text{[‘.‘} \text{attribute } \text{‘)}\text{’} \\
\]

The condition statement in a rule should be evaluated true before the rule can be triggered. Conditional predicates can be connected by AND or OR operators recursively.

\[
\text{cond\_stmt} = [ \text{‘(‘} \text{cond\_pred} \text{[‘)’]} \\
\{\text{connector} [ \text{‘(‘} \text{cond\_pred} \text{[‘)’]} \} \\
| [ \text{‘(‘} \text{cond\_stmt} \text{[‘)’]} \text{connector} [ \text{‘(‘} \\
\text{cond\_pred} \text{[‘)’]} \\
| [ \text{‘(‘} \text{TRUE} \text{[‘)’]} \\
\text{cond\_pred} = [ \text{NOT} ] \text{logical\_expression} \\
| \text{[NOT] spatial\_operation} \\
| \text{[NOT] collection\_expression} \\
| \text{[NOT] user-defined\_function} \\
\text{collection\_expression} = \text{universal\_quant} \\
| \text{existential\_quant} \\
| \text{membership\_expr} \\
| \text{aggregate\_opr} \\
\text{connector} = \text{AND} | \text{OR} \\
\]

Actions in rules can be general functions such as system functions or user defined functions. One of the frequent actions is ABORT, which cancels all the changes made by the current transaction.

\[
\text{action\_stmt} = \text{action} \text{action} \{\text{AND action\_stmt}\} \\
\text{action} = \text{built-in\_procedure} | \text{user-defined\_procedure} \\
\text{built-in\_procedure} = \text{create\_procedure} \\
| \text{replace\_procedure} \\
| \text{remove\_procedure} \\
| \text{update\_procedure} \\
| \text{INFORM\_INTEGRITY\_VIOLATION} \\
| \text{ABORT} \\
\]

3.2 Active Rules for Integrity Constraints of Spatial Objects

Integrity constraints for spatial objects specify rules that should be satisfied whenever spatial objects and their attributes are created, updated, or deleted. In principle, they are similar to those of ordinary databases. However, the coexistence of geometry, topology, attributes, and multiple layers in spatial databases makes the semantics more complicated. In the following, the characteristics of integrity constraints for spatial objects are summarized.

- Geometrical Constraints: They restrict the output of geometrical operations using spatial coordination values for either single or multiple spatial objects in a geographic layer.
SDBC supports primitive geometry (e.g., point, curve, surface, line, and polygon) and collection geometry (e.g., multisurface, multicurve, multipoint, multipolygon, and multilinestring).

- Topological Constraints: They restrict the topological relationship among spatial objects, which can be defined in Boolean operations. Some examples of topological relationship are equals, disjoints, touches, crosses, within, overlaps, contains, and intersects.

- Attribute Constraints: They can be classified into spatial domain constraints and non-spatial domain constraints. Spatial domain constraints are related to geometrical dimension (e.g., point, line, and region) and geometrical attributes such as color and type of lines. For example, a subway route should consist of a set of line segments and points that represent stations. Non-spatial domain constraints restrict the types, range of values, and units of the non-spatial attribute of abstracted spatial objects. For example, we can restrict the age of a sewer pipe should be less than 20 years.

- Interlayer Constraints: They restrict the relationship of geometry, topology, and attribute among objects from different geographical layers. For example, pipes in the waterworks layer and those in the sewer layer should not intersect.

The constraints in the above list can restrict either singular values or aggregate values. The example of aggregate operators includes AVG, MAX, MIN, SUM, and COUNT. Using the syntax defined in the previous section, an integrity rule can be defined as follows.

**Example 1.** This example is an set-oriented rule defined for the classes BuildingLayout and RoadLayout. It states that apartments or tenements cannot constructed within 50 meters from any express way.

```
DEFINE SET-ORIENTED RULE R2 FOR (BuildingLayout, RoadLayout);
VAR building: BuildingLayout; roads: SET OF RoadLayout;
roads := QUERY(temp FROM RoadLayout WHERE temp.type = EXPRESS);
EVENT INSERT(building);
CONDITION (building.type = APARTMENT OR building.type = TENEMENT) AND EXISTS temp IN roads: Distance(building.area, temp.area) < 50;
ACTION INFORM_INTEGRITY_VIOLATION;
END_RULE;
```

### 3.3. Semantics of Active Rules

An active rule is defined as either an instance-oriented or a set-oriented rule. An instance-oriented rule is applied to a single instance and a set-oriented rule is applied to a set of instances of either the same class or different classes. Distinction between an instance-oriented rule and a set-oriented rule is made by the users and considered by the preprocessor for optimized code generation. Even a set-oriented rule is triggered many times by an iterative insertion operation.
for instance, it is sufficient to fire the rule only one time for the whole inserted instances.

We assume that the integrity of database is maintained at the beginning of a transaction and it should be assured before the transaction is committed. This means that the integrity of the database might be violated temporarily during a transaction. For example, during a pair of delete and paste operation of a design object, the disappearance of the object could violate design rules temporarily. We allow this kind of temporal violation of integrity rules by deferring the evaluation of integrity rules to the end of transactions.

Events can be classified as external events and internal events. The external events are included in application programs and the internal events are included in the action parts of active rules. Through the internal events, a series of rules can be triggered by an external event. When the triggering relationship among active rules forms a cycle, it might cause an infinite loop of triggering at run-time. A triggering graph can be defined as a directed graph $G(V, E)$, where $V$ is a set of vertices and $E$ is a set of directed edges. Each rule in the rule base is represented by a vertex. There is an edge $e_j$ from a vertex $v_j$ to another vertex $v_2$ if the action of the rule corresponding to $v_j$ can be considered as the event of the rule corresponding to $v_2$. In order to prevent infinite loops, no new rule is allowed to form a cycle in the triggering graph.

4. Dynamic Rule Interpretation System

The basic idea is twofold: rules are managed by a separate entity, the rule manager, and whenever a rule is required, it is fetched and interpreted dynamically. To ensure a fast interpretation, the rules are stored in a bytecode format. Separation of rule-managing function will allow updating rules anytime without modifications in application programs or DBMS. The change in rules will be reflected instantly in application programs via dynamic interpretation. Figure 2 shows the overall picture of our rule interpretation system. In the left side, there is a rule manager which is responsible for compiling rule programs (or rule definitions) into bytecodes and servicing them to application programs. On the right side, we see an application program linked with a rule and Lua library that help the application program in requesting and interpreting rules during the run-time.
4.1 Rule manager

The rule manager accepts rule programs, converts them into bytecodes, and responds to outside requests for the rules. For the bytecode format, Lua bytecode is being used.\textsuperscript{1} Conversion to Lua bytecodes is being done by the Lua compiler. To use the Lua compiler, the rule program is first transformed to Lua program.

4.1.1 Symbol table

When changing rule programs into Lua programs, we need the attribute information for system-provided classes. Rule programmers can use system-provided classes without including their definitions, and the rule system should include them when changing the rule programs into Lua programs. The needed attribute information of the system-wide classes are extracted from the schema definitions written in ODL(Object Definition Language) by the ODL preprocessor (as shown in Figure 2) and stored in the symbol table.

\textsuperscript{1} Lua bytecode is chosen, because the Lua interpreter is small and fast. Also Lua programming language provides a convenient mechanism to communicate between the host language and Lua (Lua is an embedded language). Using this feature, we can implement time-consuming DB operations, especially spatial operations, in the host language, and call them from Lua programs.
4.1.2 Rule preprocessor

The rule preprocessor is responsible for transforming the rule programs into Lua programs. With the help of the symbol table, it first generates codes for loading all objects referred in the rule program in the interpreter stack. The condition and action part is transformed into an if-statement. If the action part updates the database, it is considered to be another event to be checked. The rule preprocessor inserts codes for requesting rule-checking in this case. Below we show an example to explain how a rule program is converted into a Lua program.

```
DEFINE INSTANCE-ORIENTED RULE R3 FOR(SewerPipe);
VAR sp: SewerPipe;
EVENT INSERT(sp);
CONDITION (sp.type = Transport OR sp.material = "Steel")
AND sp.diameter < 350;
ACTION update(sp.diameter, 350);
END_RULE;
```

The above is a rule program written in a rule language defined in Section 3. It defines an instance-oriented rule with name R3. The target class and event this rule is applied is "SewerPipe" and "INSERT" respectively; that is if an object of "SewerPipe" is inserted (or created), rule R3 should be applied. This program will be transformed into the following Lua program.

```
sp = {oid, type, material, diameter}
sp.oid, sp.type, sp.material, sp.diameter = load_object("SewerPipe")
function Exec_R3(sp)
  if ((sp.type == Transport or sp.material == "Steel")
      and sp.diameter < 350
    then
    update("SewerPipe", sp.oid, "diameter", "350")
    raise_internal_event(1,"SewerPipe","UPDATE","diameter")
  end
end
Exec_R3(sp)
```

In the first two lines above, an object of "SewerPipe" class is defined and loaded. Without loading this object, the access to the field of it as in the fourth line will have no meaning. In the fourth line, "type", "material", and "diameter" attributes are examined, and if the condition is satisfied, the "diameter" attribute is updated. Since this rule is updating the database, the preprocessor raises “UPDATE” event for class "SewerPipe" in the 7th line. This
event will be collected in the event set and handed over to the rule manager when the above rule is executed (refer to the rule triggering mechanism in Section 4.2.3).

Another example is shown below for a set-oriented rule.

```plaintext
DEFINE SET_ORIENTED RULE R1 FOR(SewerPipe, WaterPipe);
VAR sp: SewerPipe; Wps: SET OF WaterPipe;
wps := QUERY(temp FROM WaterPipe)
EVENT INSERT(sp);
CONDITION EXISTS temp IN wps: Intersect(sp,temp);
ACTION INFORM_INTEGRITY_VIOLATION;
END_RULE;
```

The above rule is set-oriented because a set of "WaterPipe" objects are involved. A set-oriented rule is more complicated to transform than an instance-oriented rule. When it involves one or more spatial queries and spatial operations, spatial indexes are created and filtering operations are inserted. The resulting Lua program is as follows.

```lua
sp = {oid, type, material, diameter}
wp = {type, material, diameter}
sp.oid,sp.type,sp.material,sp.diameter =
    load_object("SewerPipe")
function Exec_R1(queried_class)
    create_RTree_key("SewerPipe",0)
    local count =
        filter_objects("SewerPipe", "s_RTree_WaterPipe")
    local objIndex = 0
    while objIndex < count do
        result = LineString_intersects_LINESTRING
            ("SewerPipe","WaterPipe",objIndex)
        if (result == 1) then
            violate_integrity(1,"R1")
            t_abort()
        else objIndex = objIndex + 1
        end
    end
    Exec_R1(sp)
end
```

In set-oriented rules, the condition part is closely related to the query part. The rule preprocessor remembers the relationship between them during the parsing stage and generates codes accordingly.

4.2 Rule execution

The application program should collect events that need to be checked for rules and initiate rule-checking processes for them. Rules are requested at the same time right before the "commit" statement. All events collected so far are sent to the rule manager, and a list of rules
(in Lua bytecode) is received in return. The bytecodes are stored in "active rule list" and interpreted one by one by the Lua interpreter.

|/* Functions for manipulating database objects */| void save_object (char *t_class, d_Ref_Any objRef, char *oid);
|void load_object (char *t_class);
|void create (char *t_class);
|void remove (char *t_class, char *oid);
|void update (char *t_class, char *oid, char *attr, void *new_val);
|/* Function for transaction */| void t_abort (void);
|/* Functions for spatial queries and rule conditions */| void create_Rtree_key (char *t_class, int extension);
|int filter_objects (char *t_class, char *rtree);
|void load_filtered_object(char * t_class, int objIndex);
|/* Functions for SDBX spatial operations */| void LineString_intersects_LineString (char *t_class1, char *t_class2);
|void LineString_overlaps_LineString (char *t_class1, char *t_class2);
|void LineString_contains_LineString (char *t_class1, char *t_class2);
|void Polygon_intersects_LineString (char *t_class1, char *t_class2);
|void Polygon_overlaps_LineString (char *t_class1, char *t_class2);
|void Polygon_touched_LineString (char *t_class1, char *t_class2);
|void Polygon_intersects_Polygon (char *t_class1, char *t_class2);
|void Polygon_overlaps_Polygon (char *t_class1, char *t_class2);
|void Polygon_touched_Polygon (char *t_class1, char *t_class2);
|void distance_of_LineString_LineString (char *t_class1, char *t_class2);
|void distance_of_Polygon_Polygon (char *t_class1, char *t_class2);
|void distance_of_Polygon_LineString (char *t_class1, char *t_class2);
|/* Function for retrieving rules */| Triggered_Rule_List * fetch_triggered_rule_list (ClassEvent classEvent);
|/* Function for informing integrity violation */| void inform_integrity_violation(int violation, char *rid)
|/* Functions for integrity checking */| void check_integrity (ClassEvent classEvent, d_Ref< d_Persistent_Object> objRef, char *oid);
|Triggered_Rule * extract_triggered_rule (Triggered_Rule_List * tr);
|ClassEvent raise_external_event (char * t_class, char *event, char * attr);
|void raise_internal_event (int triggering, char * t_class, char *event, char *attr);
|/* API function for application program */| int validate_integrity (vector<d_Ref<d_Persistent_Object> >& created_objects,
|vector<d_Ref<d_Persistent_Object>&> deleted_objects,
|vector<pair<char*, d_Ref<d_Persistent_Object> >& modified_objects,
|vector<pair<char*, d_Referent<d_Persistent_Object> >& violated_objects); |

Figure 3. Rule library functions

4.2.1 Rule library

The functions called in Lua program (or Lua bytecode after compilation) but not defined should be provided in the rule library. The Lua compiler passes the string of the function name as part of arguments when converting function calls to bytecodes. The Lua interpreter, when processing a function call, retrieves this function name, finds the corresponding function pointer in the host language, and calls it. Therefore all functions in the host language that are to be called from the Lua code should be registered with the interpreter.

Figure 3 shows a list of functions in the rule library. Among them, database object manipulation functions are used to modify database objects and are schema dependent. They are schema dependent because each manipulation function should treat each class differently.
when loading or updating them. The ODL preprocessor in Figure 2 generates schema-
dependent library functions automatically from the schema definitions (written in ODL). The
rest of the library functions are schema independent. In the next section, we will show how
these functions are used in application programs or in Lua programs.

4.2.2 Rule interpretation

To check rules, the application program collects events and calls "validate_integrity()" just
before the "commit" statement. "validate_integrity()" passes the event list over to the rule
manager, receives rule codes in bytecode format, and calls the rule interpreter for each of them.
An example of an application program with rule-checking codes is in Figure 4 Rule-checking
codes which are added by the preprocessor are in bold face.

```c
#include "mylib.h"
int main(){
    d_Database pipeDB;
    ..............
    pipeDB.open("pipeFD");
    ..............
    Point p[4];
    p[0] = Point(192235,254320);
    p[1] = Point(192245,254350);
    ..............
    LineString pipe_geo(4,p);
    d_Ref<SewerPipe>sp = new(oovTopDB)
        SewerPipe(Transport,"Cast",200.0,pipe.geo);
    created_objs.push_back(sp);
    ..............
    validate_integrity(created_objs, deleted_objs,
        modified_objs, violated_objs);
    ..............
}
```

Figure 4. An example of application program with rule-checking codes added

To see how rules are interpreted, assume rule R1 and R3 in Section 4.1.2 are the rules
related to this event (creation event of "SewerPipe" object). When the application program runs,
the validate_integrity function will send an event list which contains only one event: INSERT
on "SewerPipe" class. Rule R1 and R3 will be returned (in bytecode format) in response.
Assume R1 is first processed. The rule interpreter detects "CALLFUNC" bytecode which calls
a host-language function that corresponds to Lua function "load_object()". Suppose the
registered function for "load_object()" is "lua_load_object()". The interpreter calls
"lua_load_object()" to load the current SewerPipe object just created. "lua_load_object()" should look like as follows.

```c
static void lua_load_object(void)
{
    lua_Object t_class = lua_getparam(1);
    load_object(lua_getstring(t_class));
}
```

It first retrieves the name of the class this object belongs to from the interpreter stack, and calls "load_object()" with this class name as an argument. Note this "load_object()" is a host-language function, different from the Lua function with the same name. The "load_object()" is a big case statement; for each class it calls a class-specific loading function.

```c
void load_object(char *t_class){
    if (!strcmp(t_class, "SewerPipe"))
        load_SewerPipe_object();
    else if (!strcmp(t_class, "WaterPipe"))
        load_WaterPipe_object();
    ........
}
```

Since t_class is "SewerPipe" in our case, "load_SewerPipe_object()" will be called, and this function will load the just-created SewerPipe object on the interpreter stack so that its members can be checked for the rule. The rest of the interpretation process is similar and straightforward. If a host-language function is called, find the corresponding function pointer and executes; otherwise interpret the given bytecode.

### 4.2.3 Rule triggering

Rule R3 in Section 4.1.2 triggers another rule in its ACTION part. In this case, the new rule (or rules) are processed first. If all triggered rules are processed, then we go back to the original rule list. This rule triggering can be nested. The innermost triggered rules will be processed first, and the rules in the outer layer will be processed in turn. The algorithm to handle triggered rules is shown as a flow graph in Figure 5.
We will use an example to explain the algorithm. Suppose an event has detected, and rule R4 and R5 are retrieved for this event. They are inserted to TRL[0] with "parent_index" equal to -1.

TRL[0] = \{(R4, R5), -1\}

R1 will be processed first. If R4 triggers R6, the triggered rule lists will be changed as follows.

TRL[0] = \{(R5), -1\}
TRL[1] = \{(R6), 0\}

Note the "parent_index" for TRL[1] is 0 meaning that after all rules in TRL[1] are processed we should go back to TRL[0]. Now R6 is processed. If this rule triggers other rules, TRL[2] will be created to contain them; otherwise TRL[1] is done, and we go back to TRL[0].

5. Experiments

The biggest advantage of our rule system is that it can handle constantly changing rules without recompilation of the application program or DBMS itself. The price for this flexibility is an increase in rule processing time. The processing time increases because of two reasons. One is interpretation; the other communication over network. In this section, we examine how much delay is caused by these two reasons, and suggest a couple of ways to improve it.
5.1 Interpretation overhead

Lua code, when interpreted, is about 20 times slower than a native code. To see how this slowness affects our rule system, we picked the rule in Figure 6, coded it in C++ and Lua respectively, and ran them 10 times.

```
DEFINE SET-ORIENTED RULE R7 FOR (BuildingLayout);
VAR building: BuildingLayout; buildings: SET OF BuildingLayout;
buildings := QUERY(temp FROM BuildingLayout WHERE temp.type == KINDERGARTEN);
EVENT INSERT(building);
CONDITION building.type == GAS_STATION AND
    EXISTS temp IN buildings : Distance (building , temp) < 50;
ACTION INFORM_INTEGRITY_VIOLATION;
END_RULE;
```

Figure 6. A set-oriented rule.

Figure 7 shows the average time taken by each code. Lua code only shows 2 to 3% increase in processing time. The reason is that the rule in Figure 6 requires a number of spatial operations to be done, which consumes most of the processing time, but these spatial operations are already provided by the rule library in native code. Lua program just calls them, and the pre-compiled native code runs for it. This case, however, is not special. Most time-consuming rules are spending most of time in processing spatial operations. If the rule does not contain any spatial operation, it will be processed very quickly, and Lua code will not cause a significant delay over native code. On the other hand, if the rule contains spatial operations, it will take time because of the spatial operations, but again Lua code and native code will not show much difference in processing time since the spatial operations are run in native code in both cases anyway.

Figure 7. Comparing Lua code to native code in rule execution time.
5.2 Communication overhead

A more serious delay comes from the communication between the application program and the rule manager. Since the rules are fetched in run time, the application program should wait until the rules arrive. In our case, we are using CORBA mechanism for communication, and the communication overhead is as shown in Figure 8. The time is measured inside our laboratory, and the internet traffic delay is not counted; but the overhead already reaches about 1.5 seconds. However we can observe that the overhead stays relatively same regardless of the number of fetched rules (less than 1% increase for 100 rules compared to 1 rule). This means fetching a large rule set at once is better than fetching smaller rule sets several times, in terms of communication time. Also, since the overhead is relatively a constant, it becomes less significant as the rule processing time is increasing.

Figure 8. Communication overhead of rule fetching.

Still the delay is not negligible, and it seems the communication overhead is unavoidable and is the price we have to pay for to get the flexibility in handling dynamically changing rules. However we can reduce the communication time in several ways. One is to use more improved communication mechanism than CORBA. The inefficiency of CORBA in communication overhead is well pointed out by researchers[8]. Other ways are caching or pushing. By caching some of the rules in the local system and by providing a local rule manager, we can improve the processing speed considerably. Pushing will be initiated by the rule manager: it can push changed or newly created rules to the local rule manager. The local manager will decide whether to cache these pushed rules depending on the relevance of them to the local DBMS.

6. Conclusion

In this paper, we present an active rule system that is based on dynamic interpretation of mobile rule codes. A prototype of this system has been implemented as a component of a larger
middleware system that provides ODMG interfaces and spatial extensions to ODBMS. By implementing it as a part of a middleware, heterogeneous ODBMS in distributed environment can be provided an active rule system in system independent manner. The active rules are in ECA type and used for validating integrity constraints in this prototype.

A global rule base manages the active rules and interacts with local event handlers to provide triggered rules for detected events. Active rules are saved in Lua byte codes and interpreted efficiently on the data from local database systems. Comparing to other architecture of active rule systems (i.e., tightly coupled built-in architecture or compiled architecture), this approach can separate the rule-managing function from local DBMS and enables the rules to be updated without any changes in application programs. Any changes in the rule base will be reflected instantly via dynamic interpretation while application programs run on a local DBMS. The overheads of transmitting triggered rules from the rule base turn out insignificant comparing to set-oriented spatial operations.

There are still rooms left for further improvements. Firstly, local DBMS can maintain the cache of triggered rules and reduce the overheads of network communcation. In this case, any changes in the global rule base should be pushed to the local cache. Secondly, rule conditions from different active rules often include common parts. Analysis using dependency graph can identify the shared parts and the optimal order of evaluation for the given rule conditions [21]. Thirdly, we can receive valuable information for optimization from the target ODBMS. We need to devise more general approaches to optimize spatial operations in various application contexts.

References
1989.


