The Implementation of Rules in the Two-Stage Entity Relationship Model

by

Jinsong Mao

A Project Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fufillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: Computer Science

Approved:

____

Cheng Hsu, Project Supervisor

____

David L. Spooner, Sponsoring Faculty Member

Rensselaer Polytechnic Institute
Troy, New York

August 1994

Part 1
Introduction

1.1 Background and Problem Statement

In order to support the enterprise information systems integration and adaptiveness, major efforts have been undertaken by the metadatabase research group at RPI (under direction by Professor Cheng Hsu) in the past years and the accomplishments include (1) Two-Stage Entity Relationship (TSER) modeling methodology; (2) Metadatabse model and management system (including a rulebase model); (3) Global Query System (4) Rule-Oriented Programming Environment for supporting the execution of the global-rule among heterogeneous, distributed systems([Hsu92], [Bouziane91], [Babin92], [Cheung91]).

The TSER modeling includes semantic level modeling(SER) and operational level modeling (OER). The SER modeling is used for system analysis and high-level design and it has two constructs, one is SUBJECT which contains data items , functional dependencies and intra-subject rules to serve as the functional units(or views) of information(data + knowledge); the other is CONTEXT which contains the inter-subject rules to provide the control knowledge among the related information units. The OER modeling is used for logical database and rulebase design and it has four constructs which are Entity, Plural Relationship(PR), Mandatory Relationship(MR), and Functional Relationship(FR) and they can be mapped from the SER level([Hsu92],[Bouziane91]).

We can use the TSER methodology to model both individual local application systems and the global enterprise information systems. The data and knowledge modeled by TSER can be populated into the metadatabase(including the rulebase) according to the GIRD(Global Information Resource Dictionary) schema.

We can use TSER to model the rules and use the rulebase model to store the rules in metadatabase, but we also need to provide some facilities to support the execution of the rules, not just representing and storing the rules. For the single local application system, we need to provide facilities to automatically generate the executable code for each local-rules based on the specific local DBMS and the metadata about the rules. For the enterprise containing multiple local application systems, it's impossible to directly generate the executable code unless the real heterogeneous, distributed DBMS has emerged. ROPE develops the local-shells technology to support the execution of the global-rules, but one of the key element for global-rule execution is to decompose the global-rule into sub-rules to maintain the precedence constraints among operations in the global-rule, [Babin 93] has developed the decomposition algorithms and it needs to be implemented so that the ROPE system can really work.

1.2 Project Goals and Significance

The goals of this project are as follows:

(1) Given the local-rules in TSER model and the metadata about the local-application system, automatically generate the executable code for the execution of the local-rules based on the local DBMS.

(2) Implement of the global-rule decomposition algorithm of ROPE to produce an equivalent set of sub-rules so that the serial behavior of these sub-rules is equal to the original global-rules.

Solving the above two problems provides facilities for the execution of both the local and global rules in the enterprise containing multiple application systems. For the rule-modeler, he can concentrate on the semantic correctness of the rules and do not need to know the coding details and even do not need to be a programmer; for the application-programmer, he can directly use the rules and embed the rules in his programs without worrying about the possible changes to the rules and the coding details of the rules. In case of executing the global rules, the implementation of the decomposition algorithm can guarantee that the distribution of the rule into local-shells retains the global consistency and synergism as the original model.

Besides the above practical significance, this project requires in depth understanding of the existing systems and ideas such as TSER data structures and modeling methodology, Metadatabase system, Global Query System and Rule-Oriented Programming Environment, and it involves database/rulebase modeling and processing, algorithm analysis and implementation, software engineering, and information integration in the heterogeneous, distributed environment, etc. Therefore, this project is appropriate for a Master’s project.

1.3 Summary

The project’s purpose is to implement the rules in the Two-Stage Entity Relationship Model to allow the rule execution in addition to the rule modeling and rule storage.

This project has successfully implemented the local-rules and can automatically generate the local fact base and the executable code for the local-rule execution. This project also has implemented the global-rule decomposition algorithm for supporting the Rule-Oriented Programming Environment(ROPE) to execute the global-rules.

The results and the underlying methodology of this project could be used for supporting the enterprise information systems with large amount of business operation rules which need to be modeled, managed and used. In addition, combined with the existed Metadatabase technology and ROPE shell technology, it can even support the evolution and integration of the different data and knowledge systems in the autonomous, distributed and heterogeneous environments. The implemented systems have been tested in the Order Processing System, Process Planning System, Shop Floor Control System and Product Database System in the Computer Integrated Manufacturing environment at RPI’s metadatabase research group.

Part 2
Technical Descriptions

This section contains the detailed description of the local-rule implementation and global-rule decomposition.

2.1 Local-Rule Implementation

The basic idea for local-rule implementation is to automatically generate the detailed local DBMS system codes(such as the code for creating the factbase, and the code for fetching data from factbase, evaluating the condition and executing the actions of the rule) for the local-rule execution. Current implementation is targeted to Oracle DBMS version 6 and it can generate (1) Oracle SQL view for creating the factbase, and (2) Oracle Pro*C code for executing the rule. The local-rule syntax will be the same as the TSER rule grammar[Babin 93] except that all the data items and routines are located in the same local application system. It contains three major parts which are trigger, condition and actions.

Based on the current existed system, we can modeling the rule through the Information Base Modeling System (IBMS) and store the rule in the rulebase of the Metadatabase system. We can easily retrieve a specific rule out from Metadatabase or directly from IBMS. Then, the local-rule system will generate the necessary codes from the rule text according to the algorithm following the next major steps:

Step 1: Parsing the rule text, gathering the trigger information and build the rule-tree for later processing.

Step 2: Determine all the necessary and relevant data items which will be used for constructing the factbase.

Step 3: Generate the MQL queries (views).

Step 4. Generate the local queries (Oracle SQL views).

Step 5. Generate the local code fro rule execution (Oracle Pro*C code).

Next, we provide more detailed description of each steps.

2.1.1 Local-Rule Language and Parsing

For the local-rule implementation, we re-use the parser which has been developed for ROPE system and build the rule-tree for later processing.

The original rule language[Bouziane91] has been extended by Babin Gilbert [Babin 92] for specifying more precisely the origin of data items, etc. The data item identifier in the rule will be like this: <view specifier>.<application name>.<item name>@<determinant> instead of just <item name>. (the "<-" defined in [Babin 92] has been changed to "@" to avoid ambiguity during parsing).

The following is the local-rule example which will be used in the following sections.

```
(rule2a) (IF every_time(-1,-1,-1,1,0)
            AND (ORDER_PROCESSING.QUANTITY@ORDER_ITEM > 10)
            AND (sum(ORDER_PROCESSING.COST@ITEM,
                  ORDER_PROCESSING.CUST_NAME@CUSTOMER = "maoj",
                  ORDER_PROCESSING.DATE_SCHED@ORDER_ITEM)
                   > 100)
          THEN ORDER_PROCESSING.S_ADDR@CUSTOMER := "send notify";
notify_operator();)
```

For the local-rule implementation, the rule text being manipulated is reversed from Metadatabse and the item identifier becomes <view specifier>.<application name>.<item name>@<oepr name> because we can determine the OE/PR from the metadata at this point. The <view specifier> is used for defining the independent queries to relate some data items in some particular fashion. Without it, we can not precisely model some semantic meaning by the rule.[Babin 93]

The data structure for the node of the rule-tree is defined as:

```
typedef struct ttnode
    {
    tptr nvalue;                 /* the value of the node */
    int ntype;                   /* the type of the node, such as constant,data item,etc */
    int valuetype;               /* the type of the value of this node */
    int nb_children;             /* the number of children of this node */
    struct ttnode **children;     /* the children list of this node */
    char **sysnames;             /* un-used for local-rule */
    char *executed_in;           /* un-used for local rule */
    char *applname;              /* the application name */
    char *alias;                 /* the alias for independent queries */
    char *oepr;                  /* the OE/PR name for this node */
    char *func_id;               /* the function identifier for aggregate function */
    } tnode;
```

The rule parser also extracts the trigger information from the condition of the rule (the trigger information is defined as part of the condition of the rule in the current TSER rule syntax using the special trigger operator), removes trigger information from the rule text (so the rule-tree being generated by the parser will not contain the trigger information any more). Then, we store the rule identifier and its trigger information in a specific file for later processing (such as local timer system will use the information in this file).

2.1.2 Preparation of Data Sets

All data items appeared in a rule are directly or indirectly related in the structural model (OER model), and the system should determine precisely how these data items are related to each other. If the rule modeler specify independent sets of related data items, we need to generate the related independent queries for each independent sets defined by <view specifier> or <alias>. (As explained in 2.1.1, the syntax for data item is alias.application.item@(oepr name)) where alias is used for the independent queries).

After step 1, we have created a rule-tree, now we need to collect and organize the information such as (1)the independent queries in this rule (if no alias, we assume there is only one default independent query); (2) The set of all data items that must be retrieved by a independent query; (3) The set of items modified by assignment statement; (4) All the aggregate functions used in the rule, etc.

For the local-rule implementation, we re-used and customized the data structures and functions developed by Yi-cheng Tao for the ROPE message generation. These data structures are used for gathering the above basic information and are explained in details as following.

The ttIset structure is used for storing all the information related to a independent query.

```
typedef struct ttIset
{
        char *viewname;              /* the name of the independent query */
        char **appllist;             /* un-used for local-rule */
        char **objlist;              /* all object related to this independent query */
        char ***D_of_obj;            /* data items in each above objects */
        char ***A_of_obj;            /* the set of items modified by assignment in a obj*/
        int  *Ind_of_Temp;           /* flag for the types of update */
        char ***Alias_of_A;          /* independent query the updated item belong to */
        char **D_of_all;             /* union of items in all D_of_objs */
        char **Icode_of_all;         /* the itemcode for each item in D_of_all */
} tIset;
```

The following ttItab structure is used for storing the information about a data item appeared in the rule.

```
typedef struct ttItab
{
        char *viewname;              /* which view this data item belong to */
        char *itemname;              /* the data item name */
        char *oeprname;              /* the oepr name this item belong to */
        char *applname;               /* the application this item belong to */
        int  iskey;                  /* whether this item is a key item */
        char *itemcode;                  /* itemcode of this item */
        char *newid;                      /* modified for local-rule for item identifier */
         int  status;                     /* whether it is Updated item,  or Retrieved item */
         int  is_nil;                     /* whether it is assigned NIL to be deleted */
        int  is_used;                     /* whether it is used before */
        int  is_final;                     /* Added for local-rule, whether it will be in final factbase */
} tItab;
```

The following ttIoepr structure is used for storing information about a OE/PR.

```
typedef struct ttIoepr
{
        char *applname;                  /* the application this oe/pr belong to */
        char *oeprname;                  /* the oeprname */
        char *lc_oeprname;               /* the local table name related to this oepr */
        char **key_of_oepr;              /* the key of this oepr */
        char **nonkey_of_oepr;           /* the non-key items of the oepr */
        char **itemcode_of_key;          /* the itemcode of the key items */
        char **itemcode_of_nonkey; /* the itemcode of the non-key items */
} tIoepr;
```

The following ttIagg structure is used for storing information about each aggregate functions such SUM, MAX, MIN, AVG, EXISTS, and COUNT.

```
typedef struct ttIagg
{
        char *viewname;                  /* which independent query this aggregate function is related to */
         char **Gk;                      /* it is grouped by which data items */
        char *Sk;                    /* condition for this aggregate function */
        char *fk;                    /* the name of the aggregate funct */
        char *dk;                    /* the item the aggregate function is applied on */
        char *oeprname;              /* un-used */
        char *applname;              /* un-used */
        char *f_dk;                   /* the updated result representation */
        int index;                   /* Added for local rule for indexing these functions */
} tIagg;
```

Four functions which were developed for ROPE message generation are re-used and customized for building the above information for local-rule implementation. These functions are identify_query(), build_set(), complete_set() and build_Iagg().

But for the final factbase construction, we do not need to follow the same way as the ROPE message generation. For the local-rule implementation, the final factbase will only contain the necessary data items and eliminate all un-neccessary data items for the rule execution.

The aggregate functions in the rule may contain data items which belong to a specific independent query (or main views), and if these data items only exists in the condition part of the aggregate functions, we do not need to put them into the final factbase.

The way to build the final factbase for local-rule is as follows: (1) traverse the rule-tree to gathering data items which do not belong to any aggregate functions and mark the is_final to be TRUE in the data item's tItab structure. (2) for all the updated data items, we select the Primary Key of the updated data item into the final factbase. (3) From the established

Iagg list which stores all the aggregate functions of the rule, we determine one data item to be put into the final factbase for each aggregate functions.

During the above process, for each data items which will be in the final factbase, we also need to get the detailed information about these data items such as format and length from Metadatabase for later Pro*C code generation.

The following data structures are used for storing the final factbase information.

```
typedef struct ttFinal
{
  char *itemname;           /* the item name of the data item in the final factbase */
  char *iformat;             /* the data format of the data item in the final factbase */
  int  ilength;             /* the length of the data format */
  char *oeprname;            /* the oepr name of the data item */
}tFinal;

tyoedef struct ttFinalBase
{
  char *viewname;    /* the view name for the independent query (base view) */
  tFinal **itemlist;  /* all final factbase data items which belong to this independent query (or base view) */
}tBase;
```

The functions for constructing the above final factbase are stored in the factbase.c file and the major functions are build_final(), determine_final_items(), build_final_from_Itab(), build_final_from_Iagg(), Add_updated_oepr_key_into_final(), etc.

2.1.3. Generation of the MQL Queries

The Metadadabase-based Query Language (MQL, [Cheung 91]) is employed to represent the creation of factbase for the local rule. The advantage of using MQL for local-rule is (1) MQL is independent from local DBMS system query language; (2) there exists some modules which were developed for MQL query processing and Global Query System(GQS) but they can also be re-used by local-rule implementation (this will be explained later in 2.1.4).

The generation of MQL queries for local-rule follows the same procedures of ROPE message generation([Babin 92], page103). The difference is (1)the identifier for view name and aggregate functions are customized for local-rule implementation; (2) the creation of final distinct view (final factbase) for local-rule implementation is different and it does not contain any un-necessary data items.

The final MQL factbase generated by local-rule implementation for rule2a is as follows:

```
Distinct (
  From view rule2a_main get
    QUANTITY
    S_ADDR
    CUST_ID
  From view rule2a_sum0 get
    COST
  For
    DATE_SCHED of view rule2a_main =
    DATE_SCHED of view rule2a_sum0
  );
```

The gen_mql.c file contains the functions for generating the MQL queries.

2.1.4 Generation of the Local Queries

Current local-rule implementation is targeted to Oracle DBMS system and its methods and codes can be easily transported to other local DBMS system.

As we mentioned in 2.1.2, each independent query defined by the rule modeler contains data items which are directly or indirectly related to each other in the structural model (OER model) and it may involves multiple OE/PRs. When the rule modeler create the rule, he may not explicitly specify the join-conditions between the OE/PRs which are involved in the same independent query and also such specification would require detailed understanding of the OER model. Anyway, it is obviously not user-friendly to require the rule modeler to explicitly specify those join-conditions. In order to release such burden from the rule modeler, the local-rule implementation needs to generate those join-conditions which are not explicit defined in the rule. Fortunately such automatic join-condition determination algorithm has been developed for GQS ([Cheung 91], page 76) and we can try to re-use those modules for our local-rule implementation.

The basic steps for generating the local queries(or views) are as follows:

Step 1: Parsing the MQL file and store the information of the views being defined.
Step 2: Processing each view to find the physical data to retrieve (here we will use the join-condition determination algorithm to complete the conditions for each independent queries).

Step 3: Generate the query codes for the local system (Oracle SQL) to provide the final local factbase for rule execution.

The data structures defined for GQS and ROPE [Babin93] are re-used for our local-rule implementation and are listed as follows:

```
typedef struct ttsource {          /* where the data item come from */
    char *appl;                    /* application name */
    char *subj;                    /* subject name */
    char **oeprs;                  /* oepr name */
    char *view;                    /* view name */
    char *file;                    /* local file name */
    char **tables;                 /* local table list */
    int nb_oepr,nb_table;          /* number of oeprs and tables */
    } tsource;

typedef struct ttitem {            /* the detailed information of the data item */
    char *itemname;                /* item name */
    char *itemcode;                /* item code */
    char *iformat;                 /* item format */
    tsource *source;               /* the source where this data item belongs to */
    int selected;                  /* whether ths item is selected in the query */
    } titem;

typedef struct ttcond {     /* the condition component of the query */
    titem *item1;           /* the first data item */
    tptr value;             /* the result of the evaluation */
    titem *item2;             /* the second data item */
    char *operator;           /* the operator for the two data items */
    int valuetype;            /* the type of the value */
    int is_join;              /* check whether it belongs to join-condition */
    } tcond;

typedef struct ttselect {     /* the whole set of conditions for the where clause of the query */
    tcond **conds;            /* the list of the condition components */
    char **ops;               /* the list of the operations between components */
    int nb_cond;              /* the number of the condition components */
    } tselect;

typedef struct ttfunction {   /* the functions used in the query */
    char *display_fct;        /* the function like group by, order by , etc */
    char *value_fct;          /* the functions like sum, max, count, etc */
    char **items;             /* the list of data items in the function */
    int nb_items;             /* the number of items */
    } tfunction;

typedef struct ttview {       /* the detailed information of the view */
    char *viewname;           /* the view name */
    titem **items;            /* the set of items in the view */
    tselect *select;          /* the condition expr in WHERE clause for this view */
    char *view1;              /* the first sub-view for exists and count aggrgate function */
    char *view2;              /* the second sub-view for exists and count aggregate function */
    char **operands;          /* the set of operators work on which items */
    char *operator;              /* like UNION, EXCLUDE set opertator between views */
    tfunction *function;      /* record functions like SUM, EXIST, GROUPBY, etc */
    int nb_items,nb_op;    /* the number of items and operations */
    } tview;

typedef struct ttquery {       /* the detailed information of the query */
    char *viewname;           /* the view name for this query */
    char *appl;                   /* the application name */
    char *msg_id;              /* un-used for local-rule */
    char *reply_id;            /* un-used for local-rule */
    tselect *select;            /* the where clause of the query */
    titem **items;             /* the list of items in the query */
    int nb_items;               /* the number of items */
    } tquery;
```

2.1.5 Generation of the SQL Views

We should generate the SQL views for (1) independent queries (or base views, or main views); (2) aggregate functions views; (3) the final factbase view.

(1) For each independent queries, we will generate the SQL view as follows:

```
            CREATE VIEW alias name (<item1>, ...., <itemN>)
                AS SELECT <loc_table_i1.item1>, ..., <local_table_im.itemN>
                FROM        <loca_table_i1>, ..., <loc_table_im>
                WHERE <join-condition 1>
```

```
                    AND <join-condition 2>....
          .                  ......
                   AND <join-condition k>
```

The local table name can be determined by the oepr name and the application name and this information has already been stored in the Metadatabase. The join-conditions for the independent query have been determined in 2.1.4. Let's see what the local independent query looks like for the example rule2a:

```
create view rule2a_main
        (COST, QUANTITY, DATE_SCHED, CUST_ID, CUST_NAME, S_ADDR)
        as select OPS_ITEM.COST,
                OPS_ORDER_ITEM.QUANTITY,
                OPS_ORDER_ITEM.DATE_SCHED,
                OPS_CUSTOMER.CUST_ID,
                OPS_CUSTOMER.CUST_NAME,
                OPS_CUSTOMER.S_ADDR
        from  OPS_ITEM, OPS_ORDER_ITEM, OPS_ORDER, OPS_CUSTOMER
        where  OPS_ORDER.CUST_ORDER_ID =
OPS_ORDER_ITEM.CUST_ORDER_ID
         AND OPS_CUSTOMER.CUST_ID = OPS_ORDER.CUST_ID
         AND OPS_ITEM.PART_ID = OPS_ORDER_ITEM.PART_ID ;
```

We can see the join-conditions in the WHERE clause which is generated by the join-condition determination algorithm.

(2) For aggregate functions like SUM, MAX, MIN, AVG

Let's use AVG as a example, for avg(d, condition, g), where d is the data item the aggregate function is applied on, g is the data item the aggregate function is grouped by.

```
CREATE VIEW viewname ( avg_d, g)
        AS SELECT  avg(d), g
        FROM < view name of the independent query this AVG belong to >
        WHERE <condition>
        GROUP BY g;
```

(3) For aggregate function like EXISTS and COUNT

Let’s suppose we have the following aggregate function in the rule:
EXISTS(<condition>, b.g), assuming the <condition> involves two independent queries a and b. The following SQL views will be generated:

```
CREATE VIEW existK_b (exist_result, g)
        AS SELECT 1, g
           FROM b, a
          WHERE <condition>
          GROUP BY b.g;

CREATE VIEW existK_a (exist_result, g)
        AS SELECT 0, g
           FROM b, a
          WHERE NOT (<condition>)
           GROUP BY b.g;

CREATE VIEW existK
        AS SELECT *
            FROM existK_a
        UNION
```

```
                         SELECT *
                        FROM existK_b;
```

For the COUNT aggregate function, there is some difference. Let’s suppose we have the following function in the rule:

COUNT(<condition>, b.g), assuming the <condition> involves two independent queries a and b. The semantic meaning of it is (1) join the view a and view b through the <condition>; (2) for the new view, count how many tuples in each set grouped by b.g;

```
CREATE VIEW countK_b (count_result, g)
        AS SELECT count(*), g
           FROM b, a
           WHERE <condition>
           GROUP BY b.g;

CREATE VIEW countK_a (count_result, g)
        AS SELECT 0, g
           FROM b, a
           WHERE NOT (<condition>)
           GROUP BY b.g;

CREATE VIEW countK (count_result, g)
        AS SELECT *
           FROM countK_a
        UNION
          SELECT *
          FROM countK_b;
```

(4) Generate the final factbase

Let’s assume the rule contains the following independent queries such as alias1, ..., aliasK and contains the following aggregate functions as agg1, ..., aggJ. Also we have determined all the data items which will be in the final factbase at 2.1.2, let's say these final data items are item1,..., item M.

```
CREATE VIEW factbase ( <item 1>, ..., <item M> )
          AS SELECT <item 1>, ...., <item M>
              FROM  alias1, ..., aliasK, agg1, ..., agg J
             WHERE <join-conditions through group by items in the aggregate functions>
```

2.1.6 Generation of the Pro*C Code

Step 1: Get the detailed information about user-defined routines in the rule from the Metadatbase such as location and include the file where the routines located at the beginning of the Pro*C code.

For example, if the rule used a user-defined routine notify_operator() which is located at /u/mdb/unix/ops/opsutil.c, then the Pro*C code should contains the following:

```
#include "/u/mdb/unix/ops/opsutil.c"
```

Step 2: Generate the host variable declaration section from the final factbase which has being built at step II.1.2. At this point, we need the detailed information such as data format and length for each data items which is in the final factbase.

For example, suppose we have three data items in the final factbase such as **od_status**, **cust_order_id**, **quantity**, and the result of the aggregate function such as the sum of the **cost**, then this declaration section generated will be as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
        int             val_QUANTITY;
        VARCHAR  val_S_ADDR[41];
        VARCHAR  val_CUST_ID[6];
        float   val_sum0_COST;
EXEC SQL END DECLARE SECTION;
```

Step 3: Generate the cursor declaration for fetching the data from the factbase.

For the rule2a example, the cursor declaration is generated as the following:

```
EXEC SQL DECLARE factbase CURSOR FOR

        SELECT DISTINCT
                QUANTITY,
                S_ADDR,
                CUST_ID,
                sum0_COST
        FROM rule2a_factbase;

EXEC SQL OPEN factbase;
```

Step 4: For each tuple in the factbase, evaluating the condition and executing the actions, for the updated data items, generate the Pro*C update statement.

The rula2a example:

```
for ( ; ; )
    {
    EXEC SQL FETCH factbase INTO            /*  fetch data from factbase, one tuple each time */
                :val_QUANTITY,
                :val_S_ADDR,
                :val_CUST_ID,
                :val_sum0_COST;

    val_S_ADDR.arr[val_S_ADDR.len] = '\0';
    val_CUST_ID.arr[val_CUST_ID.len] = '\0';

    if ((val_QUANTITY  >  10 ) && ( val_sum0_COST >  100 ))         /* condition evaluation */
    {
        strcpy(val_S_ADDR.arr, "send notify");
        val_S_ADDR.len = strlen("send notify");

        notify_operator();                  /* call user-defined routine */

        EXEC SQL UPDATE OPS_CUSTOMER                /* update local table */
                SET S_ADDR = :val_S_ADDR
```

```
                        WHERE CUST_ID = :val_CUST_ID;
        }

        }
```

2.1.7 Local-Rule Trigger

Current local-rule implementation can support the following types of trigger events:

(1)time trigger

when_time_is(), a specific time has been reached.

every_time(),   after a specified time interval

(2)program trigger

Application programmer can directly call the generated Pro*C code from his program to execute the specified rule.

(3)rule trigger (or rule chaining)

Rule A can trigger rule B by putting the rule identifier of rule B in its action list just like calling the user-defined routines, so the rule B will be fired following the execution of rule A.

(4)user trigger

User can execute the rule at any time because the executable code has been generated.

Current implementation does not support the data trigger which will fire the rule when some database content has been modified (database monitor). This kind of database monitoring trigger can be implemented with the facilities provided by Oracle Version 7 and it will be discussed in Part 3 (current local-rule system based on Oracle Version 6).

A local timer system has been designed for the time-triggered local-rule. From II.1.1 we know the trigger information is extracted from the rule text by the parser and store in a trig_mes.doc file. The time-trigger information is stored as :

<ruleid> is triggered by time using "<time>"

```
typedef struct ttrig
{
   char *ruleid;                /* the rule identifier */
   char *time;                  /* the <time> information from the trig_mes.doc file */
   int  delta[5];               /* for non-absolute time trigger */
   int  triggered;              /* there may be many points it be triggered,but only once */
   int  absolute;               /* check whether it is when_time_is or every_time */
   int  year, month, day, hour, minute;        /* next event time */
}ttrig;
```

The timer first parse the information stored in trig_mes.doc and store the time-trigger information in the ttrig structure. Then at a pre-defined time interval,it checks the ttrig list to see which rule's event time equal to the current local time, if it match, then execute the rule; for the non-absolute time-triggered rule, it will update its next event time.

2.2 Global-Rule Decomposition

The global-rule contains data items and user-defined routines which are distributed in different application systems, so the execution of the global-rule will be more complex than the execution of the local-rule. We need to serialize the execution of the user-defined routines and update directives in different systems the appropriate time. Algorithms have been developed by Babin Gilbert([Babin 92], page 105 - page 126) in his Ph.D theses and those algorithms are implemented in this project.

The major steps for global-rule decomposition are as follows:

Step 1: rearrange the condition and actions to obtain a set of condition and actions which are processed in a single system.

Step 2: Reorder the top-level operations of the above rearranged rule-tree to obtain an optimal partitioning of user-defined routines and update directives.

The step 1 is implemented in arrange.c file which contains the following major functions:

preorder_determine_SysO (tnode *O_node)
Traverse the rule-tree in pre-order to determine the set of application systems in which the operation node O_node(a sub-tree, including all its child nodes) to be executed.

reverse_order_adjust_AppO (tnode *O_node)
For each operation nodes, adjust the system where the node is executed in a reverse order.

serialize(tnode *root, tnode *O_node, int *O_pos)

It contains two major function which are serialize_assign() and serialize_operation(). They are used for serialize the direct child node of the root node to guarantee each direct child node under the root node will be executed in the same application system.

generate_matrix_of_precedence(tnode **list, int n)

Determine the strict precedence relationships among the direct child nodes of the root in the rule-tree and the result is stored in node_matrix[][] for later portioning process.

The step 2 is implemented in part.c and contains the following major functions:

Generate_initial_partitions(F,root,node_num)

Generate the initial partitions (assign each node under root into a partition).

Sort_partition(tpart **Flist)

Place the partitions in a linear sequence and maintain the original precedence relationship.

Optimize_partition(tpart **Flist)

Try to reduce the total number of partitions to get the optimal partitions by merging adjacent partitions executed in the same application system.

Generate_ordered_list_of_operation(tnode *root, int **nodelist)

For the partition F containing operations O1,..,On, generate the ordered list of this operations and maintain its original precedence constraints.

We use Matrix[][] to store the precedence of partitions and use matrix[][] to store the precedence of the operations in each partition.

The following data structure is used for storing the information about a partition:

```
typedef struct ttpart
        {
        int** ids;              /* the list of ids of the operation nodes in this partition */
       char* apname;          /* the application system this partion belongs to */
        int index;              /* the index of this partition in the precedence matrix */
        } tpart;
```

The arrange.c and part.c have been incorporated into the ROPE message generation for generating messages to each local application system shell for further processing to realize the final execution of the global-rule. Each partition is used for generating a subrule in the ROPE message generation.

Part 3
Limitations and Suggestions

3.1 Rule Triggers

Current local-rule implementation can support the following event triggers, they are (1)time-triggered, the rule can be triggered at a specific time or at a specified time interval; (2)user-triggered: user can executed any rules which are stored in the rulebase at any time; (3)program-triggered: application programmer can directly call the Pro*C function defined for a rule in his application program; (4))rule-triggered: rule A can put rule B in its action list and trigger it just like calling a user-defined routine. It can also use the combination of time-trigger and some conditions such as checking the SUM on a data item or COUNT on some kinds of tuples or checking whether EXISTS a specific tuple to monitor the changes to a table, but this methods have its limitation. So, basically, current local-rule implementation can not fully support data-trigger to monitor the table changes such as INSERT, DELETE, UPDATE, etc.

Oracle 7 provide us some possibilities to implement the data monitoring trigger which is defined in current TSER rule grammar for monitoring the OE/PR changes. The syntax of the data monitoring trigger is defined as following [Babin93]:

changes_in_oepr (application, oe/pr, detect_update, detect_delete, detect_insert, time_patter, month, day, year, hour, minute)

Oracle 7 provide facilities for database triggers and allows user to define procedures that can be implicitly executed when an INSERT, UPDATE, or DELETE statement is issued against the associated table ([Oracle92]). Oracle 7 treats the triggers as stored procedures and the triggers are stored in the database which are separated from their associated tables.

The Oracle 7 trigger has three basic parts which are (1) a triggering event or statement; (2) a trigger restriction; (3) a trigger action. So, basically, mapping our data-trigger defined in [Babin92] into Oracle 7's database trigger should be straight forward. The information in change_in_oepr() can be mapped into the Oracle trigger statement, the rule condition can be mapped into the Oracle trigger restriction, and the rule action can be mapped into Oracle trigger action.

In addition, our current rule system only support single trigger, it is suggested to extend the syntax to support multiple triggers. In order to fully take advantage of the Oracle 7, the syntax of change_in_oepr() should also be extended to include more things such as data items for monitoring the changes on a specific data item in the table.

3.2 Local-Rule Implementation

Our current system can generate the Oracle SQL view for creating the factbase and Oracle Pro*C code for the rule execution. In the future, it is suggested to extend the current GIRD schema or the rulebase model to store the meta-local-rule information in the rulebase such as the view names and Pro*C routine name related to specific local-rule. This will be helpful for the future rule management such as delete, insert and update the local-rulebase.

As we mentioned in 2.1.4, the local-rule implementation need to re-use the join-condition determination algorithm to complete the join-conditions for the independent query of the rule. This algorithm have to search the whole OER model to find the shortest path for linking the related OE/PRs which are involved in this independent query. It may be possible to reduce the search space if this algorithm can take advantage of the knowledge of which SUBJECT or CONTEXT this rule belongs to, so we just need to search the specific set of OE/PRs instead of the whole OE/PRs.

The rule text processed by current local-rule implementation is retrieved from Metadatabase, so when we parse this rule text to build the rule tree, we do not need to strictly check its semantic correctness such as the data type matching for assignment, etc. This is because we have already checked them when we populate the rule into the Metadatabase. If we want to process the rule text directly from IBMS, we need to add the semantic checking modules into the current rule parser. In addition, the rule processing needs the OER information so it's impossible to process the rule at the SER level for the local-rule implementation.

3.3 Global-Rule Decomposition

Current ROPE shell technology and ROPE message generation only support the serialized execution of the global rule. The global-rule is decomposed into sub-rules where each sub-rule is related to a partition generated by the decomposition algorithm in 2.2. For any partitions (or sub-rule), there exists the precedence relationship which is either inherited from the original rule or enforced by the algorithm. It's suggested to fully exploit the potential parallelism among the operation nodes of the global-rule tree in the future.

Part 4
Test and Sample Results

4.1 System Environment

The system environment this project is developed and tested is as follows:

(1) Oracle DBMS version 6 on a IBM RS6000 workstation (CHIP, located at CII 7129).

(2) Metadatabase System (on CHIP)

(3) Order Processing System (on CHIP)

We use the Order Processing System to test the local-rule implementation and use the global-rules defined in the Metadatabase to test the global-rule decomposition.

4.2 User Manual

4.2.1 Local-Rule Implementation

User can following the procedure described below to play with the local-rule system.

(1) Create your local-rule text and store them in the ruletext.doc file. In ruletext.doc the format is one rule per line as follows:

```
(rule_id) (IF (<trigger> AND <condition>) THEN action;...;action;
```

(2) Type "rule" to run the program for generating the SQL view and the Pro*C code.

(3) For each rule with rule id of rulexxx, three files will be generated for each rule and they are rulexxx.mql, which stores MQL views; rulexxx.sql, which stores SQL view; and rulexxx.pc which stores the Pro*C code.

Example: suppose you create two rules in the ruletext.doc with rule id to rule007 and rule101, then the following files will be generated:

```
rule007.sql  /* store the SQL views for factbase of rule007 */
rule007.pc   /* store the Pro*C code for rule007 */
rule101.sql  /* store the SQL views for factbase of rule101 */
rule101.pc   /* store the Pro*C code for rule101 */
```

(4) Compile all your *.pc files being generated for your rules.

(5) For those time-triggered rules, just type "timer" to start the timer system to check and trigger those rules.

(6) You can also execute the rules by typing its rule id and call the generated Pro*C function for the rule in your own application program. If you create new rules, you can also put the Pro*C function in the action list of your rule to realize the rule chaining.

4.2.2 Global-Rule Decomposition

The global-rule decomposition is actually incorporated into the ROPE message generation, so what you are using is actually the ROPE message generation.

(1) You can type "get_rule" to retrieve all the global-rules from Metadatabase and these rules are stored in ruletext.doc; You can also create your own global-rules but the data items and routines used in your rules should have already existed in the Metadatabase.

(2) Type "rparser" to run the ROPE message generation program (the global-rule decomposition has been incorporated into this program) and for each rule, it will generate one message file for each application system and you can see all the intermediate results of the decomposition process such as the rearranged rule-tree, the precedence matrix for the partitions being created.

4.3 Sample Results

4.3.1 Local-Rule Implementation

The following is one of the example rules for testing the local-rule implementation, and this rule should first be stored in the ruletext.doc file.

```
(rule101)
          (IF every_time(-1,-1,-1,0,1)
          AND (sum(ORDER_PROCESSING.QUANTITY@ORDER_ITEM,
                ORDER_PROCESSING.CUST_NAME@CUSTOMER = "JINSONG",
                ORDER_PROCESSING.DATE_SCHED@ORDER_ITEM) > 40)
          THEN ORDER_PROCESSING.OD_STATUS@ORDER := "Delayed";
                notify_operator(“a”);)
```

After running the local-rule implementation program, the following files will be generated:

1. trig_mes.doc	This file store the trigger information.

2. rule101.mql	This file store the MQL queries
3. rule101.sql	This file stores the SQL views for factbase.
4. rule101.pc	This file stores the Pro*C code for the rule.

The following lists the contents of each files.

(1) The content of trig_mes.doc is as follows:

```
rule101 is triggered by time using "+?/?/? 0:01"
```

(2) The content of rule101.mql is as follows:

```
Define view rule101_main
      From OE/PR ORDER_ITEM get
            QUANTITY
            DATE_SCHED
      From OE/PR CUSTOMER get
            CUST_NAME
      From OE/PR ORDER get
            OD_STATUS
            CUST_ORDER_ID;

Define view rule101_sum0
   sum ( From view rule101_main get
      QUANTITY
     DATE_SCHED
   For CUST_NAME of view rule101_main="JINSONG" );

Distinct (
   From view rule101_main get
      OD_STATUS
      CUST_ORDER_ID
   From view rule101_sum0 get
      QUANTITY
   For
      DATE_SCHED of view rule101_main =
      DATE_SCHED of view rule101_sum0
   );
```

(3) The content of rule101.sql is as follows:

```
create view rule101_main
        (QUANTITY, DATE_SCHED, CUST_ORDER_ID, OD_STATUS, CUST_NAME)
        as select OPS_ORDER_ITEM.QUANTITY,
                  OPS_ORDER_ITEM.DATE_SCHED,
                  OPS_ORDER.CUST_ORDER_ID,
                  OPS_ORDER.OD_STATUS,
                  OPS_CUSTOMER.CUST_NAME
        from  OPS_ORDER_ITEM, OPS_ORDER, OPS_CUSTOMER
        where  OPS_ORDER.CUST_ORDER_ID =  OPS_ORDER_ITEM.CUST_ORDER_ID
         AND OPS_CUSTOMER.CUST_ID = OPS_ORDER.CUST_ID ;

create view rule101_sum0
        ( sum_QUANTITY, DATE_SCHED )
        as select sum(QUANTITY), DATE_SCHED
           from rule101_main
           where  rule101_main.CUST_NAME = 'JINSONG'
           group by rule101_main.DATE_SCHED;

create view rule101_factbase
        ( OD_STATUS, CUST_ORDER_ID, sum0_QUANTITY )
        as select
                  rule101_main.OD_STATUS,
                  rule101_main.CUST_ORDER_ID,
                  rule101_sum0.sum_QUANTITY
           from  rule101_main, rule101_sum0
           where rule101_main.DATE_SCHED = rule101_sum0.DATE_SCHED;
```

(4) The contents of rule101.pc is as follows:

```
#include <stdio.h>

#include "/u/mdb/unix/ops/ops_util.c"

EXEC SQL INCLUDE sqlca;

EXEC SQL BEGIN DECLARE SECTION;
   VARCHAR username[20];
   VARCHAR password[20];
EXEC SQL END DECLARE SECTION;

main()
{

   strcpy(username.arr, "ops");
   username.len = strlen(username.arr);
   strcpy(password.arr, "ops");
   password.len = strlen(password.arr);
   EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
   EXEC SQL CONNECT :username IDENTIFIED BY :password;

   rule101();
```

```
    return;
sqlerror:
    printf("error in rule execution of rule101\n");
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK;
    exit(1);
}

rule101()
{
        EXEC SQL BEGIN DECLARE SECTION;
                VARCHAR  val_OD_STATUS[11];
                VARCHAR  val_CUST_ORDER_ID[10];
                int      val_sum0_QUANTITY;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
        EXEC SQL DECLARE factbase CURSOR FOR

                SELECT DISTINCT
                         OD_STATUS,
                         CUST_ORDER_ID,
                         sum0_QUANTITY
                FROM rule101_factbase;

        EXEC SQL OPEN factbase;

        EXEC SQL WHENEVER NOT FOUND GOTO end_of_fetch;

        for ( ; ; )
        {
        EXEC SQL FETCH factbase INTO
                         :val_OD_STATUS,
                         :val_CUST_ORDER_ID,
                         :val_sum0_QUANTITY;

        val_OD_STATUS.arr[val_OD_STATUS.len] = '\0';
        val_CUST_ORDER_ID.arr[val_CUST_ORDER_ID.len] = '\0';

        if ( val_sum0_QUANTITY >  40 )
        {
               strcpy(val_OD_STATUS.arr, "Delayed");
               val_OD_STATUS.len = strlen("Delayed");

               notify_operator( "a" );

               EXEC SQL UPDATE OPS_ORDER
                       SET OD_STATUS = :val_OD_STATUS
                       WHERE CUST_ORDER_ID = :val_CUST_ORDER_ID;
        }

        }
end_of_fetch:
        EXEC SQL CLOSE factbase;
        EXEC SQL COMMIT WORK RELEASE;
        return;

sqlerror:
        EXEC SQL WHENEVER SQLERROR CONTINUE;
        EXEC SQL ROLLBACK WORK;
        exit(1);
}
```

After compiling the above rule101.pc file, we can directly run this Pro*C program for the rule101. Let's look at the status of Order Processing System before the execution of this rule.

```
SQL>  select quantity, part_id, date_sched, cust_name, od_status, order_line_id
  2  from ops_order_item oi, ops_order o, ops_customer c
  3  where oi.cust_order_id = o.cust_order_id
  4  and o.cust_id = c.cust_id
  5  and cust_name = 'JINSONG';
```

The output is as follows:

```
QUANTITY  PART_ ID   DATE_SCHED  CUST_NAME     OD_STATUS  ORDER_LINE_ID
---------- ----- -------- ----------------------------- ---------- ------------- -------------------- ------------------ ----------------
5                    A        Monday          JINSONG          ON TIME         00011_020
10                   Z        Monday          JINSONG          ON TIME         00011_021
5                    PZ200    Monday          JINSONG          ON TIME         00011_022
14                   PZ201    Monday          JINSONG          ON TIME         00011_023
```

At this point, if we execute the rule101, it will not do any actions because the SUM(quantity) = 29 < 40 and the rule condition evaluation is FALSE.

Now, let's update the quantity to satisfy the rule condition:

```
SQL> update ops_order_item
  2  set quantity = 50
  3  where order_line_id = '00011_020';

1 row updated.
SQL> commit;
Commit complete.
```

Now, let's run the rule101 again, this time the rule condition is evaluted to be TRUE and some actions should happen, let's see:

```
SQL> select quantity, part_id, date_sched, cust_name, od_status, order_line_id
  2  from ops_order_item oi, ops_order o, ops_customer c
  3  where oi.cust_order_id = o.cust_order_id
  4  and o.cust_id = c.cust_id
  5  and cust_name = 'JINSONG';

QUANTITY  PART_ID  DATE_SCHED  CUST_NAME  OD_STATUS  ORDER_LINE_ID
---------- ----- -------- ----------------------------- ---------- ---------- ----------------- ----------------- -----------------
50          A       Monday      JINSONG    Delayed     00011_020
10          Z       Monday      JINSONG    Delayed     00011_021
5           PZ200   Monday      JINSONG    Delayed     00011_022
14          PZ201   Monday      JINSONG    Delayed     00011_023

SQL>
```

Now, we can see the OD_STATUS has been changed after the rule101 is executed.

Next, we will use the local timer system for testing the above rule. The above rule is triggered by time using "+?/?/?/ 0:01", which means this rule should be triggered every one minute. Let's see:

```
/u/mdb/unix/timer[23] timer
+?/?/? 0:01

6/25/1994 16:38:21

...... Checking who should be triggered ......


6/25/1994 16:38:31

...... Checking who should be triggered ......


6/25/1994 16:38:41

...... Checking who should be triggered ......


6/25/1994 16:38:51

...... Checking who should be triggered ......
```

```
6/25/1994 16:39:1


 The **** rule101 **** is triggered!

Nothing happens if the condition is false, otherwise ...

... rule101: condition is TRUE, operator is notified...
... rule101: condition is TRUE, operator is notified...
... rule101: condition is TRUE, operator is notified...
... rule101: condition is TRUE, operator is notified...
... rule101: condition is TRUE, operator is notified...
... rule101: condition is TRUE, operator is notified...
... rule101: condition is TRUE, operator is notified...
... rule101: condition is TRUE, operator is notified...

...... Checking who should be triggered ......


6/25/1994 16:39:12

...... Checking who should be triggered ......


6/25/1994 16:39:22
^C/u/mdb/unix/timer[24]
```

The message of "... rule101: condition is TRUE, operator is notified..." is generated by the user-defined routine in the rule101 which is notify_operator().Because the final factbase of the rule101 contains 8 tuples, so this routine is called eight times.

4.3.2 Global-Rule Decomposition

The testing global-rule which is stored in the ruletext.doc is as follows:

```
(PREPARE_NEW_PLAN_R1)

(IF every_time(-1,-1,-1,1,0)
    AND (A.OI_STATUS = "DESIGNED")
    AND NOT exist(A.ORDER_PROCESSING.PART_ID@(ORDER_ITEM)
                       = B.PARTID@(PARTREV),
                       A.ORDER_LINE_ID)
THEN
          B.PARTREV@(PARTREV) :=
                   request_design_revision(A.ORDER_PROCESSING.PART_ID@(ORDER_ITEM));
         A.OI_STATUS := "IN ENG";
          B.ROUTING := 0;
          B.PLANSTATUS := "WORKING";)
```

The output related to the global-rule decomposition is listed as follows:

```
BEGIN SERIALIZE AND PARTITION!

THE ORIGINAL RULE:
 ntype = |RULE|
 valuetype = |RULE|
 nvalue = |PREPARE_NEW_PLAN_R1|
 nb_children = |7|

   ntype = |IF|
   valuetype = |IF|
   nvalue = |IF|
   nb_children = |1|

     ntype = |OPERATOR|
     valuetype = |AND|
     nvalue = |AND|
     nb_children = |2|

       ntype = |OPERATOR|
       valuetype = |EQ|

       nvalue = |=|
       nb_children = |2|

         ntype = |ITEM|
         valuetype = |IDENT|
         nvalue = |OI_STATUS|
          alias = |A|
         oepr = |ORDER_ITEM|

         ntype = |CONSTANT|
         valuetype = |STRING|
         nvalue = |DESIGNED|

       ntype = |OPERATOR|
       valuetype = |NOT|
       nvalue = |NOT|
       nb_children = |1|

         ntype = |FN_NAME|
         valuetype = |IDENT|
         nvalue = |exist|

         nb_children = |2|

           ntype = |OPERATOR|
           valuetype = |EQ|
            nvalue = |=|
            nb_children = |2|

               ntype = |ITEM|
               valuetype = |IDENT|
               nvalue = |PART_ID|
               alias = |A|
               oepr = |ORDER_ITEM|

               ntype = |ITEM|
               valuetype = |IDENT|
               nvalue = |PARTID|
               alias = |B|
               oepr = |PARTREV|

            ntype = |ITEM|
            valuetype = |IDENT|
            nvalue = |ORDER_LINE_ID|

            alias = |A|
            oepr = |ORDER_ITEM|

ntype = |OPERATOR|
valuetype = |ASSIGN|
nvalue = |:=|
nb_children = |2|

   ntype = |ITEM|
   valuetype = |IDENT|
   nvalue = |PARTREV|
   alias = |B|
   oepr = |PARTREV|

   ntype = |FN_NAME|
   valuetype = |IDENT|
   nvalue = |request_design_revision|
   executed_in = |PROCESS_PLAN|
   nb_children = |1|

      ntype = |ITEM|
      valuetype = |IDENT|
      nvalue = |PART_ID|
      alias = |A|
      oepr = |ORDER_ITEM|

ntype = |OPERATOR|
valuetype = |ASSIGN|
nvalue = |:=|
nb_children = |2|

   ntype = |ITEM|
   valuetype = |IDENT|
   nvalue = |OI_STATUS|
   alias = |A|
   oepr = |ORDER_ITEM|
      ntype = |CONSTANT|
      valuetype = |STRING|
      nvalue = |IN ENG|

   ntype = |OPERATOR|
   valuetype = |ASSIGN|
   nvalue = |:=|
   nb_children = |2|

      ntype = |ITEM|
      valuetype = |IDENT|
      nvalue = |ROUTING|
      alias = |B|
      oepr = |PLAN|

      ntype = |CONSTANT|
      valuetype = |INTEGER|
      nvalue = ||

   ntype = |OPERATOR|
   valuetype = |ASSIGN|
   nvalue = |:=|
   nb_children = |2|

      ntype = |ITEM|
      valuetype = |IDENT|
      nvalue = |PLANSTATUS|
      alias = |B|
      oepr = |PLAN|

      ntype = |CONSTANT|
      valuetype = |STRING|
      nvalue = |WORKING|

   ntype = |UPDATE_DIRECTIVE|
   valuetype = ||
   nvalue = ||
   executed_in = |PROCESS_PLAN|

   ntype = |UPDATE_DIRECTIVE|
   valuetype = ||
   nvalue = ||
   executed_in = |ORDER_PROCESSING|
```

```
PREORDER DETERMINE SYSO:
 ntype = |RULE|
 valuetype = |RULE|
 nvalue = |PREPARE_NEW_PLAN_R1|
 sysnames[0] = |PROCESS_PLAN|
 sysnames[1] = |ORDER_PROCESSING|
 nb_children = |7|
ntype = |IF|
valuetype = |IF|
nvalue = |IF|
nb_children = |1|

  ntype = |OPERATOR|
  valuetype = |AND|
  nvalue = |AND|
  nb_children = |2|

    ntype = |OPERATOR|
    valuetype = |EQ|
    nvalue = |=|
    nb_children = |2|


      ntype = |ITEM|
      valuetype = |IDENT|
      nvalue = |OI_STATUS|
      alias = |A|
      oepr = |ORDER_ITEM|

      ntype = |CONSTANT|
      valuetype = |STRING|
      nvalue = |DESIGNED|

    ntype = |OPERATOR|
    valuetype = |NOT|
    nvalue = |NOT|
    nb_children = |1|

      ntype = |FN_NAME|
      valuetype = |IDENT|
      nvalue = |exist|
      nb_children = |2|

        ntype = |OPERATOR|
        valuetype = |EQ|
        nvalue = |=|
        nb_children = |2|

          ntype = |ITEM|
          valuetype = |IDENT|
          nvalue = |PART_ID|
          alias = |A|
          oepr = |ORDER_ITEM|

          ntype = |ITEM|
          valuetype = |IDENT|
          nvalue = |PARTID|
          alias = |B|
          oepr = |PARTREV|

        ntype = |ITEM|
            valuetype = |IDENT|
            nvalue = |ORDER_LINE_ID|
            alias = |A|
            oepr = |ORDER_ITEM|

ntype = |OPERATOR|
valuetype = |ASSIGN|
nvalue = |:=|
sysnames[0] = |PROCESS_PLAN|
nb_children = |2|

   ntype = |ITEM|
   valuetype = |IDENT|
   nvalue = |PARTREV|
   alias = |B|
   oepr = |PARTREV|

   ntype = |FN_NAME|
   valuetype = |IDENT|
   nvalue = |request_design_revision|
   executed_in = |PROCESS_PLAN|
   sysnames[0] = |PROCESS_PLAN|
   nb_children = |1|

      ntype = |ITEM|
      valuetype = |IDENT|
      nvalue = |PART_ID|
      alias = |A|
      oepr = |ORDER_ITEM|

ntype = |OPERATOR|
valuetype = |ASSIGN|
nvalue = |:=|
nb_children = |2|

   ntype = |ITEM|
   valuetype = |IDENT|
   nvalue = |OI_STATUS|
   alias = |A|
   oepr = |ORDER_ITEM|

   ntype = |CONSTANT|
   valuetype = |STRING|
   nvalue = |IN ENG|

ntype = |OPERATOR|
valuetype = |ASSIGN|
nvalue = |:=|
nb_children = |2|

   ntype = |ITEM|
   valuetype = |IDENT|
   nvalue = |ROUTING|
   alias = |B|
   oepr = |PLAN|
      ntype = |CONSTANT|
      valuetype = |INTEGER|
      nvalue = ||

    ntype = |OPERATOR|
    valuetype = |ASSIGN|
    nvalue = |:=|
    nb_children = |2|

      ntype = |ITEM|
      valuetype = |IDENT|
      nvalue = |PLANSTATUS|
      alias = |B|
      oepr = |PLAN|

      ntype = |CONSTANT|
      valuetype = |STRING|
      nvalue = |WORKING|

    ntype = |UPDATE_DIRECTIVE|
    valuetype = ||
    nvalue = ||
    executed_in = |PROCESS_PLAN|
    sysnames[0] = |PROCESS_PLAN|

    ntype = |UPDATE_DIRECTIVE|
    valuetype = ||
    nvalue = ||
    executed_in = |ORDER_PROCESSING|
    sysnames[0] = |ORDER_PROCESSING|

REVERSE ADJUST APPLO TREE!!
 ntype = |RULE|
 valuetype = |RULE|
 nvalue = |PREPARE_NEW_PLAN_R1|
 sysnames[0] = |PROCESS_PLAN|
 sysnames[1] = |ORDER_PROCESSING|
 nb_children = |7|

    ntype = |IF|
    valuetype = |IF|
    nvalue = |IF|
    nb_children = |1|

      ntype = |OPERATOR|
      valuetype = |AND|
      nvalue = |AND|
      nb_children = |2|

        ntype = |OPERATOR|
        valuetype = |EQ|
        nvalue = |=|
        nb_children = |2|
            ntype = |ITEM|
            valuetype = |IDENT|
            nvalue = |OI_STATUS|
            alias = |A|
            oepr = |ORDER_ITEM|

            ntype = |CONSTANT|
            valuetype = |STRING|
            nvalue = |DESIGNED|

         ntype = |OPERATOR|
         valuetype = |NOT|
         nvalue = |NOT|
         nb_children = |1|

            ntype = |FN_NAME|
            valuetype = |IDENT|
            nvalue = |exist|
            nb_children = |2|

               ntype = |OPERATOR|
               valuetype = |EQ|
               nvalue = |=|
               nb_children = |2|

                  ntype = |ITEM|
                  valuetype = |IDENT|
                  nvalue = |PART_ID|
                  alias = |A|
                  oepr = |ORDER_ITEM|

                  ntype = |ITEM|
                  valuetype = |IDENT|
                  nvalue = |PARTID|
                  alias = |B|
                  oepr = |PARTREV|

               ntype = |ITEM|
               valuetype = |IDENT|
               nvalue = |ORDER_LINE_ID|
               alias = |A|
               oepr = |ORDER_ITEM|

   ntype = |OPERATOR|
   valuetype = |ASSIGN|
   nvalue = |:=|
   executed_in = |PROCESS_PLAN|
   sysnames[0] = |PROCESS_PLAN|
   nb_children = |2|

      ntype = |ITEM|
      valuetype = |IDENT|
      nvalue = |PARTREV|
      alias = |B|
      oepr = |PARTREV|
      ntype = |FN_NAME|
      valuetype = |IDENT|
      nvalue = |request_design_revision|
      executed_in = |PROCESS_PLAN|
      sysnames[0] = |PROCESS_PLAN|
      nb_children = |1|

         ntype = |ITEM|
         valuetype = |IDENT|
         nvalue = |PART_ID|
         alias = |A|
         oepr = |ORDER_ITEM|

   ntype = |OPERATOR|
   valuetype = |ASSIGN|
   nvalue = |:=|
   nb_children = |2|

      ntype = |ITEM|
      valuetype = |IDENT|
      nvalue = |OI_STATUS|
      alias = |A|
      oepr = |ORDER_ITEM|

      ntype = |CONSTANT|
      valuetype = |STRING|
      nvalue = |IN ENG|

   ntype = |OPERATOR|
   valuetype = |ASSIGN|
   nvalue = |:=|
   nb_children = |2|

      ntype = |ITEM|
      valuetype = |IDENT|
      nvalue = |ROUTING|
      alias = |B|
      oepr = |PLAN|

      ntype = |CONSTANT|
      valuetype = |INTEGER|
      nvalue = ||

   ntype = |OPERATOR|
   valuetype = |ASSIGN|
   nvalue = |:=|
   nb_children = |2|

      ntype = |ITEM|
      valuetype = |IDENT|
      nvalue = |PLANSTATUS|
      alias = |B|
      oepr = |PLAN|
         ntype = |CONSTANT|
         valuetype = |STRING|
         nvalue = |WORKING|

      ntype = |UPDATE_DIRECTIVE|
      valuetype = ||
      nvalue = ||
      executed_in = |PROCESS_PLAN|
      sysnames[0] = |PROCESS_PLAN|

      ntype = |UPDATE_DIRECTIVE|
      valuetype = ||
      nvalue = ||
      executed_in = |ORDER_PROCESSING|
      sysnames[0] = |ORDER_PROCESSING|



   LAST REARRANGED TREE!!
    ntype = |RULE|
    valuetype = |RULE|
    nvalue = |PREPARE_NEW_PLAN_R1|
    sysnames[0] = |PROCESS_PLAN|
    sysnames[1] = |ORDER_PROCESSING|
    nb_children = |7|

       ntype = |IF|
       valuetype = |IF|
       nvalue = |IF|
       nb_children = |1|

          ntype = |OPERATOR|
          valuetype = |AND|
          nvalue = |AND|
          nb_children = |2|

             ntype = |OPERATOR|
             valuetype = |EQ|
             nvalue = |=|
             nb_children = |2|

                ntype = |ITEM|
                valuetype = |IDENT|
                nvalue = |OI_STATUS|
                alias = |A|
                oepr = |ORDER_ITEM|

                ntype = |CONSTANT|
                valuetype = |STRING|
                nvalue = |DESIGNED|

             ntype = |OPERATOR|
             valuetype = |NOT|
             nvalue = |NOT|
             nb_children = |1|
         ntype = |FN_NAME|
         valuetype = |IDENT|
         nvalue = |exist|
         nb_children = |2|

            ntype = |OPERATOR|
            valuetype = |EQ|
            nvalue = |=|
            nb_children = |2|

               ntype = |ITEM|
               valuetype = |IDENT|
               nvalue = |PART_ID|
               alias = |A|
               oepr = |ORDER_ITEM|

               ntype = |ITEM|
               valuetype = |IDENT|
               nvalue = |PARTID|
               alias = |B|
               oepr = |PARTREV|

            ntype = |ITEM|
            valuetype = |IDENT|
            nvalue = |ORDER_LINE_ID|
            alias = |A|
            oepr = |ORDER_ITEM|

   ntype = |OPERATOR|
   valuetype = |ASSIGN|
   nvalue = |:=|
   executed_in = |PROCESS_PLAN|
   sysnames[0] = |PROCESS_PLAN|
   nb_children = |2|

      ntype = |ITEM|
      valuetype = |IDENT|
      nvalue = |PARTREV|
      alias = |B|
      oepr = |PARTREV|

      ntype = |FN_NAME|
      valuetype = |IDENT|
      nvalue = |request_design_revision|
      executed_in = |PROCESS_PLAN|
      sysnames[0] = |PROCESS_PLAN|
      nb_children = |1|

         ntype = |ITEM|
         valuetype = |IDENT|
         nvalue = |PART_ID|
         alias = |A|
         oepr = |ORDER_ITEM|
   ntype = |OPERATOR|
   valuetype = |ASSIGN|
   nvalue = |:=|
   nb_children = |2|

      ntype = |ITEM|
      valuetype = |IDENT|
      nvalue = |OI_STATUS|
      alias = |A|
      oepr = |ORDER_ITEM|

      ntype = |CONSTANT|
      valuetype = |STRING|
      nvalue = |IN ENG|

   ntype = |OPERATOR|
   valuetype = |ASSIGN|
   nvalue = |:=|
   nb_children = |2|

      ntype = |ITEM|
      valuetype = |IDENT|
      nvalue = |ROUTING|
      alias = |B|
      oepr = |PLAN|

      ntype = |CONSTANT|
      valuetype = |INTEGER|
      nvalue = ||

   ntype = |OPERATOR|
   valuetype = |ASSIGN|
   nvalue = |:=|
   nb_children = |2|

      ntype = |ITEM|
      valuetype = |IDENT|
      nvalue = |PLANSTATUS|
      alias = |B|
      oepr = |PLAN|

      ntype = |CONSTANT|
      valuetype = |STRING|
      nvalue = |WORKING|

   ntype = |UPDATE_DIRECTIVE|
   valuetype = ||
   nvalue = ||
   executed_in = |PROCESS_PLAN|
   sysnames[0] = |PROCESS_PLAN|

   ntype = |UPDATE_DIRECTIVE|
   valuetype = ||
   nvalue = ||
   executed_in = |ORDER_PROCESSING|
    sysnames[0] = |ORDER_PROCESSING|



THE MATRIX OF PRECEDENCE!!
0 -1 -1 -1 -1 0 0
1 0 0 0 0 -1 0
1 0 0 0 0 0 -1
1 0 0 0 0 -1 0
1 0 0 0 0 -1 0
0 1 0 1 1 0 0
0 0 1 0 0 0 0
before sort and optimize:

0 -1 -1 -1 -1 0 0
1 0 0 0 0 -1 0
1 0 0 0 0 0 -1
1 0 0 0 0 -1 0
1 0 0 0 0 -1 0
0 1 0 1 1 0 0
0 0 1 0 0 0 0
 Flist[0] set include:  0
 Flist[1] set include:  1
 Flist[2] set include:  2
 Flist[3] set include:  3
 Flist[4] set include:  4
 Flist[5] set include:  5
 Flist[6] set include:  6
after sort :

0 -1 -1 -1 -1 0 0
1 0 0 0 0 0 -1
1 0 0 0 0 0 -1
1 0 0 0 0 -1 0
1 0 0 0 0 0 -1
0 0 0 1 0 0 0
0 1 1 0 1 0 0
 Flist[0] set include:  0
 Flist[1] set include:  4
 Flist[2] set include:  3
 Flist[3] set include:  2
 Flist[4] set include:  1
 Flist[5] set include:  6
 Flist[6] set include:  5
after optimize :

0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 2 0 0 -1
0 0 0 0 2 0 0
0 0 1 0 0 0 0
0 1 1 1 0 0 0
 Flist[0] set include:  0  4  3  2  1
 Flist[1] set include:  6
 Flist[2] set include:  5

The below is partition 0:


the initial input value of operations of a partition: 0  4  3  2  1

the matrix for the operations in this partition:
0 -1 -1 -1 -1
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0

 the order of the operations in this partition: 0  1  2  3  4



The below is partition 1:


the initial input value of operations of a partition: 6

the matrix for the operations in this partition:
0

 the order of the operations in this partition: 6



The below is partition 2:


the initial input value of operations of a partition: 5

the matrix for the operations in this partition:
0

 the order of the operations in this partition: 5
END PARTITION
```

/* end of the program output */

The partitions generated by the decomposition algorithm is used by the ROPE message generation and each partition will become a subrule and stored in the message files sent to local application systems. Let's see the sub-rule portion of the message files generated for this rule.

For the above global-rule, because it only involves two different application systems, so all the sub-rules generated by the ROPE message generation will be stored in the two message files related to those two application systems, these two message files are mdbops0.000 and mdbpps0.000. The following just list portions of those two files to illustrate the sub-rules generated from the partitioning result of the global-rule decomposition algorithm.

```
INSERT PREPARE_NEW_PLAN_R1 : STARTS WITH pps$PREPARE_NEW_PLAN_R1$1
DECOMPOSED INTO
   ops$PREPARE_NEW_PLAN_R1$2 : ""
      (":store" )
      "pps$PREPARE_NEW_PLAN_R1$3"


"mdbops0.000" 13 lines, 331 characters


INSERT PREPARE_NEW_PLAN_R1 : STARTS WITH pps$PREPARE_NEW_PLAN_R1$1
triggered by time using "+?/?/? 1:00"
DECOMPOSED INTO
   pps$PREPARE_NEW_PLAN_R1$1 : "(A$ITEM_102.ITEM_98  =  ""DESIGNED"" ) AND
      NOT  exist.0.ORDER_LINE_ID$$$RESULT
"
      ("B$ITEM_66.ITEM_67  :=  ""WORKING"" " "B$ITEM_66.ITEM_73  :=  0 " "A$IT
EM_102.ITEM_98  :=  ""IN ENG"" " "B$ITEM_55.ITEM_55  := request_design_revision(
A$ITEM_102.ITEM_100 )" )
      "ops$PREPARE_NEW_PLAN_R1$2"

   pps$PREPARE_NEW_PLAN_R1$3 : ""
      (":store" )
       ""

"mdbpps0.000" 114 lines, 2991 characters
```

Part 5

Program Listing

This section contains most of the functions for local-rule implementation and global-rule decomposition, these functions are stored in the following *.c or *.pc files as following:

rule.pc	This file contains the main function for the local-rule implementation.
factbase.c	This file contains functions for determine independent queries, aggregate functions and the data items in the final factbase.
gen_mql.c	This file contains functions for generating the MQL views.
local.pc	This file contains functions for parsing the MQL file, finding the physical data to retrieve, and call the local code generation routines. This serves as the interface to the local system, and it also re-use most functions stored in mql.c, qproc.c which are not listed in this program listing.
gen_sql.c	This file contains functions for generating the Oracle SQL views.
gen_pc.c	This file contains functions for generating the Oracle Pro*C code.
getmdb.pc	This file contains functions for getting detailed information from Metadatabase.
timer.c	This file contains functions for parsing the trigger information and trigger the local-rules according to their event time. The modules re-used the lexical analyzer of ROPE shell which is stored in lexshell.c.
arrange.c	This file contains functions for re-arrange the global-rule tree to obtain a set of condition and actions processed in a single system.
part.c rule tree	This file contains functions for partitioning the nodes of the global- so that the nodes contained in each partition are executed in the same system.