Gilbert Babin and Cheng Hsu. "Decomposition of Knowledge for Concurrent Processing." IEEE Transactions on Knowledge and Data Engineering. 8(5). October 1996. pp. 758-772.

Decomposition of Knowledge for Concurrent Processing

Gilbert Babin and Cheng Hsu

The results presented in this paper were obtained while G. Babin was working on his Ph.D. dissertation at Rensselaer Polytechnic Institute [1].

G. Babin is with the Département d'informatique, Université Laval, Ste-Foy, Québec, Canada, G1K 7P4. E-mail: babin@ift.ulaval.ca.

C. Hsu is with the Department of Decision Sciences and Engineering Systems, Rensselaer Polytechnic Institute, Troy, New York, USA, 12180-3590. E-mail: hsu@rpi.edu.

Gilbert Babin and Cheng Hsu. "Decomposition of Knowledge for Concurrent Processing." IEEE Transactions on Knowledge and Data Engineering. 8(5). October 1996. pp. 758-772.

Abstract

In some environments, it is more difficult for distributed systems to cooperate. In fact, some distributed systems are highly heterogeneous and might not readily cooperate. In order to alleviate these problems, we have developed an environment that preserves the autonomy of the local systems, while enabling distributed processing. This is achieved by (1) modeling the different application systems into a central knowledge base (called a metadatabase), (2) providing each application system with a local knowledge processor, and (3) distributing the knowledge within these local shells. This paper is concerned with describing the knowledge decomposition process used for its distribution. The decomposition process is used to minimize the needed cooperation among the local knowledge processors, and is accomplished by "serializing" the rule execution process. A rule is decomposed into a ordered set of subrules, each of which is executed in sequence and located in a specific local knowledge processor. The goals of the decomposition algorithm are to minimize the number of subrules produced, hence reducing the time spent in communication, and to assure that the sequential execution of the subrules is "equivalent" to the execution of the original rule.

Keywords

Heterogeneous distributed database management systems, production systems, autonomous systems, distributed knowledge processing, knowledge distribution, metadatabase.

I. Introduction

Enterprises in today's global market place are hard pressed to deal with diversity, in both products and technologies. They typically need to customize their products (and therefore processes) to respond to customer's rapidly changing needs, and, at the same time, need to integrate multiple systems that are autonomous, distributed, and heterogeneous. These requirements place a great challenge to the underlying information technology.

One approach to deal with this situation has been developed at Rensselaer Polytechnic Institute (Troy, NY, USA), known as the Metadatabase approach, which uses a concurrent architecture containing: (1) a central knowledge base and (2) distributed rule processors. The central knowledge base, called a metadatabase, contains a description of the different application systems of the enterprise and the knowledge describing how these different application systems are integrated [2], [4], [5], [6], [7], [8], [9]. This knowledge includes database integrity rules, enforcing consistency across distributed databases, and business rules, describing how the information is transfered from one application system to the other.

For each of the application systems, we define a rule processor that will be in charge of executing the knowledge pertaining to a specific system, hence assuring that the application systems stay autonomous [1]. This approach, called ROPE (Rule-Oriented Programming Environment), therefore creates a distributed rule processing environment, where the fact bases and the inference engines are distributed. The collaboration of the rule processors is minimized by utilizing as much information contained in the metadatabase as possible.

One of the constraints imposed of the rule decomposition and execution is the fact that users may define routines that can be linked to the rule processor of one of the local systems. This allows for the

rule processing system to be expanded. However, the different user-defined routines (either procedures or functions) are usually localized in only one location (i.e., only one of the distributed processor executes a specific user-defined routine). This implies that the different rule processors must collaborate to a certain extent in order to execute the rule; i.e., the rule must execute the user-defined routine from the appropriate location. The approach we propose reduces, if not eliminates, this need for collaboration by decomposing the rule in such a way that (1) the different rule processors do not need to share the control of the execution of the rule, or (2) there is no need for a central process controller. Process control, however, is passed sequentially from one rule processor to the other, using a simple message protocol. Concurrent processing of rules occurs at a macroscopic level; i.e., although each individual subrule within a rule is executed in sequence, the set of rule processors may actually be processing multiple subrules pertaining to *different rules* concurrently.

From a technical standpoint, the central issue underlying the decomposition is the fact that the condition and action clauses of a rule may contain events and use routines both of which are distributed across the different rule processors. Thus, a rule must granularize its events and routines along with their associated logic in the rule such that each group will be executed in a particular rule processor, and all groups can be distributed to different processors. Specifically, the process of decomposition takes a rule and produces an equivalent set of subrules that satisfies the following criteria/conditions: (1) the serial behavior of the subrules must be equivalent to the original rule, (2) each element of a subrule is uniquely identifiable, (3) intra-rule data must be separated from inter-rule data to minimize coupling between rules, (4) data items' usage must be exactly determined, and (5) the set of subrules produced must be minimized.

The decomposition algorithm we have developed uses the fact that a rule can be broken down into five stages of execution: (1) rule triggering, (2) data retrieval, (3) condition evaluation and actions execution, (4) result storage, and (5) rule chaining. The idea is to serialize the execution of the rule using these five stages and to assure that the condition and actions of each subrule produced are localized in the same rule processor.

Another approach to the execution of distributed rules [10] will only distribute the condition of the rule, based on the data needed during its evaluation. Although efficient in distributed systems where each and every distributed processor is identical, it might prove inappropriate when the distributed processors are heterogeneous and the rules refer to specific routines located in specific processors.

Using the Metadatabase approach, we can also take advantage of the global query facility and its query language ([1],[4]) to create a temporary fact base, which in turn is used to process the rule. Once a rule is triggered, the rule processor currently executing it is in charge of building that temporary fact base. The fact base serves as a basis for the execution of the rule itself (condition and actions). Once the rule has finished execution, the content of the temporary fact base is used to update the actual data in the different persistent fact bases involved in the distributed environment.

The Metadatabase research project is still underway at Rensselaer and Université Laval (Ste-Foy,

Québec, Canada). Some earlier results include a prototype metadatabase, including a Metadatabase Management System, a Global Query System, and the Rule-Oriented Programming Environment for heterogeneous distributed database integration. The paper will focus on the decomposition of the rule logic into a set of subrules, this aspect being more useful outside the context of the Metadatabase research. A prototype of the Decomposition Algorithm presented in this paper is still under development.

In the next section, we introduce some definitions that will serve for the rest of the presentation. These definitions range from elements composing a rule to theoretical constructs used in the different algorithms. Section III puts emphasis on the logic underlying the decomposition of knowledge and presents some theoretical aspects of the localization of the different elements composing the decomposed rule. Section IV describes in detail, using a simple example, how a rule is broken into an optimal set of subrules. In Section V, we present a discussion on the approach used. Finally, Section VI contains a conclusion. Throughout the paper, we illustrate the different sections of the paper using a sample rule.

II. Definitions

The following definitions accompany the discussion on the decomposition process and will be used when describing the implementation design for the decomposition of the rule.

A. Rule and Subrule

A rule r is composed of a trigger t, a condition c and an ordered list of actions $\{a_i\}$. A rule is fired when the trigger event occurs. In the case of a chained (rule-triggered) rule, the rule is fired only if the actions of a chaining rule have been executed (see below). If the condition c is true, the actions $\{a_i\}$ are executed in order. If the rule contains only actions, then the condition c is implied and assumed to be true.

The trigger t represents a unique event activating the rule. In Rensselaer's metadatabase prototype environment, this event can be of one of three types: (1) a specific time has been reached, called time trigger, (2) some fact base content has been modified, called data trigger, or (3) some action has been performed by the local application, called program trigger. Corresponding to these event types, we define three types of rules: (1) time-triggered rules, (2) data-triggered rules, and (3) program-triggered rules. In addition to these three rule types, we define rule-triggered rules — or, simply, chained rules. Chained rules are rules that are triggered directly by other rules through rule chaining without using triggers. A rule r chains to rule r' if the actions in r can change the condition tested by r'. The rule r' is a chained rule and rule r is a chaining rule. We assert that time-triggered, data-triggered, and program-triggered rules cannot be chained rules (but may be chaining rules). The trigger is a firing condition of the rule where the condition is true only when the event occurs. Since a trigger is an event independently defined outside of any actions, it cannot be changed/reset directly from an action; and therefore, the trigger-using rules cannot be rule-triggered. Hence, a rule cannot belong to more than one of the above four classes.

Let us consider the following rule:

```
every evening at 6PM
if (is_completed(WO_ID) = FALSE) AND
   (due_date(CUST_ORDER_ID) > todays_date())
then
   status := "overdue";
   write_customer_notice(CUST_ORDER_ID);
```

The rule basically states that if an order is not completed and it is past due (the condition of the rule), its status should be set to "overdue" and a notice should be sent to warn the customer of that fact (the actions of the rule). The rule is evaluated every evening, at closing time (the trigger).

A subrule s_i of a rule r is composed of a condition c and an ordered list of actions $\{a_i\}$, just like a rule, but has no trigger. One of the actions in the subrule could be an update directive; this directive tells the current shell to execute its update query located in the current shell (see Sect. IV for an example). Note that the list of actions might be empty.

B. Expression and Operation

An expression, denoted $e(p_1, \ldots, p_n)$ where $p_1, \ldots, p_n$ represent the parameters of the expression, is defined as (1) a constant, (2) a variable (persistent or run-time), or (3) a function (including mathematical operators), producing a single value output based on an ordered list of expressions (parameters). In the case of a function, the value of each parameter is evaluated (in any order) before the function is evaluated based on these values. The output produced by an expression is (1) the value of a constant, (2) the content of a variable, or (3) the single value output from a function.

In our example, we can identify two constants:

- *FALSE*,
- "overdue";

there are three variables:

- *WO_ID*,
- *CUST_ORDER_ID*, and
- *status*;

finally, there are six functions:

- *is_completed*(*WO_ID*),
- *due_date*(*CUST_ORDER_ID*),
- *todays_date*(),
- >(*due_date*(...), *todays_date*()),
- =(*is_completed*(...), *FALSE*), and
- *AND*=(...),>(...)).

We use >(), =(), and *AND*() to represent the use of operators '>', '=', and 'AND', respectively.

An operation, denoted $o(p_1, \ldots, p_n)$ where $p_1, \ldots, p_n$ represent the parameters of the operation, is defined as (1) an expression, (2) a procedure, that performs some tasks based on an ordered list of expressions (parameters), (3) an assignment statement, modifying the value of one variable, based on an expression (the value and the expression are the parameters of the assignment statement), (4) a condition evaluation (i.e., the $if()$ statement), based on one expression (5) an update directive, or (6) a series of operations (see definition below). For assignment statements, The result from the expression is assigned to the variable. In the case of a procedure, the value of each parameter is evaluated (in any order) before the procedure is executed based on these values. A series of operations is defined as a list of operations $o_1, \ldots, o_n$, with at least one o_i being a procedure, an assignment statement, or an update directive. By definition, a rule is an operation that performs a series of operations.

In the example, in addition to the expressions identifier earlier, we can identify one procedure:

- *write_customer_notice*(*CUST_ORDER_ID*);

we identify one assignment statement:

- :=(*status*, "overdue");

there is one condition evaluation:

- *if*(*AND*(...));

finally, there is one series of operations, the rule itself. We use :=() to represent the an assignment statement.

The only operations that do include parameters are (1) functions, (2) procedures, (3) assignment statements, (4) an $if()$ statement, and (5) series of operation, in which case the parameters are the operations in the series itself. In order to perform one of these operations, we must first perform the operations defined by the parameters. The operation must therefore wait for at least one level of operation to execute. But each of these parameters might in turn have to wait for their parameters to be performed before they can. This means that the original operation must wait for at least two levels of operations to execute before it is itself executed. This notion is qualified using the notion of depth. The depth is defined as:

$$\mathsf{D}(o(p_1, \ldots, p_n)) = \begin{cases} 1 & \text{if } n = 0, \\ 1 + \max_{1 \le i \le n} \mathsf{D}(p_i) & \text{if } n > 0. \end{cases}$$

Two series of operations $o_1, \ldots, o_n$ and $o'_1, \ldots, o'_m$ are equivalent if and only if the output produced by the series $o_1, \ldots, o_n$ equals the output produced by $o'_1, \ldots, o'_m$, for every set of data items values, when the two series are executed on the same set of data items values.

C. Evaluation Tree of a Rule

A tree of evaluation for an operation is a tree where each leaf is a constant or a variable, and each node is an operation. A node n with subtrees $s_1, \ldots, s_k$, whose roots (called respectively $s_1, \ldots, s_k$) are direct descendents of n, performs the operation of node n using the results from the root operations of subtrees

$s_1, \ldots, s_k$ as parameters. The index i on s_i indicates that s_i is the i^{th} subtree of n, starting from the left. The notation will be used to represent a node and its subtrees: $n(s_1, \ldots, s_k)$. This notation emphasizes the fact that the node n is an operation using the results from the subtrees $s_1, \ldots, s_k$. When $k = 0$, the operation is a leaf operation. In short, we note a node using n or $n()$. The height of a tree with root $t(s_1, \ldots, s_n)$ corresponds to the number of nodes between the root and the leaf that is the most distant to the root (including the root and the leaf itself). It is defined as:

$$\mathsf{H}(t(s_1, \ldots, s_m)) = \begin{cases} 1 & \text{if } m = 0, \\ 1 + \max_{1 \le i \le n} \mathsf{H}(s_i) & \text{if } m > 0. \end{cases}$$

The tree of evaluation determines the order of execution of the operations in a rule. Given the domain of all nodes in the evaluation tree, noted N, we define the matrix of precedence as a function M: $N \times N \Rightarrow \{-1, 0, 1\}$ such that:

$$M(n, n') = \begin{cases} -1 & n \text{ must be evaluated before } n', \\ 1 & n' \text{ must be evaluated before } n, \\ 0 & \text{otherwise.} \end{cases}$$

Each node n is given an evaluation order number $E(n)$ such that

$$\begin{aligned} E(n) < E(n') &\Leftrightarrow M(n, n') = -1, \\ E(n) = E(n') &\Leftrightarrow M(n, n') = 0, \\ E(n) > E(n') &\Leftrightarrow M(n, n') = 1. \end{aligned}$$

Based on the matrix of precedence, we can generate an oriented graph $G(V, A)$, called the precedence graph. Each vertex $v \in V$ corresponds to a node $n \in N$. Furthermore, there is an arc from vertices v to v', corresponding respectively to nodes n and n' if and only if $M(n, n') = 1$. The strict precendence graph is a subgraph of the precendence graph where additionally, there exists no vertex v'', corresponding to node n'', such that $M(n, n'') = 1$ and $M(n'', n') = 1$.

Lemma 1: Isomorphism Between Evaluation Trees and Operations. For an operation

$$\begin{aligned} o(\quad & p_1(p_{11}(\ldots), \ldots, p_{1m_1}(\ldots)), \\ & \ldots, \\ & p_n(p_{n1}(\ldots), \ldots, p_{nm_n}(\ldots))) \end{aligned}$$

there exists one and only one isomorphic evaluation tree t, with

$$\mathsf{H}(t) = \mathsf{D}(o).$$

■

By isomorphic tree, we mean that there is a one-to-one correspondance between each node of t and each operations used in o, and that, for a node n, corresponding to operation o, the first child of n corresponds

to the first parameter of o, the second child of n corresponds to the second parameter of o, and so on. The proof to this lemma is done by induction on $\mathsf{D}(o)$, the depth of the operation, by showing that we can build such a tree (see [1] for a detailed proof).

Corollary 1: The evaluation order number of an operation is defined as the evaluation order number of its corresponding node in the isomorphic tree. ■

By Lemma 1, we know that each node of the tree is isomorphic to some operation. Hence, the operation can be substituted for the node, and vice versa. Therefore, $E(o)$ is defined as $E(n)$. Figure 1 shows the evaluation tree corresponding to the sample rule.

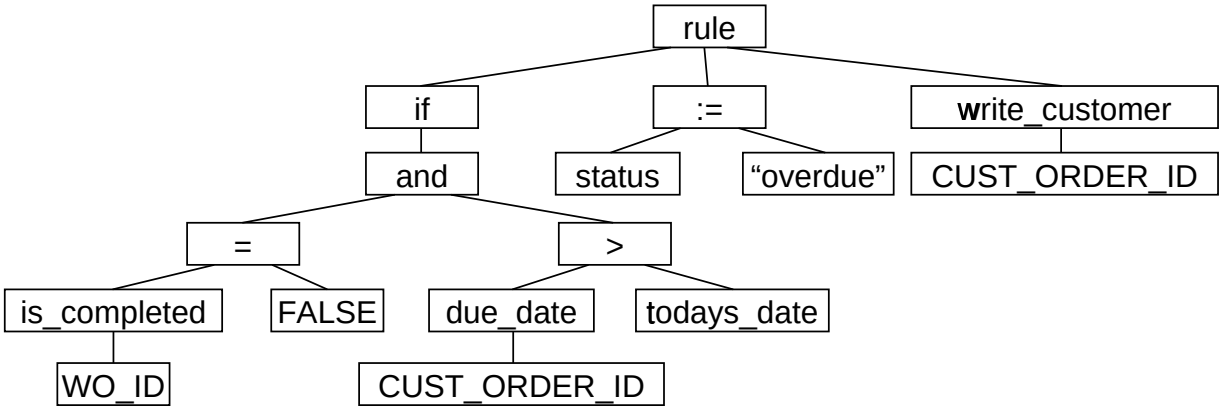

Fig. 1. An Evaluation Tree

D. Optimal Operation Partitioning

As defined in Section II-C, a rule is a series of operations. The parameters of the rule are themselves operations. Some of these operations can be executed be any rule processor:

- constants
- variables (persistent or run-time)
- system-defined routines (functions and procedures)
- assignment statements
- condition evaluations

Other operations, however, can only be executed in a specific rule processor. These operations are:

- user-defined routines (functions and procedures)
- update directives

The goal of the decomposition process is to serialize the execution steps of a rule in such a way that each execution step can be executed in the appropriate rule processor. As we will see in Section IV, we first make sure that all the top-level operations can be executed in a single rule processor. But that is not sufficient. We will also need to reorganize the operations to minimize inter-rule processor communications.

To achieve this, we define the concept of operation partitioning. The idea is to group operations in partitions where (1) every operation within a partition can be executed in the same rule processor and (2) the order of the partitions reflects the order of execution of the operations. The optimal operation partitioning will be a partitioning with the smallest number of partitions. In the following, we only mention user-defined routines and update directive, since they are the only operations that must be executed in a specific rule processor.

For a rule containing the user-defined routines and update directives $f_1()$, ..., $f_l()$, the optimal operation partitioning is defined as the ordered sequence of sets F_1, ..., F_m, containing user-defined routines and update directives, with the following properties: (1) for every pair of user-defined routines or update directive $f()$ and $f'()$ in partition F_i, $f()$ and $f'()$ are located in the same rule processor, (2) for every pair of user-defined routines $f()$ and $f'()$ in partition F_i, there exists no routine $f''()$ such that $E(f()) < E(f''()) < E(f'())$ unless $f''()$ is located in the same rule processor as $f()$ and $f'()$, (3) for each pair $f()$ and $f'()$ not located in the same rule processor, where $f()$ is in partition F_i and $f'()$ in partition F_{i+1}, $E(f()) \leq E(f'())$. The first two properties qualify the members of a partition; the members of a partition must be in the same rule processor and they must be executable in sequence. The last property qualifies the ordering of the different partitions; i.e., two consecutive partitions contain routines from different rule processors, and the set of routines in a partition must be executed before the routines in the next partition (in the ordering of partitions).

More formally, we have the partitions F_1, ..., F_m, where $F1 = \{r_{11}, \ldots, r_{1n_1}\}$, ..., $Fm = \{r_{m1}, \ldots, r_{mn_m}\}$. We define $S(r)$, to be a function that returns the name of the rule processor where r must be executed. The optimal operation partitioning will minimize m, such that:

$$M(r_{ij}, r_{lk}) \leq 0 \qquad 1 \leq i < m, \quad i < l \leq m, \quad 1 \leq j \leq n_i, \quad 1 \leq k \leq n_l;$$

$$M(r_{ij}, r_{lk}) \geq 0 \qquad 1 < i \leq m, \quad 1 \leq l < i, \quad 1 \leq j \leq n_i, \quad 1 \leq k \leq n_l;$$

$$S(r_{ij}) = S(r_{ik}) \qquad 1 \leq i \leq m, \quad 1 \leq j \leq n_i, \quad 1 \leq k \leq n_i;$$

$$S(r_{ij}) \neq S(r_{i-1,k}) \qquad 1 \leq i \leq m, \quad 1 \leq j \leq n_i, \quad 1 \leq k \leq n_{i-1};$$

$$S(r_{ij}) \neq S(r_{i+1,k}) \qquad 1 \leq i \leq m, \quad 1 \leq j \leq n_i, \quad 1 \leq k \leq n_{i+1}.$$

The first condition states that every operation in a partition must be executed before every operation in the following partitions. The second condition states that every operation in a partition must be executed after every operation in the preceding partitions (redundant with the first condition). The third condition states that operations within the same partition must be in the same system, while the fourth and fifth conditions ensure the operations in partitions adjacent to the current partition are in different systems than the operations in the current partition.

E. Global Queries

Traditionally, a global query is defined as any type of queries (retrieval, insertion, deletion, and update queries) to be performed on multiple distributed databases. In the context of this work, we limit the use of global query to global retrieval queries. The decomposition process takes advantage of previous results in the field of global query processing; starting from the set of data items to be retrieved, the Global Query System [4] generates (1) a set of local queries in the local database DML, used to retrieve data items from the different databases involved in the global query, and (2) an integration script written in MQL (Metadatabase Query Language, defined in [4], with extensions in [1]), used to assemble the results from the local queries.

III. The Basic Logic of Decomposition

The decomposition algorithm determines where the elements constituting each rule are to be located. These elements are: (1) the trigger, (2) a global query to create the initial rule's fact base, (3) a set of subrules, and (4) a set of update queries. The decomposition algorithm is used (1) to populate a new rule processor and (2) whenever the global rule base is modified. The rule population only occurs once for each rule; the rule, however, can be used any number of time. Therefore, the efficiency of the decomposition algorithm is not an issue.

The following lemmas determine how to decompose the rule and where each element produced should be localized. They all make use of the classification of rules provided in Section II.

Lemma 2: Trigger Localization. Data triggers and program triggers should be located in particular rule processors where their data or program resides. Time triggers and chaining triggers can be arbitrarily placed in any rule processors. ■

The logic of a data trigger is to detect changes occurring in a specific database table. This database table is located in only one rule processor, hence determining the shell where the data trigger must be located. Program triggers detect events occurring within a specific application system, which in turn determines that the trigger must be located in the rule processor corresponding to that application system. Time triggers refer to a specific time that was reached. This type of event has no relationship with the rule processors, and can therefore be detected in any shell. Chaining triggers are also independent from any rule processor; they represent the end of the execution of a specific rule.

Lemma 3: Global Query Localization. The global query for generating the temporary fact base of the rule should be located in the same rule processor as the rule trigger. ■

One of the criteria for evaluating the decomposition process is the total number of subrules produced, for each subrule produced would add one more message to the shells to execute the rule. By placing the global query at the same location as the rule trigger, we remove the need to transmit the trigger to another shell, and thereby reduce the total number of messages needed. Furthermore, it also removes the need to define special message types in the Message Language to communicate the triggers to the other shells.

Lemma 4: Subrules Localization. If a rule does not use any user-defined routines nor modify any data items, the decomposition process should produce exactly one subrule (the rule itself) which can be placed in any rule processor. If the rule uses one or more user-defined routine or modifies one or more data items, the decomposition process should produce one subrule for each group in the optimal operation partitioning of the set of all such operations (user-defined routines and data item modification). Each of these subrules is located in a particular rule processor where the operation should take place. ■

First, in Section II, we have explained that user-defined routines must be linked to the local rule processor in order for the shell to use them. However, the different routines can be located in different rule processors. Hence, the rule must be decomposed to execute each user-defined routine in the appropriate shell at the appropriate time. Second, the rules operate on persistent data items; when a value is assigned to any of these data items, it is necessary to update its value in the local databases, at the end of the rule execution. Obviously, the database modification must occur in a specific rule processor. The decomposition process must determine what changes are needed in which database, in which rule processor. To obtain the optimal decomposition of the rule, we employ the optimal operation partitioning of user-defined routines and database assignments which is defined in Section IV. This partitioning assures minimum number of subrules. For rules that do not use any of such operations, there will be no requirement on their location since the global query initially fetches all necessary data for the rule execution, hence building the rule's initial fact base. Some of them, however, may satisfy certain special conditions and thus must follow certain rules as delineated in the next lemma.

Lemma 5: First Subrule and Trigger Localization. The trigger and the first subrule produced by the rule decomposition process should reside in the same rule processor whenever it is possible — i.e., when either one can be placed in any rule processor. ■

By storing the first subrule in the same rule processor as the trigger, we do not reduce the total number of messages produced to execute the rule, but do reduce the total time needed to execute the rule. The data retrieval is located in the same rule processor as the trigger (Lemma 3), hence, the message generated to execute the first subrule will already be in the appropriate rule processor, thereby eliminating the need for transmitting a message to another shell and reducing the total execution time.

Theorem 1: Rule Element Localization Theorem. Data-triggered and program-triggered rules. The

global query is in the same rule processor as the trigger. If the rule uses no user-defined routines or modifies no data item, the unique subrule produced is also located in the same rule processor as the trigger. Time-triggered and rule-triggered rules. The global query and the trigger are in the same rule processor as the rule's first subrule. If the rule uses no user-defined routines or modifies no data item, all the elements are located together, in any rule processors. ■

This theorem follows directly from Lemmas 2 through 5.

IV. The Decomposition Algorithm

In this section, we will explain in detail how the subrules are produced. The only assumption we make about functions and procedures (user-defined routines) is that they do not change the value of their parameters. If a user-defined routine does change its parameters, it can be rewritten as multiple functions. For example, in the function $f(a, b, c)$, the value of c is modified, based on the value of a, b, and c, we could write a new function $g(a, b, c)$ to assign the new value to c (:=(c, $g(a, b, c)$)).

The decomposition algorithm proceeds to implement the above logic as follows:

I. Generate constructs corresponding to the trigger definitions contained in the rule.

II. Generate a global query to retrieve the data items used by the rule and the queries to store the rule's results.
  IIa Identify the rule's global queries.
  IIb Determine the data items to retrieve and to update.
  IIc Generate the global query.
  IId Generate the update queries.

III. Generate a set of subrules corresponding to the rule's condition and actions.
  IIIa Rearrange the condition and actions (Sect. IV-A).
  IIIb Obtain an optimal operation partitioning (Sect. IV-B).
  IIIc Generate the subrules (Sect. IV-C).

IV. Generate rule chaining information.

For each rule, the decomposition process will generate: (1) trigger information (including chaining information), (2) a global query, used to initially create the rule's fact base, (3) update queries, used to store the values updated by the rule, and (4) the subrules produced. In Step I, we extract the trigger information from the rule and convert it to the appropriate construct used by the rule processor. This step is straightforward and does not in any way influence the rest of the rule text. At the end of this step, the trigger is removed and the sample rule would become:

```
if (is_completed(WO_ID) = FALSE) AND
   due_date(CUST_ORDER_ID) > todays_date())
then
   status := "overdue";
   write_customer_notice(CUST_ORDER_ID);
```

The next step, namely Step II, deals with persistent data items. First, it recognizes such items and generate the appropriate global query to retrieve them across the distributed fact bases. Second, it

determines which of these items are updated by the rule, creating update queries (one per application system) to modify the relevant fact bases. For each query, it will generate an "update directive" to be added in the rule. The "update directive" is used to tell the rule processor when and where an update query should be executed. Once this step is completed, we obtain the following rule (which is equivalent to the original rule):

```
if (is_completed(WO_ID) = FALSE) AND
   (due_date(CUST_ORDER_ID) > todays_date())
then
   status := "overdue";
   write_customer_notice(CUST_ORDER_ID);
   update fact base for system Order Processing;
```

We then proceed in the algorithm by decomposing the rule into a set of equivalent set of subrules (Step III). Sections IV-A– IV-C will focus on this process. The rule serialization proceeds in 3 substeps: (1) rearrange the condition and actions to obtain a set of condition and actions processed in a single rule processor (Step IIIa; see Sect.

IV-A), (2) obtain an optimal operation partitioning of the user-defined routines and update directives (Step IIIb; see Sect. IV-B), and (3) generate the subrules for a rule (Step IIIc; see Sect. IV-C).

Finally, the rules are analyzed to determine the chaining information, i.e., which rule should be considered for firing when a rule finish executing (Step IV). This way, the different rule processors do not have to determine the rules to fire whenever a rule finishes, but rather look the content of a table indicating which rules to fire.

A. Step IIIa: Rearrange the condition and actions

The goal of this step is to obtain a sequencing for the operations required by the condition and actions such that each can be executed in a single rule processor. We achieve this by using the concept of a tree of evaluation for the rule. The trees of evaluation of the condition ($if(exp)$) and the actions a_1, ..., a_l become subtrees of the rule's tree of evaluation. The resulting tree is $rule(if(exp), a_1, \ldots, a_l)$.

If the original tree is:

$$rule(\ldots, T, \ldots)$$

where subtree T (either the condition of the rule or an action) includes a node (an operation) $t_i(s_1, \ldots, s_n)$, then the objective of this sequencing is to ensure that every s_i be in the same rule processor as t_i. Let us assume that there exists s_j, where s_j is in a rule processor different from s_k (the user-defined routine executed by s_j is in a different rule processor than s_k). We need to reorganize the rule so that s_j is performed before T, to preserve the evaluation order (parameters must be evaluated before the operation using them). This is achieved by generating a new subtree T', such that T' is the operation $:=(temp,s_j)$, and by replacing node t_i by node $t'_i(s_1, \ldots, s_{j-1}, temp, s_{j+1}, \ldots, s_n)$ in subtree T. We place subtree T'

just before subtree T in rule evaluation tree. Specifically, T' becomes a direct child of the rule (a sibling of T), but must be placed to the left of T, like this:

$$rule(\ldots, T', T, \ldots)$$

Because T' is evaluated before T, the rearranged rule is equivalent to the original rule. At this point the two subtrees can be executed in different rule processors, yet, the final result is equivalent to the original rule. We use this process recursively, starting from the root of the whole rule. Let $App(o)$ and $Sys(o)$ be defined as follows:

$App(o)$ The (name of the) rule processor where operation o is to be executed. This is set to NIL if o can be executed in any processor.

$Sys(o)$ The set of all rule processors needed to perform operation o.

We define the rule rearrangement algorithm as follows:

IIIa.1 Generate $r(o_1, \ldots, o_n)$ the isomorphic tree of the rule.

IIIa.2 Determine $App(o)$:
- Traverse $r()$ in preorder. For each node o
  - if o is a user-defined routine
    then $App(o) \leftarrow S(o)$
  - if o is an update directive
    then $App(o) \leftarrow S(o)$
  - otherwise $App(o) \leftarrow$ NIL

IIIa.3 Determine $Sys(o)$:
- Traverse $r()$ in preorder. For each node o:
  - if o is a leaf
    then $Sys(o) \leftarrow \{App(o)\}$
    where $\{App(o)\} = \emptyset$, if $App(o) =$ NIL.
  - if $o(o_1, \ldots, o_n)$ is not a leaf,
    then $Sys(o) \leftarrow Sys(o_1) \bigcup \ldots \bigcup Sys(o_n) \bigcup \{App(o)\}$
    where $\{App(o)\} = \emptyset$, if $App(o) =$ NIL.

IIIa.4 Serialize the rule $r(o_1, \ldots, o_n)$
- For every o_i, $1 \leq i \leq n-1$, in reverse order
  - if $App(o_i) =$ NIL and $Card(Sys(o_i)) = 1$
    then $App(o_i) \leftarrow$ the unique element in $Sys(o_i)$
  - if $App(o_i) =$ NIL and $Card(Sys(o_i)) > 1$ and $\exists\ o_j$, where $App(o_j) \in Sys(o_i)$ and j is the smallest $j > i$
    then $App(o_i) \leftarrow App(o_j)$
- For every o_i, $1 \leq i \leq n$
  - $Serialize(r, o_i)$.

The procedure *Serialize*(), which uses *Serialize_assign*() and *Serialize_operation*(), can be found in Appendix A. To illustrate the use of the Rule Rearrangement Algorithm, consider our sample rule. The user-defined routine *is_completed*() is located in system Shop Floor Control, while all the other user-defined routines — namely, *due_date*(), *todays_date*(), and *write_customer_notice*() — are located in the

Order Processing System. Furthermore, the update directive "update fact base for system Order Processing" is also located in the Order Processing System. Figures 2 and 3 illustrate the rule evaluation tree before and after the rearrangement. The following lemmas and theorem are used to prove the exactness of the rearrangement algorithm. Their proofs can be found in [1]. Lemmas 6 and 7 define some properties of the evaluation number within an operation and a series of operations, respectively. More specifically, Lemma 6 raises the fact that parameters to procedures and functions can be executed in any order, while Lemma 7 shows that some operations within a series of operations can be permuted, as long as they respect the relative evaluation order of these operations.

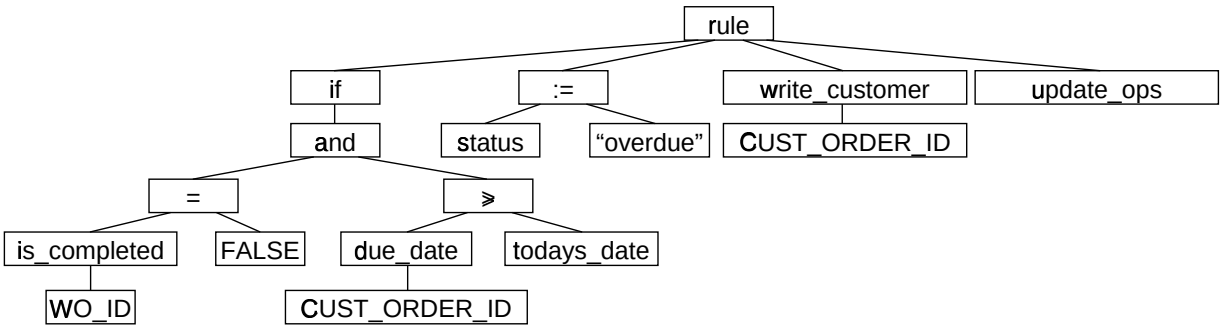

Fig. 2. Tree of Evaluation Before Rearrangement

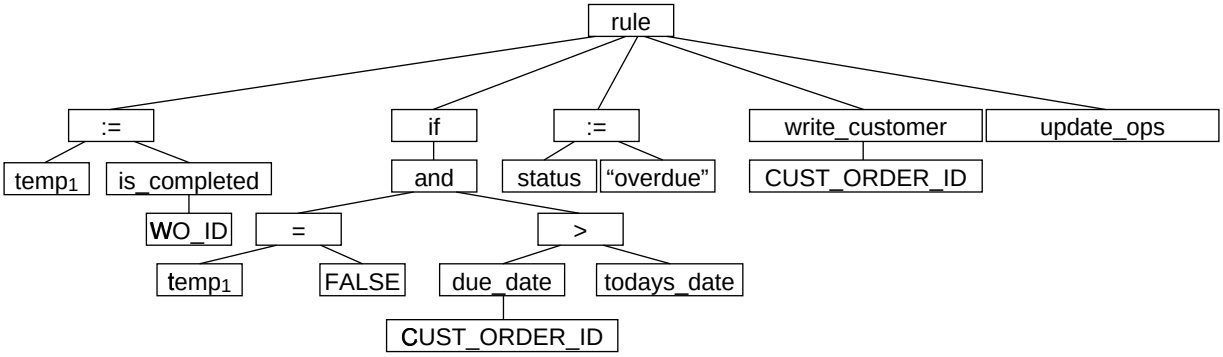

Fig. 3. Tree of Evaluation After Rearrangement

Lemma 6: Evaluation Order of Sibling Expressions. For an operation

$$Op(e_1, \ldots, e_n)$$

where Op is a procedure or a function with the sibling expressions e_1, ..., e_n as parameters, then

$$E(e_1) = \ldots = E(e_n).$$

■

Lemma 7: Permutations of Operations. The series of operations

$$Op(o_1, \ldots, o_n)$$

with

$$E(o_1) \leq \ldots \leq E(o_n)$$

can be rewritten into the series of operations

$$Op'(o_{\rho(1)}, \ldots, o_{\rho(n)})$$

with $\rho()$ defining a permutation of 1, ..., n and

$$E(o_{\rho(1)}) \leq \ldots \leq E(o_{\rho(n)})$$

■

Lemmas 8 through 11 describe how operations can be modified without affecting the final result. For instance, Lemma 8 states that any procedure, function, or condition evaluation can be evaluated by first assigning the value of all parameters to temporary variables (in any order) and then using these temporary variables as parameters to the operation. On the other hand, Lemma 9 shows that any number of these parameters can be assigned to temporary variables.

Lemma 8: Serialization of an Operation. The operation

$$Op(e_1, \ldots, e_n)$$

where Op is not an assignment statement nor a series of operations, and e_1, ..., e_n are expressions, can be rewritten into the equivalent series of operations

$$:= (v_{\rho(1)}, e_{\rho(1)}), \ldots, := (v_{\rho(n)}, e_{\rho(n)}), Op(v_1, \ldots, v_n)$$

where $\rho()$ is defining a permutation of 1, ..., n and v_1, ..., v_n are variables. ■

Lemma 9: Deserialization of Operations. The series of operation

$$:= (v_1, e_1), \ldots, := (v_n, e_n), Op(v_1, \ldots, v_n)$$

where Op is not an assignment statement nor a series of operations, e_1, ..., e_n are expressions, v_1, ..., v_n are variables, and $E(:=(v_1, e_1)) = \ldots = E(:=(v_n, e_n))$. can be rewritten into the equivalent series of operation

$$:= (v_1, e_1), \ldots, := (v_{i-1}, e_{i-1}), := (v_{i+1}, e_{i+1}), \ldots,$$

$$:= (v_n, e_n), Op(v_1, \ldots, v_{i-1}, e_i, v_{i+1}, \ldots, v_n)$$

■

Lemma 10 concentrates on the case of assignment statements, stating that the subexpressions, used to evaluate the expression assigned to a temporary variable, can be evaluated before the expression is assigned to the variable. In turn, Lemma 11 indicates that only some subexpressions can be assigned to temporary variables.

Lemma 10: Serialization of an Assignment Statement. The operation

$$:= (v, e(e_1, \ldots, e_n))$$

where v is a variable and e, e_1, ..., e_n are expressions, can be rewritten into the equivalent series of operation

$$:= (v_{\rho(1)}, e_{\rho(1)}), \ldots, := (v_{\rho(n)}, e_{\rho(n)}), := (v, e(v_1, \ldots, v_n))$$

with $\rho()$ defining a permutation of 1, ..., n and v_1, ..., v_n are variables. ■

Lemma 11: Deserialization of Assignment Statements. The series of operation

$$:= (v_1, e_1), \ldots, := (v_n, e_n), := (v, e(v_1, \ldots, v_n))$$

where v is a variable, e, e_1, ..., e_n are expressions, v_1, ..., v_n are variables, and $E(:=(v_1, e_1)) = \ldots = E(:=(v_n, e_n))$. can be rewritten into the equivalent series of operation

$$\begin{aligned} &:= (v_1, e_1), \ldots, := (v_{i-1}, e_{i-1}), \\ &:= (v_{i+1}, e_{i+1}), \ldots, := (v_n, e_n), \\ &:= (v, e(v_1, \ldots, v_{i-1}, e_i, v_{i+1}, \ldots, v_n)) \end{aligned}$$

■

Lemmas 12 and 13 validate the procedures defined to perform the serialization process, respectively *Serialize_assign*() and *Serialize_operation*(). Their proof relies on Lemmas 6 through 11. We conclude this section with the Rule Serialization Theorem, which shows that the function *Serialize*() produces a series of subrules equivalent to the original rule.

Lemma 12: Assignments Serialization Process. The procedure *Serialize_assign* (r, $:=(v, e(a_1, \ldots, a_n))$) generates a series of operations equivalent to r. ■

Lemma 13: Operation Serialization Process. The procedure *Serialize_operation* (r, $o(p_1, \ldots, p_n)$) generates a series of operations equivalent to r, if o is not an assignment statement. ■

Theorem 2: Rule Serialization Theorem. The Rule Rearrangement Algorithm produces an operation (the rule itself) equivalent to the original operation. ■

This theorem follows direcly from Lemmas 12 and 13, since *Serialize*() calls either function *Serialize_assign*() or function *Serialize_operation*(), based on the operation passed as a parameter.

B. Step IIIb: Obtain an optimal operation partitioning

After the evaluation tree is rearranged, we are assured that every condition and action used in the (rearranged) rule can be executed in a single rule processor. In Step IIIb, we generate a partition F_1, ..., F_m for all the user-defined routines and update directives used by the rule. In the query generation algorithm (Step II.d), we added directives to process the update queries for specific fact bases. By default, these directives are placed at the end of the rule's action list. However, these directives can be moved

TABLE I

CREATING THE MATRIX OF PRECEDENCE

Name	Guideline
1. Descendants	If n is a node and s is a direct descendant of n then $M(n,s) \leftarrow 1$ and $M(s,n) \leftarrow -1$
2. Transitivity	If $M(n,n') = 1$ and $M(n',n'') = 1$ then $M(n,n'') \leftarrow 1$ and $M(n'',n) \leftarrow -1$
3. Non-permutable nodes	If operations o and o', with corresponding nodes n and n', respectively, cannot be permuted, then $M(n,n') \leftarrow 1$ and $M(n',n) \leftarrow -1$

around within the rule, as long as they are executed after the last assignment to a variable stored in that rule processor is performed. Hence, if actions a_1, a_5, and a_6 are the only actions modifying values of variables from the Shop Floor Control System, and the directive d executes the update query, then $E(d) > E(a_1)$, $E(d) > E(a_5)$, and $E(d) > E(a_6)$. We use this knowledge when creating the optimal partition.

The guidelines presented in Table I determine how to create the matrix of precedence. Guideline 1 is straightforward: you cannot perform an operation before all its parameter operations have been performed. Guideline 2 recursively elicit all the dependencies expressed in the rule. The parameters of the parameter of an operation must be performed before the operation itself. Guideline 3 extracts the semantics of the rule and generates the appropriate dependencies between operations (condition and actions). Specifically, Guideline 3 applies in the following situations: (1) the condition of the rule cannot be permuted with any action or (2) for two actions, a_1 and a_2, such that a_1 is placed before a_2 in the original rule, i.e., the modeler wants to execute a_1 before a_2, and a_2 uses the results from a_1. The Rule Partitioning Algorithm uses the rearraned rule and reorders the top-level operations to obtain an optimal operation partitioning of user-defined routines and update directives.

We define:

$id(o)$ A unique identifier, such that for two sibling operations o and o', $id(o) < id(o')$ if and only if o is on the left of o' in the isomorphic evaluation tree.

$App(F)$ The rule processor where the operations in the partition must be executed.

$\mathcal{M}(F, F')$ The matrix of precedence of the partitions defined as

$$\mathcal{M}(F,F') = \begin{cases} -1 & o \in F, o' \in F', M(o,o') = -1, \\ 1 & o \in F, o' \in F', M(o,o') = 1, \\ 0 & \text{otherwise.} \end{cases}$$

Then, the algorithm is as follows:

IIIb.1 Generate the matrix of precedence for the rule.

IIIb.1.1 Assign unique identifier $id(o)$ to the operations

— Traverse the isomorphic evaluation tree in preorder

— $id(o) \leftarrow$ the position of node o in the traversal

IIIb.1.2 Apply Guidelines 1–3 to all operations.

— if o has parameter o'
then $M(o,o') \leftarrow 1$, $M(o',o) \leftarrow -1$.

— if c is the condition and a is an action then $M(c,a) \leftarrow -1$, $M(a,c) \leftarrow 1$.

— if a is an assignment statement and v is a variable, with $id(a) < id(v)$ and a assigns a new value to v
then $M(a,v) \leftarrow -1$, $M(v,a) \leftarrow 1$.

— if $a(v,e)$ is an assignment statement and d is an update directive with $id(a) < id(d)$ and a assigns a new value to v and $App(v) = App(d)$
then $M(a,d) \leftarrow -1$, $M(d,a) \leftarrow 1$.

— While there are changes in matrix M

— if $M(o,o') = 1$ and $M(o',o'') = 1$
then $M(o,o'') \leftarrow 1$, $M(o'',o) \leftarrow -1$.

IIIb.2 Modify the matrix of precedence to obtain the strict precedence graph for $o_1, \ldots, o_n$ in the rule $r(o_1, \ldots, o_n)$

IIIb.2.1 Remove all entries for operations other than $o_1, \ldots, o_n$

IIIb.2.2 Remove transitive precedence dependencies

— While there are changes in matrix M

— if $M(o,o') = -1$ and $M(o',o'') = -1$ and $M(o'',o) = 1$
then $M(o'',o) \leftarrow 0$.

— For all o, o' in $o_1, \ldots, o_n$

— if $M(o,o') = -1$ and $M(o',o) = 0$
then $M(o,o') \leftarrow 0$

IIIb.3 Generate an optimal operation partitioning of $o_1, \ldots, o_n$ in the rule $r(o_1, \ldots, o_n)$

IIIb.3.1 Generate the initial partition $F_1, \ldots, F_m$

— $F_i \leftarrow \{o_i\}$

— $App(F_i) \leftarrow App(o_i)$, $1 \le i \le n$

— $\mathcal{M}(F_i,F_j) \leftarrow M(o_i,o_j)$, $1 \le i \le n$, $1 \le j \le n$

IIIb.3.2 Merge adjacent partitions executed in the same rule processor

— While there are changes in the partition

— if $\mathcal{M}(F,F') = -1$ and ($App(F) = App(F')$ or $App(F') =$ NIL)

— $F \leftarrow F \bigcup F'$

— $\mathcal{M}(F,F') \leftarrow 0$ and $\mathcal{M}(F',F) \leftarrow 0$

— For all partitions F'', if $\mathcal{M}(F',F'') = 1$

— $\mathcal{M}(F',F'') \leftarrow 0$ and $\mathcal{M}(F'',F') \leftarrow 0$

— $\mathcal{M}(F, F'') \leftarrow 1$ and $\mathcal{M}(F'', F) \leftarrow -1$

IIIb.3.3 Sort the initial partition by execution order

— $F_1, \ldots, F_m \leftarrow Sort\_partition(F_1, \ldots, F_m)$

IIIb.3.4 Optimize the partition

— $F'_1, \ldots, F'_{m'} \leftarrow Optimize\_partition(F_1, \ldots, F_m)$

Function *Sort_partition*(), which uses functions *Optimize_partition*() and *Found_all_partitions*(), is given in Appendix A. The following discussion justifies the partitioning process by stating Lemma 14, which indicates that the Rule Partitioning Algorithm captures all the dependencies between the operations, and by illustrating its usage with our sample rule.

Lemma 14: Total Dependency Generation. The Rule Partitioning Algorithm captures all the operation' precedence dependencies in Step IIIb.1. ■

First, by definition, the parameters to an operation must be evaluated before the operation is processed. This case is handled by the rule

if o has parameter o'
then
  $M(o, o') \leftarrow 1$,
  $M(o', o) \leftarrow -1$.

Next, the condition must be evaluated before any action is executed. The rule

if c is the condition and
  a is an action
then
  $M(c, a) \leftarrow -1$,
  $M(a, c) \leftarrow 1$.

deals with that situation. In addition, since only assignment statements can modify variables used by the rule, by definition, we only need to define dependencies between the assignment statements and the operations using the result from the assignments. Furthermore, only two types of operations can be influenced by the assignments (1) the modified variable itself and (2) the update directive for the fact base where the variable is located. The rules

if a is an assignment statement and
  v is a variable, with
    $id(a) < id(v)$ and
    a assigns a new value to v
then
  $M(a, v) \leftarrow -1$,
  $M(v, a) \leftarrow 1$.

if $a(v, e)$ is an assignment statement and
  d is an update directive with
    $id(a) < id(d)$ and

a assigns a new value to v and
$App(v) = App(d)$
then
$M(a, d) \leftarrow -1$,
$M(d, a) \leftarrow 1$.

enter these dependencies in the matrix of precedence. Finally, we generate any transitive dependency implied by previously defined dependencies using rule

if $M(o, o') = 1$ and
$M(o', o'') = 1$
then
$M(o, o'') \leftarrow 1$,
$M(o'', o) \leftarrow -1$.

until no new dependency is created.

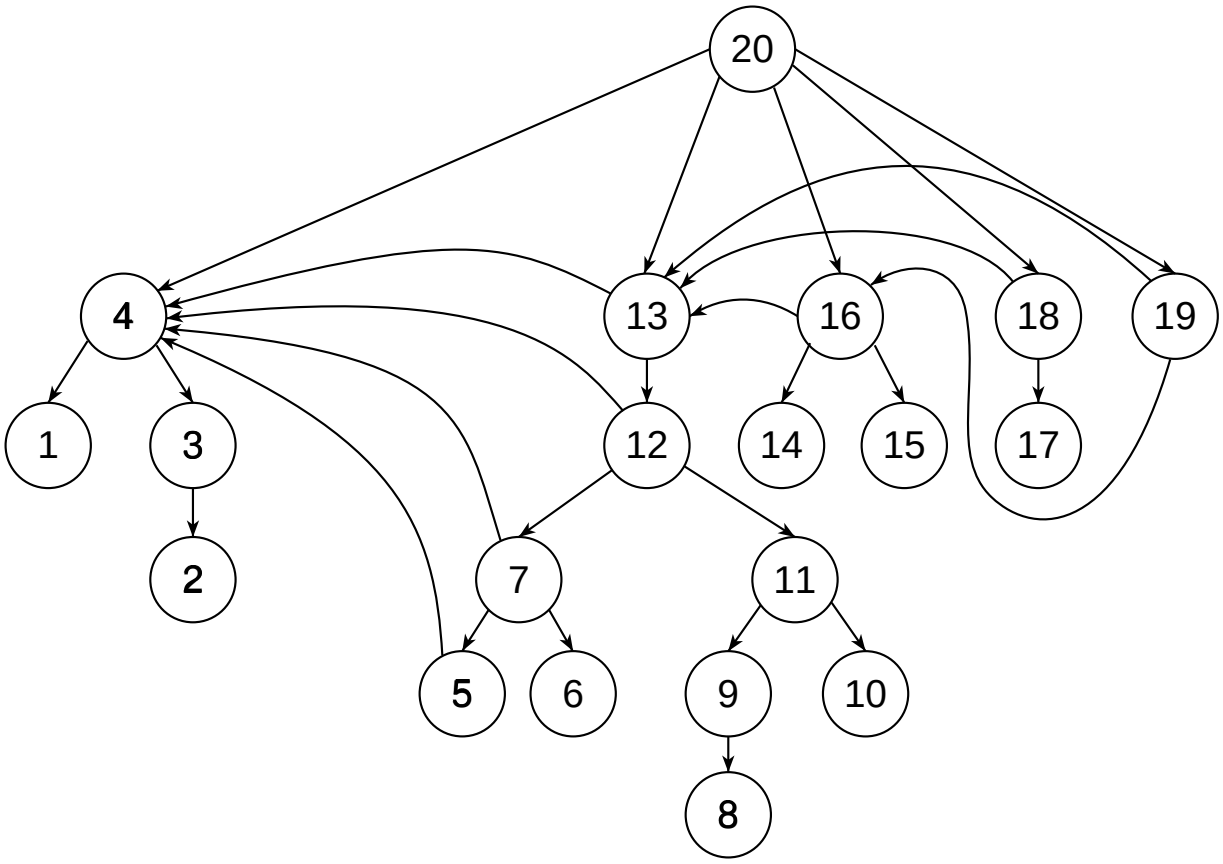

Fig. 4. Precedence Graph Between Operations

In our example, initially, the Rule Partitioning Algorithm generates the matrix of precedence for every operation in the rearranged rule, making explicit all the dependencies between operations (Step IIIb.1). This is illustrated in Figure 4, using the precedence graph corresponding to the whole matrix $M()$. Note

that the nodes' number correspond to the preorder traversal of the tree. The rules used to obtain the matrix are complete and generate all the dependencies needed. We use the transitive dependencies to elicit all the dependencies.

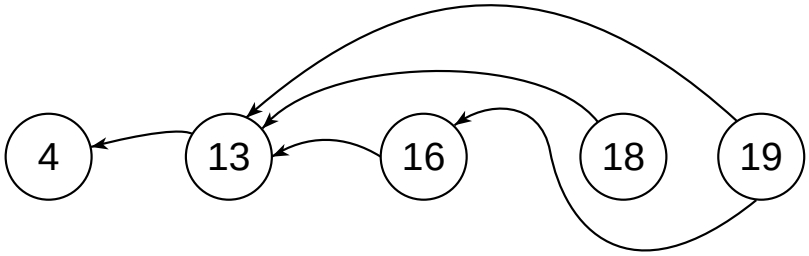

Fig. 5. Precedence Graph for Top-level Operations

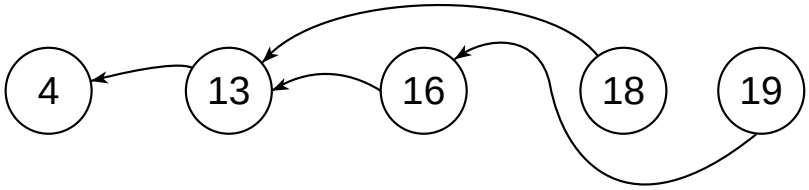

Fig. 6. Strict Precedence Graph

To make sure that all the dependencies are elicited, we must keep all the operations in the evaluation tree. Once this is done, however, any dependency between the rule condition and actions should also be explicited. Since we only need to keep track of dependencies among the condition and actions, we can remove the nodes corresponding to parameters to the condition and actions. For the same reason, we can remove the root node, which represent the whole rule itself (Step IIIb.2.1; Figure 5). Furthermore, we remove the transitive precedence dependencies, which are no longer needed. This results in a matrix of precedence containing minimal information, as illustrated by the strict precendence graph shown in Figure 6 (Step IIIb.2.2).

After we have the strict precedence graph, we create an optimal operation partition (Step IIIb.3). We first create the initial partition by placing the condition and every action in a separate set (Step IIIb.3.1). Then, we merge adjacent partitions F and F' (i.e., $\mathcal{M}(F, F') = 1$ and $\mathcal{M}(F', F) = -1$), when either (1) F and F' is executed in any rule processor (i.e., $App(F) = \text{NIL}$ or $App(F') = \text{NIL}$) or (2) F and F' are executed in the same rule processor (i.e., $App(F) = App(F')$) (Step IIIb.3.2). The resulting partition for our example is shown in Figure 7. The next step is to linearize the partitions; i.e., we place them in a linear sequence, preserving the precedence dependencies (Step IIIb.3.3). This makes the optimization process easier. Finally, we optimize the sequence of sets (obtained from the linearization) by alternatively (1) grouping adjacent sets executed in the same rule processor and (2) permuting sets that have no precedence dependencies (i.e., $\mathcal{M}(F, F') = 0$) as long as either operation modify the current

partition (Step IIIb.3.4). In Figure 7, this ordering is trivial, since the second partition is dependent on the execution of the first partition. If such was not the case, however, we could alter the order of the partitions, without any change in the logic of the rule.

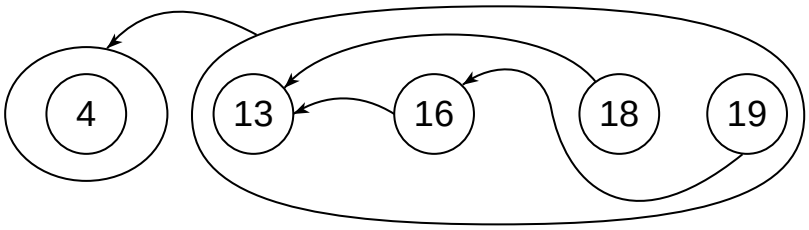

Fig. 7. Partition Optimization

C. Step IIIc: Generate the subrules

For each set F_i in the optimal operation partition, we generate one subrule $S_i()$ that will execute the actions implied by the user-defined routines and update directives stored in F_i. The next algorithm takes the partition generated by the Rule Partitioning Algorithm and generates the subrules (condition and actions) composing the original rule.

IIIc.1 Generate an ordered list of operation in each partition
- — For every partition F containing operations $o_1, \ldots, o_n$
  - — $o_1, \ldots, o_n \leftarrow Sort\_operations(o_1, \ldots, o_n)$

IIIc.2 Generate a subrule for each partition
- — For each partition F_i, $1 \leq i \leq m$, containing operations $o_1, \ldots, o_n$
  - — generate subrule identifier as $App(F_i)\$rule\_id\$i$
  - — for each o_j, $1 \leq j \leq n$
    - — write the appropriate Message Language statements

The function $Sort\_operations()$, which uses function $Found\_all\_operations()$ is defined in Appendix A. Figure 8 shows the different order of evaluation of the original operations in the second partition (Fig. 7). The algorithm used to determine the actual order of operations is logically equivalent to the algorithm used to determine the order of the different partitions.

Theorem 3: Rule Partitioning Theorem. The Rule Partitioning Algorithm and the Subrule Generation Algorithm together generate a series of operation equivalent to the original rule. ■

Since the Rule Partitioning Algorithm generates all the dependencies between operations, by Lemma 14, and that all the permutations performed on the operations preserve their relative order of evaluation, the ordering of the operations is equivalent to the original rule, by Lemma 7. Therefore, the subrules generated are equivalent to the original rule.

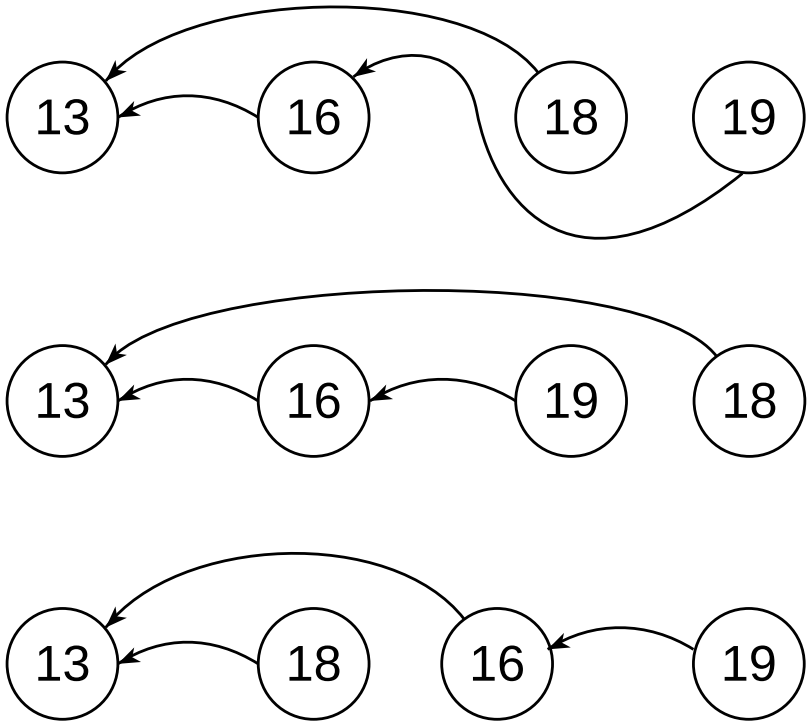

Fig. 8. Valid Order of Evaluation of Operations

V. Discussion

A. Subrule-Rule Equivalence

The only changes we make to the rule when transforming it for decomposition are to rearrange the tree of evaluation and store intermediary results in temporary variables (Step IIIa of the Decomposition Algorithm). Let us look at the following rule:

$$\text{if } (x > y) \text{ then } w := f(x, g(y, w)); \tag{1}$$

This rule can be rewritten as:

$$\text{if } (x > y) \text{ then } temp := g(y, w); w := f(x, temp); \tag{2}$$

These two rules are equivalent. When evaluating an expression, we generate an evaluation tree to determine in which order the different operations must be performed. In (1), we first evaluate the value of x, then the value of y. Next, we compare these two values; if the result from the comparison is true, we then execute the assignment statement. To do so, we first evaluate the value of each parameter. In the case of (1), we evaluate x and $g()$. Therefore, we can see that $E(x) < E(f()) < E(:=(w,f))$ and $E(g()) < E(f()) < E(:=(w,f))$. Since we assume parameters to a function call cannot be modified, $g()$ can be evaluated before x. Hence, if we evaluate $g()$ before $:=()$, then $E(g()) < E(:=())$, which implies

that $E(g()) < E(f())$. We can see that the order of evaluation is preserved in (2). Therefore, we can state that (2) is equivalent to (1).

B. Minimal Set of Subrules

The Decomposition Algorithm minimizes the total number of subrules generated. The total number of rules depends on two factors: the number of user-defined functions (i.e., routines; n_f) used in the rule and the number of application databases updated by the rule (n_u). There are in total n_u queries in the set U, the set of update directives in the rule.

Recall that a $rule(s_1, \ldots, s_n)$ is decomposed into subrules $S_1(), \ldots, S_j()$ according to, among other things, user-defined functions and update directives. The maximal number of subrules produced for each rule is $n_f + n_u$ if $n_f \neq 0$ or $n_u \neq 0$; otherwise the number is 1, when $n_f = 0$ and $n_u = 0$. The algorithm partitions the user-defined routines and update directives used in the rule into a minimal partition F_1, ..., F_m, such that for every $f()$ and $g()$ in the partition F_i, then $f()$ and $g()$ are located in application system "app" and there exists no $h()$ located in an application system different from "app", such that $E(f()) < E(h()) < E(g())$. This is accomplished by the Decomposition Algorithm as discussed in Section IV. Each F_i is therefore the largest set of functions located in the same application, executed in sequence. By construction, we will generate one subrule for each Fi_i. Therefore, $m \leq n_f$. Because it represents the optimal operation partitioning of the function calls, it is the smallest number of subrules needed to execute these functions.

We argue that minimizing the total number of subrules will reduce the total time needed to execute the rule. In order to assess the performance of executing the (original) global rule in the distributed design with respect to the best and worst case scenarios, we make the following assumptions: (1) it takes s steps to execute a global rule, (2) each step occurs on a different knowledge processor, (3) it takes on average t units of time to execute an operation for a step (i.e., execute a subrule, a local query), (4) it takes n units of time to transmit a message, and (5) messages are processed every d unit of time. In the best case, there are no delays, hence it takes $s \cdot t$ units of time to perform the operation. In the worst case, we have to wait $d + n$ units of time between each step. In that case, the execution time is $s \cdot t + (s - 1) \cdot (n + d)$. The worst case equation provides some insight on how to optimize the performances of the system. We can see that the only theoretical variable in the formula is the number of steps; the more steps we have per rule, the longer it will take to execute the rule. This is why the decomposition algorithm minimizes the number of subrules to reduce the number of times a message is passed to another system. The rest are technological parameters: the performance can only be improved by (1) reducing the processing time t with a faster rule processor, (2) increasing the speed of the network, hence reducing n, and (3) increasing the frequency at which messages are processed ($1/d$).

VI. Conclusion

The Rule Decomposition Algorithm is one element making possible the concurrent execution and management of knowledge. The decomposition algorithm extracts all the knowledge needed from the metadatabase to process the enterprise's integrity and business rules. By providing sufficient information to the rule processors, the distribution of knowledge enables them to process the knowledge in a concurrent, autonomous fashion.

There are current research efforts in database systems focusing on the inclusion of knowledge at the conceptual schema. The different results from those efforts make it possible to (1) abstract the knowledge from the application systems' code, making maintenance easier, (2) store data constraints knowledge within the DBMS itself, and (3) automatically enforce these constraints within the DBMS [3], [11], [12], [13]. All these efforts are concentrating on the processing of knowledge in a single DBMS.

However, with the growing number of distributed information systems, results on the distributed execution of rules must also be achieved, as is done in [10]. We can ask ourselves if the solution to the distributed execution of rules is complete. We basically want to know: (1) how to decompose a rule, (2) how to distribute the decomposed parts of the rule, and (3) how to execute the rule [10]. First, the *Decomposition Algorithm* (Sect. IV) prescribes how the rule should be broken down into subrules. Second, the *Rule Element Localisation Theorem* (Sect. III) indicates where each components of the decomposed rule must be stored. Finally, we have explained the rule execution process which is at the basis of the decomposition process (Sect. I).

Acknowledgments

The research presented in this paper was sponsored by the National Science Foundation (grant # DDM 9215620), the AIME Program (Rensselaer Polytechnic Institute), and Samsung Electronics. The authors would like to thank Brahim Chaib-draa for his comments on the first version of the paper. We would also like to thank the reviewers of IEEE Transactions on Knowledge and Data Engineering for the constructive comments they provided.

References

[1] G. Babin, *Adaptiveness in Information Systems Integration*, PhD thesis, Decision Sciences and Engineering Systems, Rensselaer Polytechnic Institute, Troy, N.Y., August 1993.

[2] M. Bouziane, *Metadata Modeling and Management*, PhD thesis, Computer Sciences, Rensselaer Polytechnic Institute, Troy, N.Y., June 1991.

[3] S. Chakravarthy, B. Blaustein, A.P. Buchmann, U. Dayal M. Carey, D. Goldhirsch, M. Hsu, R. Jauhari, R. Ladin, M. Livny, D. McCarthy, R. McKee, and A. Rosenthal, "Hipac: A research project in active, time-constrained database management", Tech. Rep. XAIT-89-02, XAIT Reference no. 187, Xerox Advanced Information Technology, Cambridge, MA, July 1989.

[4] W. Cheung, *The Model-Assisted Global Query System*, PhD thesis, Computer Sciences, Rensselaer Polytechnic Institute, Troy, N.Y., November 1991.

[5] C. Hsu, "The metadatabase project at rensselaer", *ACM Sigmod Record*, vol. 20, no. 4, pp. 83–90, 1991.

[6] C. Hsu and G. Babin, "Rope: A rule-oriented programming environment for adaptive, integrated multiple systems", in *ISMM First International Conference on Information and Knowledge Management*, Baltimore, MD, November 1992, p. 663.

[7] C. Hsu and G. Babin, "A rule-oriented concurrent architecture to effect adaptiveness for integrated manufacturing enterprises", in *International Conference on Industrial Engineering and Production Management*, Mons, Belgium, June 1993, pp. 868–877.

[8] C. Hsu, G. Babin, M. Bouziane, W. Cheung, L. Rattner, and L. Yee, "Metadatabase modeling for enterprise information integration", *Journal of Systems Integration*, vol. 2, no. 1, pp. 5–39, January 1992.

[9] C. Hsu, M. Bouziane, L. Rattner, and L. Yee, "Information resources management in heterogeneous, distributed environments: A metadatabase approach", *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp. 604–625, June 1991.

[10] I.-M. Hsu, M. Singhal, and M.T. Liu, "Distributed rule processing in active databases", in *Proceedings of the 8th Intl Conf. on Data Engineering*, July 1992, pp. 106–113.

[11] D.R. McCarthy and U. Dayal, "The architecture of an active data base management system", in *Proceedings of the 1989 ACM SIGMOD*, New York, NY, 1989, ACM Press, pp. 215–224.

[12] Y.-M. Shyy and S.Y.M. Su, "K: A high-level knowledge base programming language for advanced database applications", in *Proceedings of the 1991 ACM SIGMOD*, New York, NY, May 1991, ACM Press, pp. 338–347.

[13] M. Stonebraker, A. Jhingran, J. Goh, and S. Potaminos, "On rules, procedures, caching and views in data base systems", in *Proceedings of the 1990 ACM SIGMOD*, New York, NY, May 1990, ACM Press, pp. 281–290.

Appendix

I. Routines for Decomposition of Knowledge

We provide here the definition of the routines used by the different algorithms.

$Serialize(root, o)$
begin
  if o is an assignment statement
    $Serialize\_assign(root, o)$
  else
    $Serialize\_operation(root, o)$
end

$Serialize\_assign(root, o(v, e(a_1, \ldots, a_n)))$
begin
  for $i = 1$ to n
    if $Card(Sys(a_i)) > 1$ or ($App(a_i) \neq App(o)$ and $App(a_i) \neq$ NIL)
      rewrite $root(\ldots, o(v, e(\ldots, a_i, \ldots)), \ldots)$
        as $root(\ldots, o', o(v, e(\ldots, t_i, \ldots)), \ldots)$
        where o' is operation $:=(t_i, a_i)$
      if $App(a_i) =$ NIL and $App(o) \in Sys(a_i)$
        $App(o') \leftarrow App(o)$
      else
        $App(o') \leftarrow$ NIL
      end if
      $App(t_i) \leftarrow$ NIL
      $Sys(t_i) \leftarrow \emptyset$
      $Serialize\_assign(root, o')$

if $App(a_i) = App(o)$ or $App(o) = \text{NIL}$
rewrite $root(\ldots, o', o(v, e(\ldots, t_i, \ldots)), \ldots)$
as $root(\ldots, o(v, e(\ldots, a_i, \ldots)), \ldots)$
$App(o) \leftarrow App(a_i)$
end if
end if
end for
$Sys(o) \leftarrow \{App(o)\}$ where $\{App(o)\} = \emptyset$, if $App(o) = \text{NIL}$.
end

$Serialize\_operation(root, o(p_1, \ldots, p_n))$
begin
for $i = 1$ to n
if $Card(Sys(p_i)) > 1$ or $(App(p_i) \neq App(o)$ and $App(p_i) \neq \text{NIL})$
rewrite $root(\ldots, o(\ldots, p_i, \ldots), \ldots)$
as $root(\ldots, o', o(\ldots, t_i, \ldots), \ldots)$
where o' is operation $:=(t_i, p_i)$
if $App(p_i) = \text{NIL}$ and $App(o) \in Sys(p_i)$
$App(o') \leftarrow App(o)$
else
$App(o') \leftarrow \text{NIL}$
end if
$App(t_i) \leftarrow \text{NIL}$
$Sys(t_i) \leftarrow \emptyset$
$Serialize\_assign(root, o')$
if $App(p_i) = App(o)$ or $App(o) = \text{NIL}$
rewrite $root(\ldots, o', o(\ldots, t_i, \ldots), \ldots)$
as $root(\ldots, o(\ldots, p_i, \ldots), \ldots)$ $App(o) \leftarrow App(p_i)$
end if
end if
end for
$Sys(o) \leftarrow \{App(o)\}$ where $\{App(o)\} = \emptyset$, if $App(o) = \text{NIL}$.
end

$Sort\_partition(F_1, \ldots, F_m)$

returned value: A new partition $F_{\rho(1)}, \ldots, F_{\rho(m)}$, where $\rho(1)$, $\ldots$, $\rho(m)$ is a permutation of $1, \ldots, m$.

begin
$P \leftarrow \emptyset$
$R \leftarrow \emptyset$
$A \leftarrow \emptyset$
for $i = 1$ to m
$found \leftarrow$ false
$j \leftarrow 1$
while not $found$ and $j \leq m$
$found \leftarrow \mathcal{M}(F_i, F_j) = -1$
$j \leftarrow j + 1$
end while

```
      if not found
         P ← P ⋃ {i}
      else
         R ← R ⋃ {i}
      end if
   end for
   c ← m
   while P ≠ ∅
      A ← A ⋃ P
      P' ← ∅
      R' ← ∅
      for all i ∈ P
         for k = 1 to m
            exchange M(F_i, F_k) and M(F_c, F_k)
            exchange M(F_k, F_i) and M(F_k, F_c)
         end for
         exchange F_c ← F_i
         c ← c − 1
      end for
      for all j ∈ R
         if M(F_c, F_j) ≠ 1 and Found_all_partitions(F_j, A)
            P' ← P' ⋃ {j}
         else
            R' ← R' ⋃ {j}
         end if
      end for
      P ← P'
      P' ← ∅
      R ← R'
      R' ← ∅
   end while
   return F_1, ..., F_m
end
```

$Found\_all\_partitions(F, A)$

returned value: Returns true if all partitions that must be executed after F are in A. Returns false otherwise.

```
begin
   all_used ← true
   i ← 1
   while i ≤ m and all_used = true
      if M(F, F_i) = −1 and F_i ∉ A
         all_used ← false
      end if
      i ← i + 1
   end while
   return all_used
```

end

$Optimize\_partition(F_1, \ldots, F_m)$

returned value: A new partition $F'_1, \ldots, F'_{m'}$.

begin
  $change \leftarrow$ true
  while $change$ = true
    $change \leftarrow$ false
    for $i = m$ to 2
      if $App(F_{i-1}) = App(F_i)$ or $App(F_i) = NIL$)
        $F_{i-1} \leftarrow F_{i-1} \bigcup F_i$
        $\mathcal{M}(F_{i-1}, F_i) \leftarrow 0$ and $\mathcal{M}(F_i, F_{i-1}) \leftarrow 0$
        for $j = 1$ to m
          if $\mathcal{M}(F_i, F_j) = 1$
            $\mathcal{M}(F_i, F_j) \leftarrow 0$
            $\mathcal{M}(F_j, F_i) \leftarrow 0$
            $\mathcal{M}(F_{i-1}, F_j) \leftarrow 1$
            $\mathcal{M}(F_j, F_{i-1}) \leftarrow -1$
          end if
        end for
        $m \leftarrow m - 1$
        $i \leftarrow i - 1$
        $change \leftarrow$ true
      end if
    end for
    for $i = 1$ to $m - 1$
      if $\mathcal{M}(F_i, F_{i+1}) = 0$
        for $j = 1$ to m
          exchange $\mathcal{M}(F_i, F_j)$ and $\mathcal{M}(F_{i+1}, F_j)$
          exchange $\mathcal{M}(F_j, F_i)$ and $\mathcal{M}(F_j, F_{i+1})$
        end for
        exchange F_i and F_{i+1}
        $change \leftarrow$ true
      end if
    end for
  end while
  return $F_1, \ldots, F_m$
end

$Sort\_operations(o_1, \ldots, o_n)$

returned value: A new series of operations $o_{\rho(1)}, \ldots, o_{\rho(n)}$, where $\rho(1), \ldots, \rho(n)$ is a permutation of $1, \ldots, n$.

begin
  $P \leftarrow \emptyset$
  $R \leftarrow \emptyset$
  $A \leftarrow \emptyset$
  for $i = 1$ to n

$found \leftarrow false$
$j \leftarrow 1$
while not $found$ and $j \leq n$
$found \leftarrow M(o_i, o_j) = 1$
$j \leftarrow j + 1$
end while
if not $found$ and o_i is not a condition
$P \leftarrow P \bigcup \{i\}$
else
$R \leftarrow R \bigcup \{i\}$
end if
end for
$c \leftarrow n$
while $P \neq \emptyset$
$A \leftarrow A \bigcup P$
$P' \leftarrow \emptyset$
$R' \leftarrow \emptyset$
for all $i \in P$
for $k = 1$ to n
exchange $M(o_i, o_k)$ and $M(o_c, o_k)$
exchange $M(o_k, o_i)$ and $M(o_k, o_c)$
end for
exchange $o_c \leftarrow o_i$
$c \leftarrow c - 1$
end for
for all $j \in R$
if $M(o_c, o_j) = 1$ and o_c is not a condition
and $Found\_all\_operations(o_j, A)$
$P' \leftarrow P' \bigcup \{j\}$
else
$R' \leftarrow R' \bigcup \{j\}$
end if
end for
$P \leftarrow P'$
$P' \leftarrow \emptyset$
$R \leftarrow R'$
$R' \leftarrow \emptyset$
end while
for all $i \in R$
for $k = 1$ to n
exchange $M(o_i, o_k)$ and $M(o_c, o_k)$
exchange $M(o_k, o_i)$ and $M(o_k, o_c)$
end for
exchange $o_c \leftarrow o_i$
$c \leftarrow c - 1$
end for

return $o_1, \ldots, o_m$

end

$Found\_all\_operations(o, A)$

returned value: Returns true if all partitions that must be executed after o are in A. Returns false otherwise.

begin

$all\_used \leftarrow$ true

$i \leftarrow 1$

while $i \leq n$ and $all\_used =$ true

if $M(o, o_i) = 1$ and $o_i \notin A$

$all\_used \leftarrow$ false

end if

$i \leftarrow i + 1$

end while

return all_used

end