Transactional Programming for Distributed Agent Systems

V.K. Murthy
School of Information Technology and Mathematical Sciences, University of Ballarat, Ballarat, Victoria 3353, Australia

Abstract

A new multiagent programming paradigm based on the transactional logic model is developed. This paradigm enables us to construct a Distributed agent transactional program (DATRAP). Such a construction is carried out in two stages: first expressing a program into a production rule system , and then converting the rule applications into a set of transactions on a database of active objects represented using high-level datastructures. The formal specification and refinement calculus are key features in the development of a DATRAP. We also indicate how to specify granularity of parallelism and also achieve several types of parallelism. One can associate with a DATRAP two different types of execution semantics called set-based and instance-based semantics. We also show how to prove correctness of DATRAP, achieve maximal concurrence and reduce the complexity of a distributed program.

1. Introduction

The interplay between artificial intelligence(AI) and software engineering has led to an extremely useful area of research that has commercial and industrial applications. In particular, the problem involved in the design of a distributed agent program is analogous to the distributed problem solving paradigm in AI, where given an initial state of a problem we attempt to reach the goal by passing through a sequence of subgoals by applying a set of production rules [1], [2], [3]. Hence, the AI problem solving paradigm can be applied to devise a distributed agent program, if each program is converted to a set of production rules and the rules are implemented as transactions acting on a database of active objects (represented using high-level datastructures), that describe the initial state, the intermediate states and the final state of the program. The realisation of production rule applications as transactions will be referred to as a "Distributed Agent Transactional Program" (DATRAP). In a DATRAP, rule conditions are expressed as queries, while rule actions contain algebraic description of the operations performed on the database. Thus the state of computation consists of a collection of named values in an active database, where the names correspond to variables and the values are assigned from the problem domain. A state maps the variable to its corresponding value. The initial state specifies the initial condition of the problem, while the final state specifies the result. The rule actions activate the database through a successsion of states (transitions). The query or condition and updates or actions can be expressed using first order logic formula or equivalently by relational expressions.

It is well known that [4] a transaction has the four properties - called ACID properties: Atomicity (indivisibility and either all or no actions or carried out), Consistency (before and after the execution of a transaction), Isolation (no interference among the actions), Durability (recovery under failure and achieving consistency). The transactional programming paradigm can hence provide for cooperation among competing actions or processes, by resolving conflicts due to data dependence and resource dependence.

Associated with the transactional paradigm is a logical model called the transactional logic model with three basic components [5] :

1. A database D which expresses facts in first order logic; that is, the content of the database is specified as a list of predicates, with specific bindings (substitution of constant values) for all its arguments.
2. A set S of transitions (elementary actions); S is more like ground rules free of any variables or atomic formula, called the name of the transition.
3. A set T of formula in first order logic called "Transactional formula or transactions" that formulate queries and update the database D using set S. (We assume that a query does not update the database). Thus T contains rules defining queries and updates tells us what to do using primitive actions from S and changes D; The functions used in S can be absorbed by T, if necessary. The database D is independent of T. That is D does not define predicates in terms of transactions; or the predicate symbols occurring in rule-heads in T cannot occur as rule bodies in D.

1.1 Operators in transactional logic

The logic of transactional paradigm is an action logic paradigm that differs from the classical situation logic paradigm. Here, we need a new connective called "serial succession operator" to specify total order, namely a particular action has to temporally precede a particular action. This operator is denoted by $; for example x$y says; do x before and then do y. The $ operator therefore takes into account the side effects due to performing action x before and doing action y after. That is action x serves as precondition

0-8186-7267-6/96 $05.00 © 1996 IEEE

for action y, and realizing action y is a post condition for action x. In practical terms, $ sets up an a priori consistency for b and this ensures serializability of transactions. This operator $ can combine two successive subgoals or paths that lead to a solution. Note that $ is more general than the conventional conjunction 'And' that is applicable to the two paths or subgoals simultaneously. Also note that $ can be identified with the action "Begin-on-Commit" (BCD) [6] which states that the Action b can only begin after a commits. This action is denoted by b BCD a; if a is not Committed then b is rejected. Similar operators have been suggested in linear logic [7].

We can associate with $, a dual operator called 'serial choice" operator %- that permits choice of one or the other action path in a serial path, but not both. The combinator term a % b means: choose action a now or choose action b after.

The serial connectives are useful in problem solving using transactional pardigm, where preconditions trigger actions. These operators lead to Serial Horn clauses that can specify a context in a production system [8]. Also these operators can describe concepts such as partial order and serializability.

We also use another operator # for a posteriori consistency as in:"End(Commit) b only after a (Commits) ends" denoted by a#b.This is a conditional synchronization and describes a cooperative schedule (casacade abort free and recoverable) . In ACTA [6] this is equivalent to the Strong Commit action (SCD) :b SCD a; which means, if a commits b commits as well.

Remark: The operator $ is similar to (,) in PROLOG but not identical ; the functor (,) specifies conjunction of goals; e.g. X,Y: the goal X,Y succeeeds if X succeeds and Y succeeds. If X succeeds and Y fails then, then an attempt is made to resatisfy X. If X fails the whole conjunction fails. This is the essence of backtracking. However, Prolog does not undo updates while backtracking. So we need to modify Prolog in order to introduce the Transaction logic approach.

The operator(;) in Prolog specifies a disjunction; if X and Y are goals then X;Y succeeds if X or Y succeeds; again % resembles (;) but they are not identical since Prolog does not undo updates in backtracking. Essentially we need to undo all $ and % actions if the total chain fails.

1.2. Serializability, cooperation and recovery

The notion of serializability is essentially concerned with the conflict equivalence of an interleaved schedule to a serial schedule (namely, the conflicting actions in the non- aborted transactions are performed in the same temporal order). Hence it ensures a priori (means :from what is before) consistency in a competitive environment (or provides for the well-being of an individual transaction).

However, this does not ensure how other transactions that read from a given transaction in a schedule have been affected and what steps are to be taken to ensure them to be healthy, free from abort and are recoverable (well-being of a set of interleaved transactions, called a schedule) after a transaction terminates. This provides for a cooperative environment. In order to make a schedule recoverable, we need to ensure that a transaction j that read(R) from a write(W) of a transaction i, must ensure that j is committed (C) only after i commits. This is a posteriori (from what comes after) consistency check.

To avoid cascade aborts we must ensure that a transaction j always reads a committed transaction i. Strict schedules (S) are schedules in which all reads and writes for any object take place only after those transactions that wrote the object have committed or aborted; this ensures the restorability of values. The set of recoverable schedules contains the set of cascade-abort-free schedules (CA) and CA contains the set of strict schedules [4].

We call a set of transactions cooperative if they are helping each other and prevent chain aborts and allow each other to recover under failure. Strict schedules try to optimize cooperation and competition. However, they may not be serializable. The class of serial schedules is properly contained within both the classes of serializable and the strict schedules. To maximize concurrency we must find a class of competing schedules larger than the class of serial schedules and yet cooperative [4].

Remark: There are some interesting correspondences between the Transactional and action paradigms [9],[10]. These will be discussed elsewhere.

2. Transactional problem solving

Problem solving consists in finding a partially ordered sequence of application of desired operators that can transform a given initial state to a desired final state - called goal. Reaching a goal through a sequence of totally ordered subgoals- is called a linear solution. For many problems a totally ordered concatenation of subgoals may not exist . Such problems require a nondeterministic search. A solution based on a nondeterministic search is called a nonlinear solution. Such a solution uses an act - verify strategy. Human problem solving also uses this strategy through preconditions and actions. The transactional paradigm provides for an act and verify strategy by offering a nonprocedural style of programming (called **'subjunctive programming'**) in which a hypothetical action (what if changes) is followed by verification and restoration. So this paradigm is well-suited for a nondeterministic / probabilistic solutions [11]. If the granularities of the transactions are chosen suitably we can provide for maximum cooperation among competing subgoals.

The subjunctive program uses a tentative execution; if the condition is safe the actions are made permanent else aborted and the recovery to the initial state begins.

Various strategies used in automating problem solving, namely, forward or backward chaining, means-end analysis, least commitment approach can be realised using a transactional paradigm.

3. Transactional paradigm

The transactional paradigm is a concurrent programming paradigm. In order to understand its semantics- namely termination and confluence properties, we need to associate with it a basic computational model. Parallel computational models such as PRAM are basically derived from the Turing machine model. However, the Post production system model (and the related language) [2],[3] seems ideally suited to implement the transactional paradigm since the condition- action pair is realised through database- query , namely Read (or Refer) and Write (delete, insert or update) actions.

3.1. Production systems

The production system paradigm occupies a prominent place in AI. This system consists of a set of rewrite rules consisting of a left-hand-side expression (LHS) and a right-hand side- expression (RHS). Given any string drawn from a database that matches the LHS, the corresponding RHS is substituted. Thus we may substitute expressions, constants or null elements permitting update, evaluation, insertion or deletion operations.

Several types of production systems are used,

1. Monotonic : Here the application of one rule does not prevent the application of another rule that could have also been applied at the time when the first rule was selected.
2. Nonmonotonic: Here the application of one rule prevents the application of another rule.
3.Partially commutative: If the application of a particular sequence of rules transforms the sytem from State 1 to State 2 , then any interleaved set of rules in the sequence would equally well do the same transformation from State 1 to State 2.
4.Commutative: A system that is both monotonic and partially commutative is called commutative. Any problem that can be solved by any other production system can be solved by commutative system- but may not be efficiently. Non-monotonic commutative systems are useful for problems in which changes occur but can be reversed and the order in which operations occur is not critical. Non-partially commutative production systems are useful when irreversible changes occur; here the order is important.

The implementation of a production system operates in three-phase cycles: matching, selecting and execution. The cycle halts when the database fulfills a termination condition. The task of match phase is similar to query matching - that is unification of the rules with the database.This phase returns a conflict set that satisfies the conditions of different rules. In the select phase we select those compatible rules after conflict resolution.In the execution phase all selected rules are fired and actions are implemented.

A crucial difference between ordinary conditionals (or guards) and rules is that the left side of the rule is a set of existentially quantified predicates. That is the match is successful only if there are elements in the memory such that the predicates of positive condition elements are simultaneously satisfied and there are no elements that satisfy the predicates of the negated condition elements. Thus the interpreter does not simply test the expressions to determine whether they are true ; rather it engages in a search to bind variables in such a way to make the expression true. It is this characteristic of pattern matching along with the dynamic size of the working memory , that gives the production systems their distinctive capabilities. In this sense it is more powerful than ordinary action systems (Section 1.3) and are as powerful as logic programs. Also such a rule is equivalent to the quantified assignment statement in Unity [12].

Parallelism can be achieved in match and execution phases thus:
In the match phase,we can:
1. match in parallel several partitions of the rule set.
2.match several partitions of the database.
In the execution phase,we can:
1.execute several rule actions on the database elements if these are independent(Inter-rule actions)
2. execute several instantiations of the same rule simultaneously. (Intra-rule actions).

In order to use the production system as the computational basis for concurrent transactional programming we need to consider how to speed up the system by permitting multiple rule application concurrently. This would require the analysis of the rules as to how the rules (and hence the respective transactions) interfere with each other when they are applied. There are three ways in which the rules can interfere: [13], [14], [15], [16], [17].

1. **Enabling dependence (ED)**: Rule i and rule j are called enable dependent if the application(or firing) of Rule i updates (W) the elements of the database , and creates the required precondition that is read (R) by Rule j and causes it to fire. As a special case, the update can be either insertion (W+) or deletion (W-) of elements and the precondition of rule j to fire is respectively the presence (R+) or absence (R-) of those identical elements. (In parallel programming these WR, W+R+, W-R- type of dependences are called dataflow dependence).
2.**Inhibit dependence (ID)**: Rule 1 and rule 2 are called inhibit dependent if the application(or firing) of Rule 1 updates (W) the elements of the database , and disables the required precondition that is read (R) by Rule 2 and prevents it from firing. As a particular case, the updates can be either insertion (W+) or deletion (W-) of elements and the precondition of rule 2 to fire can be respectively the absence (R-) or presence (R+) of those elements. The WR ,W+R- or W- R+ type of dependences are called inhibit dependences.
3. **Opposition dependence (OD)**: Rule 1 and rule 2 are opposition dependent if the firing of rule 1 updates (W) or deletes(W-) or adds(W+) elements while rule 2 respectively overwrites (W) or simply adds(W+) or deletes(W-) the same elements thereby interfering (In parallel programming this WW type of dependence is called data-output dependence).

The rules are called compatible , if they are not Inhibit dependent (ID) and not Opposition dependent (OD). The rules communicate either through Enable dependence (ED) or Inhibit dependence (ID).

Remark: In conventional database transaction processing, we only consider updates and the conflicts considered are WR, RW or WW conflicts. In production system we have no RW conflicts (known as "data anti-dependence" in parallel programming) because read or match phaseR happens only after execute phase W, except at the starting step, where we assume a dummy write action on a consistent initial database . Also we introduce special cases of update such as insert and delete actions , to maximize concurrency. Such a production system suits very well the needs of a distributed programming paradigm based on multisets [11].

We can relate the parallelism in production rules with vector, pipeline and data parallelism thus:

1.Vector parallelism:If all the rules are compatible and not communicative
then we can apply all the rules simultaneously This is similar to vector parallelism.

2.Pipeline parallelism: Here multiple rules are fired in parallel and passing data in a pipeline fashion.

3. Data parallelism: Multiple instantiations of the same rule are fired in parallel based on distinct data.

For practical programming problems it is better to choose a commutative production system since other systems have not yet been fully understood from the mathematical logical viewpoint. If we restrict the production system so that ID is not possible then we can construct a commutative system. One way to achieve this is by choosing the positive world of facts in T so that the LHS never refers to negated conditions. This is called the closed world assumption.The Gamma paradigm [18] and its extension [11] are based on the **closed world assumption** and corresponds to a commutative production system in which we do not check for absence of elements.

Remark : The production rule systems can be extended to event driven systems [17]. The event driven production rule systems have the syntax:
ON event
IF (precondition)
DO action
In this case input events are R actions and the output events are W actions. Such an event driven production system is useful for mapping our Transactional Production system (TPS) to distributed systems using communication primitives. We will consider this issue in a forthcoming paper.

3.2. Implementing production systems

There are two methods to map production rule execution on transactions:
(Geppert et al [6] describes the rule-based implementation of the transaction model using brokers to realise subsystems called Common-object oriented request broker architecture (CORBA)).

Method 1: A series of valid rule firings is treated as a transaction :
This approach leads to a successive modification of elements in a database and is suitable only for sequential computation since it does not permit concurrency, cooperation or competition among the rules .

Method 2: Every rule that fires is identified as a transaction:
This method is more appealing, since we can find a direct correspondence between concurrency in transaction processing systems with concurrently enabled rule systems. Therefore much of the known results in concurrency control in transaction processing can be directly applied.That is we can identify inter-rule and intra-rule consistency with inter - transaction and intra-transaction consistency. This aspect is useful for multiagent production systems in distributed computing. In using Method 2 we can use the locks described in Section 3.3 for concurrency control.

When the transactional paradigm is implemented using the multiple rule production systems the serializability of transactions guarantees correctness. However, note the difference between the ordinary database management system(DBMS) transactions and the paradigm we are using. The ordinary DBMS transactions come from the outside world, whereas here in our Transactional Production system (TPS) model, the instantiations are autonomous- that is being derived entirely from the rules and the current database state. Here, we check for internal consistency of both the rules and the database, while in the conventional DBMS transactions ,we are only concerned with the DBMS consistency with respect to its internal design specification. Hence, the act-verification strategy is embedded in this paradigm. In other words,while in the conventional DBMS situation we simply abort a transaction if it leads to inconsistency, in the TPS model we may modify or refine the rules to make them consistent so that eventually the transactions do not get aborted and produce the desired results by proper termination and confluence. The application of TPS to creative algorithm development strategy and its similarity to proofs-as-programs paradigm [9] can be understood from this aspect.

3.3 Compatibility locks

When rules are identified as transactions operating on a database it is necessary to introduce locks to avoid conflicts and avoid incompatibility between different transactions trying to manipulate the same set of elements or an intersecting set of elements so as to produce a serializable schedule. This can be achieved using locks,timestamps or certifiers [4] . If locks are used and shared memory of data items are used then the locks are to be compatible.

For database locking the locks are compatible only for R-R actions but not for R-W, W-R or W-W actions of two transactions for the same dataitem. In the case of rule systems we split the W actions into two basic actions: add (W+) or delete (W-) objects, and split the R actions into two basic actions: namely referencing Positive(presence) precondition (R+) or negative (absence) precondition (R-), to permit

higher concurrency. This results in a new lock compatibility matrix that is an enlarged version of the classical database lock compatibility matrix. This compatibility matrix is got from the compatibility conditions of rules so that they are neither OD nor ID. This matrix tells us the conditions under which the conflict resolution can be eliminated for some contexts to allow compatible rules ('yes' entries) to execute concurrently. Note that the six 'no' entries in this matrix reflects logical conflicts that can affect the serializability. For the ten entries 'yes' , the order of execution of the events is unimportant from the point of view of serialization of transactions. This is because successive addition of facts or deletion of facts and their eventual reading will not affect the consistency for these cases: (Note, however,that in a practical situation, hardware limitations may restrict these simultaneous actions).
For two different rules i and j the compatibility matrix for the above four actions is given below:

		j			
		W+	W-	R+	R-
i					
W+		yes	no	yes	no
W-		no	yes	no	yes
R+		yes	no	yes	yes
R-		no	yes	yes	yes

LOCK COMPATIBILITY MATRIX

Remark: If R+,R- are destructive as in Gamma [11], [18],the above compatibility matrix is inapplicable and we need exclusive locks to restrict the concurrency among the actions.

4. Derivation of an executable DATRAP

The specification is the key feature to abstract program construction [19]. Its precondition (or a priori condition) describes the initial states; its postcondition (or a posteriori condition) describes the final states. A specification is thus a condition and a post condition. We need to introduce the suitable actions in order to bring the final state to satisfy the postcondition through a set of transitions .

To systematically derive an executable DATRAP we use the following rules:
1. Transform the specification into an invariant (An invariant for a set of successively enabled rules is a logical formula that is true initially and is true when every enabled rule fires sequentially or in parallel) and a termination condition (where no two rules have any more ED and the invariant is still true).
2.Derive a pre- condition of a rule as a negation of the termination condition so that the precondition exhibits the desired local property that can be checked as a local operation.
3.Devise the actions to modify the database in such a way that the termination conditions can be locally validated ,while maintaining the invariant.
4. Ensure the confluence and termination of the rules ; that is the associated transactions commit and are serializable. Find the right topological sort that can provide best performance using a critical path analysis. (The application of the above rules produce the desired effect of ensuring that the union of all preconditions is logically equivalent to the negation of the termination condition and the union of all actions is equivalent to the termination condition and each action maintains the invariant in every rule.)

To choose granularity and levels of parallelism we need to split the precondition, action and post condition into a sequence of events. This is called refinement. In a refinement, a specification is improved by strengthening its postcondition so that the new post condition implies the old and weakening the precondition so that the old precondition implies the new. The refinement enables us to choose simpler actions. The stepwise refinement in DATRAP is carried out thus:
1.Use a succession of transitions that can be executed by choosing several actions and move through a succession of states, where each state satisfies a priori and a posteriori conditions.
2. Group the rules into subsets of rules called "context".
3. Subdivide each context into subcontexts until the interaction between the rules in the same context is easy to understand.
4.Limit the interaction between contexts. Obviously for concurrent execution of two or more contexts, they must be compatible.

5. Semantics of DATRAP

The rule in DATRAP can be thought of as a pair (Q,M) where the query Q selects the objects from an active database that have to execute a specified method-call M . In other words, the DATRAP constructs procedures with preconditions and hence is best suited for realising all types of parallelism-agenda, specialist and result parallelism.

The DATRAP can be defined by the syntax:
Rule rule_name= (Query , method_call).
We can associate two types of execution semantics with DATRAP; these are called "set-based semantics" and "instance-based- semantics". In set-based semantics all the conflict-free objects are processed at a time, whereas in instance based semantics, one qualifying object is processed at a time. The choice between these two semantics depends on the underlying architecture and applications. Due to lack of space we will not consider these aspects any further.

The implementation of rule based-system requires that the application of the rules eventually terminate and are confluent .Termination for a rule set is guaranteed if rule processing always reaches a stable state in which no rules will be enabled through 'false' conditions. Therefore rule processing does not terminate iff rules provide new conditions to fire indefinitely. Confluence means that for each initial database state the execution order is immaterial [20]. Confluency of rule sets is thus similar to confluency of rewrite systems [2], [21] except that the rule set behaviour is affected by the underlying database state

and a rule is confluent if it is confluent on all resulting database states. To analyse termination we can construct a rule-dependency graph (RDG). Here each rule Ri is denoted by a node and a directed arc between two nodes i and j indicate that Ri enables Rj; that is actions of Ri create the right conditions for Rj to fire. If the RDG has no cycles then the system terminates.

To draw an edge between i and j we need to find those database states for which the dependency exists. This analysis involves the unification, substitution and renaming of bound variables. Baralis et al [22] provide the propagation rules for update, delete and insert actions when rule actions are relational algebraic operations. As in [13] we can use Lambda calculus to construct dependency graphs.

The right partial order of rule execution is determined from a topological sort or a linear extension of the RDG.This ensures confluence. In this topological sort two rules can be executed simultaneously iff they are not self - dependent and and are mutually independent. Independence implies that the order of execution of rules is immaterial. However, dependent rules need to be partially ordered to ensure serializability.

The complexity of a distributed program can be reduced by constructing minimally dependent rules or maximally independent rules .

6.Performance analysis of DATRAP

The performance of a DATRAP depends upon the choice of a suitable topological sort that minimizes a certain objective. The usual scheduling objectives depend on the completion times of the jobs in the schedule, which also depends on the availability of resources [23],[24].

For improved performance of a DATRAP we must consider the optimal scheduling problem of a set of transactions,{T1,T2,...,Tn} , on a set of processors {P1,P2,...,Pm} or agents where transactions are executed, and also a set of additional resources - such as registers/ cache, denoted by {R1,R2,..,Rs}, that are required during their execution. A topological sort provides only an abstract partial order among the different transactions that result in a serializable multiagent program. Since it is not unique, we need to choose that topological sort which is optimal and allocate transactions to processors and agents in such a way to minimize traffic and computation time [23].

7. Example

Consider the problem of finding a lowest cost path between any two vertices in a directed graph whose edges have a certain assigned positive costs.

The lowest cost path problem requires the entity set of vertices, the relationship set of ordered pairs of vertices (x,y) representing edges, and the attribute of cost c for each member of the relation set, denoted by (x,y,c). Given a graph G the program should give for each pair of vertices (x,y) the smallest sum of costs path from x to y.

The vertex from which the lowest cost paths to other vertices are required is called the root vertex r. Let c denote the cost between any two adjacent vertices x and y, and let s denote the sum of costs along the path from the root to y; we assume that c (and hence s) is positive.This information is described by the ordered 4-tuple(x,y,c,s); in general all the relationships among the different vertices are described by an appropriate 4-tuple format of the form:(vertex label, vertex label,cost, sum of costs from root).

The fourth member of the 4-tuple, namely the sum of costs from a specified root remains initially undefined and we set this to a large number *.We then use the production rules to modify these tuples or to remove them.

To find the lowest cost path to all vertices from the specified root r, we use the TPS for the tuple processing and let the 4-tuples interact; this interaction results in either the generation of modified 4-tuples or the removal of some 4-tuples of the representation .

Specification to find the shortest path:

Let C(i,j) be the cost of path (i,j). A better path is one for which

that can pass through some vertex k such that:

$$C(i,k) + C(k,j) < C(i,j)$$

That is our production rule is: If C(i,k) +C(k,j) < C(i,j) then

delete C(i,j) and set C(i,j) = C(i,k) +C(k,j).

The invariant is: if C(i,j) is the initial cost then all the costs are always less than or equal to C(i,j)

We refine this by using the rule : If C(i,k) < C(p,k) , delete C(p,k) and retain C(i,k). Thus the following three production rules result:

Rule 1: If there are tuples of the form (r,r,0,0) and (r,y,c,*), replace (r,y,c,*) by (r,y,c,c) and retain (r,r,0,0).

Rule 1 defines the sum of costs for vertices adjacent to the root, by deleting * and defining the values.

Rule 2: If there are tuples of the form (x,y,c1,s1) and (y,z,c2,s2), where s2 > s1+c2 then replace (y,z,c2,s2) by (y,z,c2,s1+c2); else do nothing.

Rule 2 states that if s2 > s1+c2 we can find a lower cost path to z through y.

Rule 3:If there are tuples of the form (x,y,c1,s1) and (z,y,c2,s2) and if s1< s2 , then remove (z,y,c2,s2) from the tuple set; else do nothing.

Rule 3 states that for a given vertex y which has two paths-one from x and another from z, we can eliminate that 4-tuple that has a higher sum of costs from the root.

The above three rules provide for nondeterministic computation by many agents and we are left with those tuples that describe precisely the lowest cost path from the root. Note that the application of these rules correspond to binding constants to variables. In fact a variable shared across two or more patterns indicates the equivalent of a JOIN operation in databases- that is finding two different tuples that share a value across certain attributes.

This process of substitution that make a well-formed formula match by assigning values to variables x and y is called unification in logic-based computing. Also the unification and reduction constitute important ingredients in distributed computing through several agents. They must lead to termination after a finitely many steps and different reduction order should lead to the same object (uniqueness).

Consider the directed graph, in which the edge costs are as shown below ; we denote the graph by the triplet (x,y,c):

(1,2,50);(1,3,10);(1,5,45);(2,3,15);(2,5,10); (3,4,15); (4,2,20); (4,5,35); (5,4,30); (6,4,3).

We encode the graph by choosing the vertex 1 as the root; the format for representing the graph is given by : (vertex label, vertex label, cost, sum of costs from root).

(1,1,0,0);(1,2,50,*);(1,3,10,*);(1,5,45,*); (2,3,15,*); (2,5,10,*);(3,4,15,*);(4,2,20,*); (4,5,35,*); (5,4,30,*); (6,4,3,*).

We then apply the three rules nondeterministically. This results in the following tuples that describe the lowest cost path subgraph .(1,1,0,0); (1,3,10,10) ;(1,5,45,45); (3,4,15,25) ;(4,2,20,45). Note that the 4-tuple (6,4,3,*) gets eliminated as vertex 6 cannot be reached from the root vertex 1.

8. Distributed agent programming

The production rule approach can be easily modified to carry out distributed programming using messages (a distributed agent computation).

A distributed agent computation is a distributed computation with the following features:

1. Each agent can be active or inactive
2. An active agent can do local computation , send and receive messages and can spontaneously become inactive.
3. An inactive agent becomes active if and only if it receives a message.
4. Initially all agents are inactive except for a specified one which initiates the computation.

The production rules described in Section 7 can be modified to perform distributed agent computation.

In order to implement a production system in a distributed agent system, we must distribute the rules suitably so that rules are asynchronously fired. Since there is no global control , interference between different rules is to be prevented by using local synchronization among the different agents.

In particular we must be able to carry out the unification operation (or join) , namely the binding of constants to variables at different agents and carry out the required operations locally and transmit the information globally using neighbourhood agent communication. That is the unification is partitioned into fragments.

We must, however, remember that since shared memory is not used, the tuple messages are to be compared locally in each agent and the revised information is transmitted globally. Therefore the algorithm terminates or stabilizes when all the agents remain silent.

As an example, we consider the shortest path problem (Section 7). We assume that there are n agents having identical names as the nodes in the graph and each agent is connected to other agents in an isomorphic manner to the given graph. Such an assumption on the topology of the network simplifies the algorithm. Here, each agent knows the identity of its neighbours, the direction and cost of connection of the outgoing edge. Thus for the given directed graph the outdegree of each node is the number of sending channels and the indegree is the number of receiving channels.The revised production rules for the distributed agent computation are as follows:

a. Agent 1 (root) sends to all its neighbours x the tuple (1,x,c,c) describing the name of the root, and the distance of x from the root (c); all the neighbours of the root handshake, receive, and store it. This corresponds to seeding the reaction.

b. Each agent x sends its neighbour y at a distance c1 from it,the tuple (x,y,c1,c+c1) describing its name, its distance to y and the distance of y from the root through x using its distance to the root c.

c. Each agent y compares an earlier tuple (x,y,c1,s1) got from a neighbour x, or the root, with the new tuple (z,y,c1',s1') from another neighbour z. If s1< s1', then y retains (x,y,c1,s1) and remains silent; else it stores (z,y,c1',s1') and sends out the tuple (y,w,c2,s1'+c2) to its neighbour w at a distance c2, advising w to revise its distance from the root.

d. An agent does not send messages if it receives messages from other process that tells a higher value for its distance from the root and ignores the message. Thus it contains only the lowest distance from the root. All the agents halt when no more messages are in circulation and the system stabilizes. It is possible to devise an algorithm to detect termination. This will be described elsewhere.

The above approach is analogous to distributed transaction processing [4],[25]. Since the order of arrival of different tuples in the agent nodes is nondeterministic, we need to prove that the computation terminates and reduces to a unique object. Note that the distributed computing amounts to choosing different orders of unification and reduction sequences at each node. That is at each node the final outcome would be the same irrespective of any intermediate decisions. This is called "serializability "in transaction processing [4]. In mathematical logical term this is essentially a "confluence property" that stabilizes the computation eventually. This is also related to the "Church-Rosser property" in a lattice [21].

9. Concluding remarks

In summary, the transactional approach is the key to develop a distributed agent programming paradigm, since it provides for :

1.A programming methodology free from control management. The transactional implementation of rules provides for high concurrency on a database of

active objects represented by high level data structures,
2. The application of locality principle in program construction; formal specification and refinenent calculus calculus can provide for the choice of appropriate granularity of transactions and the level of parallelism; also, using rule-dependency graphs, we can prove termination and confluence (that is preservation of semantics) and choose the right partial order of execution through a topological sort .
3. The choice of an appropriate topological sort provides for high degree of concurrency in program-processor mapping and implementation in distributed machines and improved performance.
4.The use of Linda and PVM to simulate distributed agent systems [26].

The transactional paradigm will have applications in multi-agent production system programming [27],[28]. A multiagent production system consists of agents that serve as processes, functions, relations or constraints depending upon the context. In a concurrent programming situation, a computation state consists of a group of agents and store they share. Agents may add pieces of information to the store through an operation called "telling" and also wait for receive information through an operation called "asking". The telling operation corresponds to W+ or W- while the asking operation corresponds to R+ or R- actions. Thus the transactional approach can provide a conceptual framework for the development of parallel and distributed relational, constrained and contextual programs using multiagent architectures.

References

[1] V.K. Murthy and E.V. Krishnamurthy, Automating Problem Solving using transactional paradigm, Proc. Intl. Conf. on AI & Expert Systems, 721-729, Gordon Breach Publ., USA, 1995.
[2]E.V.Krishnamurthy, Introductory Theory of Computer Science, Springer Verlag, New York,1984.
[3] E.Rich and K.Knight, Artificial Intelligence, McGraw Hill, New York,1991.
[4] E.V.Krishnamurthy and V.K.Murthy, Transaction Processing Systems, Prentice Hall, Sydney, 1991.
[5] A.J.Bonner and M. Kifer, Application of transaction logic to knowledge representation, Lecture Notes in Computer Science, Temporal Logic, Vol.827, pp.67-81, 1994.
[6] A. Geppert and K.R.Dittrich, Rule based implementation of transactional model specification, pp. 127-143, in Rules in Database Systems, Editors: N.Paton and M.H.Williams, Springer Verlag, New York, 1994.
[7] J-Y. Girard, Linear Logic: A Survey, pp.63-112, in Logic and Algebra of Specification, Edtors: F.L.Bauer and W.Brauer, Springer Verlag, New York, 1993.
[8] U.Montanari and F.Rossi, Concurrency and concurrent constraint programming, Lecture Notes In computer Science, Vol.910, pp.171-192, Springer Verlag, New York,1995.
[9] F.L.Bauer and W.Brauer, Logic and Algebra of Specification, Springer Verlag, New York, 1993.
[10] B. Jonsson, Compositional Specification and verification of distributed systems, ACM Trans. Programming languages and Systems, Vol.16, pp.259-303, 1994.
[11] V.K. Murthy and E.V. Krishnamurthy, Probabilistic Parallel Programming based on multiset transformation, Future Generation Computer Systems, Vol.11, pp.283-293,1995.
[12] K.M.Chandy and J.Misra, Parallel Program Design, Addison Wesley, New York,1990.
[13] J.G. Schmolze, Guaranteeing Serializable results in Synchronous parallel production systems, J. Parallel and distributed computing, Vol.13, 348-365 , 1991.
[14] S. Kuo and D. Moldovan, implementation of multiple rule firing production systems on hypercube, J. Parallel and distributed computing, Vol.13, 383-394, 1991.
[15] S.Kuo and D.Moldovan, The state of the art in parallel production systems, J. Parallel and distributed computing, Vol.15,1-26,1992
[16] T.Ishida, Parallel, Distributed and multiagent Production Systems,
Lecture Notes in Computer Science,Vol. 890, Springer Verlag, New York, 1991.
[17] N.W.Paton and M.H.Williams, Rules in database Systems, Springer Verlag, New York, 1994.
[18] J.-P.Banatre and D.L.Me'tayer, Programming by Multiset transformation, Comm. ACM, Vol.36, 98-111, 1993.
[19] C. Morgan, Programming from Specification, Prentice Hall, Englewood Cliffs, New York, 1994.
[20] L.van der Voort and S.A. Siebes, Enforcing confluence of rule execution: pp.195-207, in Rules in Database systems, Editors: N.W.Paton and M.H..Williams, Springer Verlag, New York, 1993.
[21]E.V.Krishnamurthy, Parallel Processing ,Addison Wesley, Reading, Mass. 1989.
[22] E. Baralis and S.Ceri, Better termination analysis for active databases, pp.164-179, in Rules in database Systems, Editors: N.W.Paton and M.H.Williams, Springer Verlag, New York,1994.
[23] V.K.Murthy and E.V.Krishnamurthy, Transactional Paradigm: Applications to Distributed Programming, IEEE .Conf. Alg.Arch, Brisbane, pp.554-558, 1995.
[24] I.Rival, Algorithms and Order, Kluwer Academic Publishers, London, 1989.
[25] V.K. Murthy and E.V. Krishnamurthy, Gamma programming paradigm and heterogeneous computing, Proc. Hawaii Intl. Conf. on System Sciences, HICSS29, 273-281, IEEE Press, 1996.
[26] A.Geist et al, PVM: Parallel Virtual Machine, MIT Press, 1994.
[27] T. Wittig, ARCHON: An architecture for multiagent systems, Ellis Horwood, New York,1992.
[28] M.Woolridge and N.R.Jennings, Agent theories, Architectures and Languages, A survey, Lecture Notes in Computer Science,Vol. 890, pp.1-39,1995.