An Executor for Multidatabase Transactions which Achieves Maximal Parallelism

E. Jane Cameron, Linda A. Ness, Amit P. Sheth
Bellcore
445 South St., Morristown, NJ 07962
{*cameron, linda*}*@bellcore.com*, *amit@ctt.bellcore.com*

1. Introduction

The traditional transaction model and execution mechanisms are inadequate to provide a high level model for specification of more complex tasks and is inappropriate for environments containing heterogeneous and autonomous systems. Extending the traditional transaction concept to permit mutually dependent subtransactions within a single transaction would allow some complex functions to be viewed as single transactions on a heterogeneous and autonomous system. Such an extension requires a model that permits subtransactions to be scheduled based on their dependencies. To achieve better performance, the model should allow the expression of the maximal parallism by simultaneous execution of subtransactions as permitted by the dependencies and the constraints of the particular execution environment.

In this paper, we concisely specify a *executor* in the logically parallel language, L.0, for a particular extended transaction model called *multidatabase transactions*. An executor is a program, which takes as input the data defining any particular multidatabase transaction, dynamically schedules the sub-transactions, and determines whether the multidatabase transaction succeeded or failed. In the program presented here, the parallelism is the maximal amount permitted by the semantics of the multidatabase transaction. A refinement of this program could restrict the parallelism to the amount permitted by the systems executing the sub-transactions.

An extended Petri Nets based mechanism for scheduling the sub-transactions of a multidatabase transaction has been proposed [ELLR90]. Using L.0 has several advantages to that approach. The main one is the underlying execution model of L.0 provides a closer fit to the semantics of multidatabase transactions. The basic idea underlying L.0 is *synchronous*[1] *execution of quantified guarded predicates*. The synchronous execution permits maximal parallelism to be modeled. The parallelism may further be restricted according to the dependency constraints and the limitations of the execution environment. Traditionally, Petri Net semantics are based on interleaving semantics. At each step, only one of the enabled transitions may be chosen for firing. Another important advantage of our approach is that quantification in L.0 permits the definition of the executor, because it permits the predicates, defining the preconditions for execution, success, and failure to be represented as data. No such executor may be defined easily using extended Petri Nets because the algorithm for specifying the extended Petri Net for each multidatabase transaction cannot be defined easily using the extended Petri Net paradigm. Finally, L.0 is implemented, thus the executor can be implemented for controlling transaction execution by interfacing L.0 to the member DBMSs.

2. Multidatabase Transactions

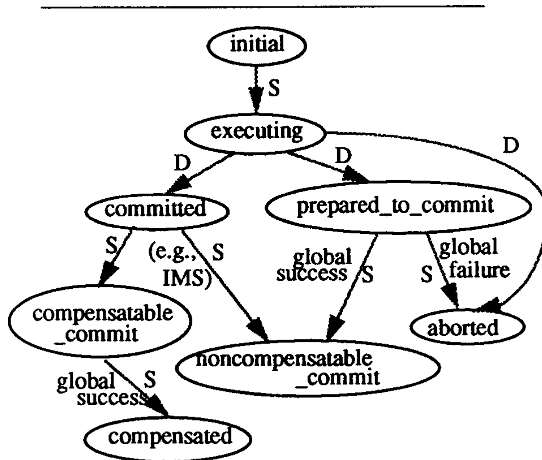
There have been several proposals to extend the *ACID* transactions. Gray [G81] proposed defining *compensating* transactions for subtransactions that can be used to undo their effects after commitments if required by the global transaction. *Sagas* [GS87] further this idea. Execution of Sagas need not be serializable. Other extensions include the concepts of conditional execution of subtransactions and functional equivalence of subtransactions [LER89] and the concept of *polytransaction* [RS91] which is a collection of related transactions that can maintain consistency of related (interdependent) data stored in multiple databases. In this paper, we use an extented transaction model called *multidatabase* or *flexible transactions* proposed in [RELL90] [ELLR90]. Multidatabase transactions make the requirements of isolation and

[1] logically clocked as in synchronous hardware circuits

TH0372-3/91/0000/0288$01.00 © 1991 IEEE

atomicity optional and are well suited for specifying complex tasks and for environments with autonomous member systems.

A *multidatabase transaction* consists of a set of subtransactions, each of which has a *precondition* for execution, together with a condition for success for the multidatabase transaction. Both the preconditions and the success conditions are specified using a conjunction of a boolean predicate on the states of the subtransactions and a simple type of temporal predicate. In [LMS91] the preconditions on the execution states of the subtransactions may involve input and output values, in addition to the execution states. The figure below shows a state diagram for the execution of a subtransaction.



S=> state transaition determined by the executor
D => state transition determined by the member system (e.g., DBMS)

There are only nine legal state transitions. *Global success* denotes the success of the multidatabase transaction, *global failure* denotes the failure of the multidatabase transaction. Six transitions are determined by the program executing the multidatabase transaction, and are conditional on the truth of a predicate. Each of the other three is determined by the member system (typically a DBMS) which executes the subtransaction.

The class of predicates used for the preconditions for execution, success, and failure is rich enough to express two types of dependencies: *positive execution dependencies* and *negative execution dependencies.* Positive execution dependencies are used to specify executions that are dependent on other subtransactions having entered the *prepared_to_commit*, *committed*, *compensatable_commit*, or *non_compensatable_commit* state. Negative execution dependencies are used to specify executions that are dependent on other subtransactions having entered the *aborted* state. The predicates can also express preferences among functionally equivalent transactions. The temporal predicates express constraints on the times that a subtransaction may be executed. Furthermore, when used as part of the success predicate, they can either force commitment of all subtransactions by a prescribed date or force failure of the entire subtransaction.

It is interesting to note that if any noncompensatable subtransaction enters the committed state, the entire multidatabase transaction must succeed (eventually). Since not every DBMS has a prepared to commit state, not every system will be able to make the *executing -> prepared_to_commit* transition.

Each of the transitions, in the above group of three, depends on the state of the database system executing the transaction. This state is not visible to the program executing the multidatabase transaction. The program executing the multidatabase transaction only needs to know about the state its subtransactions, i.e. it requires no other information about the database system. This fact is the key to the *executor for multidatabase transactions*.

3. Specifying an Executor for Multidatabase Transactions in L.0

An executor for multidatabase transactions can be easily specified in the logically parallel language L.0. This is a program which takes as input the data defining a multidatabase transaction, dynamically schedules the component transactions as permitted by the preconditions, and dynamically detects whether the precondition for success or failure has occurred.

The data defining a multidatabase transaction consists of data representing its success predicate, the names of its subtransactions and data representing the predicate which is the precondition for each of them. No extra data is needed to define its failure predicate, since it is assumed to be:

< ¬ success predicate> & <no subtransaction is executing> & <no subtransaction's precondition for execution is true>

3.1. A Brief Introduction to L.0

The basic model underlying the language L.0 [CCNS90] [N90a] is *synchronous execution of quantified guarded predicates* [N90b]. The simplest L.0 program consists of a set of guarded predicates. The guards are sometimes called *causes* and the predicates being guarded are called *effects*. The set of guarded predicates may be described using universal quantification. Each execution step consists of testing *all* of the guards, and then updating the values of the variables to force the truth of the predicate defined by the conjunction of all of the *effects* whose causes are true. If this is not possible, the execution halts. L.0 may be viewed as a logically parallel language because *all* of the effects whose guards are satisfied are made true simultaneously.

The initial state space may be specified as a (minimal) solution to a conjunction of predicates. The predicates permitted in L.0 include assignment (as usual), existence and non-existence (somewhat similar to logic programming), and equality. In addition, cause predicates may be defined by universal or existential quantification of other simpler cause predicates.

Since specifying control in such simple L.0 programs requires the introduction of many context "variables", several other control constructs have been added. The primary one of interest here is the weak *until* operator of temporal logic. The "control" for an executor may be specified concisely using *until*, as the program on the next page shows.

In this program, there is one instantiation of the group of guarded predicates for each subtransaction. Each instantiation contains six guarded predicates: the first three are preceded by the keyword *whenever*; the last three are preceded by the keyword *until.* In each instantiation, the first three can be regarded as the first argument to the *until* operator, and the last three can be regarded as determining the second argument to the *until* operator. In each instantiation, at each step, up to the first time one (or more) of the guards of the *until* guarded predicates is true, the semantics of the instantiated group is the standard semantics, i.e., all of the the guards are checked, etc. At this time an instantiated group behaves as if only the *until* guarded predicates existed. After this the group behaves as if it consisted of no guarded predicates, because the obligations imposed by the *until* construct have been met.

```
Flex_execute (data)
{{forall ?trans st {exists(data:sub_trans:?trans);};
 {whenever
  <execution_precondition(?trans)> &
   data:sub_trans:?trans:state = "init"
  then
   {data:sub_trans:state = "executing";
   include execute(?trans; write data:sub_trans:state);};

  whenever
   data:sub_trans:?trans:state = "committed"
   & data:sub_trans:?trans:type = "compensatable"
  then
   {data:sub_trans:state = "compensatable_commit"; };

  whenever
   data:sub_trans:?trans:state = "committed"
   & data:sub_trans:?trans:type = "non_compensatable"
  then
   {data:sub_trans:state = "non_compensatable_commit"; };

  until
   data:sub_trans:?trans:state = "prepared_to_commit"
   & <success_precondition>
  then    data:sub_trans:?trans:state = "committed";

  until (data:sub_trans:?trans:state = "prepared_to_commit" |
       data:sub_trans:?trans:state = "compensatable_commit")
       & <failure_precondition>
  then  data:sub_trans:?trans:state = "aborted";

  until data:sub_trans:?trans:state = "compensatable_commit"
          & <failure_precondition>
  then  {data:sub_trans:?trans:state = "compensated";
          include compensate(?trans); }; }; };

 maintain clock := read_external_clock();
};
```

We are assuming that an external clock exists, and that it is read at each logical program step by the function *read_external_clock. The maintain* ... causes the value of the "variable" clock to be set to this value at each step. This clock value is used in evaluating the temporal preconditions for execution of subtransactions.

include execute(?trans; write sub_trans:?state); is a "procedure call" to an L.0 "procedure" which provides the interface to the database.

3.1.1. Remarks

The above program assumes that only predicates from a restricted class are used as preconditions for execution of subtransactions, and for success or failure of the entire multidatabase transaction. Here the class consists of only conjunctions of a temporal predicate and a state predicate. Any input representation of the predicates must be in either disjunctive or conjunctive normal form so that the quantification operators of L.0 can be used to reconstruct them. In this case we assumed disjunctive normal form and nested the operators appropriately.

The following is L.0 code for reconstructing the success predicate assuming disjunctive normal form:

```
(forsome ?disjunct st {exists(success:temporal:?disjunct);}
 {forall ?beg ?end st
        { exists(success:temporal:?disjunct:?beg:?end);}
                {(?beg < clock < ?end);};  };) &
(forsome ?disjunct st {exists(success:state:?disjunct);}
 {forall ?i ?j st {exists(success:state:?disjunct:?i:?j);};
        {sub_trans:?i:state = ?j;}; };)
```

This program achieves the maximal parallelism permitted by the "optimistic" semantics of multidatabase transactions.

4. Summary and Future Work

We discuss a possible implementation of a controller that controls the execution of the subtransactions of a multidatabase transaction. Our approach supports maximal parallelism allowed in the model which is only limited by the transaction specification. The language constructs allows more natural specification of the multidatabase transaction. The scheduling of multiple multidatabase transactions is automatically supported in L.0 provided a suitable conflict resolution scheme is supported.

We are refining the executor to express the constraints imposed by a particular heterogeneous environment. Inherent parallelism and constraint mechanisms supported by L.0 should allow easy experimentation with conflict resolution schemes and safety (e.g., deadlock resolution) properties. We are investigating linking of L.0 to DOL [ROEL90] so that multidatabase transactions can be executed on real databases.

Acknowledgements: The idea of developing this executor originated during our discussions with Prof. Marek Rusinkiewicz. His help in understanding multidatabase transactions and in developing the concepts presented here is gratefully acknowledged.

References

[CCNS90] E.J. Cameron, D.M. Cohen, L.A. Ness, and H.N. Srinidhi, "L.0: A Language for Modeling and Prototyping Communications Software," *Third Intl. Conf. on Formal Description Techniques*, Madrid, November 1990.

[ELLR90] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A Multidatabase Transaction Model for InterBase, *VLDB*, Brisbane, Australia, 1990.

[G81] J. Gray, "The Transaction Concepts: Virtues and Limitations," *VLDB*, 1981.

[GS87] H. Garcia-Molina and K. Salem, "Sagas," *SIGMOD*, May 1987.

[LER89] Y. Leu, A. Elmagarmid and M. Rusinkiewicz, "An Extended Transaction Model for Multidatabase Systems," *Technical Report CSD-TR-925*, Computer Sciences Department, Purdue University, 1989.

[LMS91] K. Lee, W. Mansfield and A. Sheth, "An Interactive Transaction Model for Distributed Cooperative Tasks," *IEEE Data Engineering Bulletin*, 14, 1, March 1991.

[N90a] L. Ness, "Issues Arising in the Analysis of L.0: A Synchronous Executable Temporal Logic Language", *Proc. of the Workshop on Computer-Aided Verification*, (DIMACS Technical Report 90-31), June 1990.

[N90b] L. Ness, "A Quantified Synchronous Model of Computation which Permits Checking of a Class of Extended Temporal Logic Assertions," *Bellcore Technical Memorandum TM-ARH-017230*, 1990.

[RELL90] M. Rusinkiewicz, A. Elmagarmid, Y. Leu, and W. Litwin, "Extending the Transaction Model to Capture More Meaning," *Sigmod Record*, 19 (1), March 1990.

[ROEL90] M. Rusinkiewicz, S. Ostermann, A. Elmagarmid and K. Loa, "The Distributed Operation Language for Specifying Multi-System Applications", *Proc. of the First Intl. Conf. on System Integration*, April 1990.

[RS91] M. Rusinkiewicz and A. Sheth, "Polytransactions for Managing Interdependent Data," *Data Engineering Bulletin*, Vol. 14, no. 1, March 1991.