1990

Specification and Execution of Transactions for Advanced Database Applications

Y. Leu

Ahmed K. Elmagarmid
Purdue University, ake@cs.purdue.edu

N. Boudriga

Report Number:
90-1030

SPECIFICATION AND EXECUTION OF TRANSACTIONS FOR ADVANCED DATABASE APPLICATIONS

Y. Leu
A. Elmagarmid
N. Boudriga

CSD-TR-1030
October 1990
(Revised December 1990)

Specification and Execution of Transactions for Advanced Database Applications*

Y. Leu, A. Elmagarmid and N. Boudriga†
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

Autonomous multidatabases, Computer Aided Design (CAD) databases, and Object Oriented Databases have requirements that constitute a few examples of systems where traditional transactions may not be sufficient. Users of these advanced applications are more sophisticated than users envisioned for On Line Transaction Processing (OLTP) applications a few decades ago. The need to relax the properties of complex database transactions is urgent. Frameworks for flexible transaction systems are needed. This paper is a step towards this objective.

This paper addresses the formalization of a new transaction model called Flex along with execution control and analysis protocols. The algorithm is formalized through the use of Predicate Transition Nets (PTN) and reachability trees.

*This research was supported, in part, by a grant from the Software Engineering Research Center at Purdue University, a National Science Foundation Industry/University Cooperative Research Center (NSF Grant No. ECD-8913133).

†University of Tunis, Faculty of Science, Tunisia.

1 Introduction

A transaction constitutes a unit of work in a database system. Systems that use transactions guarantee their basic properties. Atomicity, consistency, isolation and durability (ACIDity) are basic properties of transaction systems. Most DBMSs strive to guarantee these properties through the use of concurrency, atomicity, and recovery algorithms. Transactions have been a successful technology to build meaningful and extensive applications over the last few decades.

With the wide spread use of DBMSs in advanced applications, the suitability of these transaction systems has come under question. It has lately been argued that while it is proper for the system to guarantee the ACIDity properties, it should be up to the application to decide which of these properties they need to be enforced and which they can trade for more flexibility or higher performance.

This paper relaxes two of these properties, namely, atomicity and isolation. While the paper is written in the context of the InterBase project[1], the model is formulated and is intended for general use.

The new model outlines three goals: function replication, dependencies (both external and internal) and compensatability. Alternative ways by which a specific task can be performed can be stated in the new model. Traditionally, all the tasks stated as a part of transaction must be performed. Using alternatives more than one equivalent task can be stated and it may be left up to the application designer to finally choose which one to commit. Dependencies are another extension provided in the model. The model allows for specifying functions that can be used to influence transaction execution. These functions are considered external parameters to the transaction or subtransaction. Time and other cost functions are given in this model. The model also allows for specifying dependencies among the subtransactions of the same transaction. These are stated in terms of either positive or negative dependencies. Finally, transactions are traditionally non-compensatable and once they are committed their effects are preserved by the system. Sagas [GMS87], on the other hand, allow for transactions to be compensated by running other transactions that undo their effects. In this new model we allow a transaction to include some subtransactions that are compensatable and others which are not. This results in mixed transactions. The mixing of both compensatable and non-compensatable subtransactions complicates the management of these transactions. It also reduces the isolation to the subtransaction level. Those subtransactions which are non compensatable must run in isolation of the rest of the system, while the effects of the compensatable subtransactions can be visible to other global transactions before their composing global transactions commit [ELLR90].

This paper is organized as follows. In Section 2, we present our extensions to the traditional transaction model. In Section 3, a transaction model, called Flex, which includes all the proposed extensions is formally defined. In Section 4, we formulate a mapping from a Flex transaction to Predicate Transition Nets which serves as a basis for execution control and analysis of Flex transactions. In Section 5, we present an execution control algorithm

[1]InterBase is a project in the Indiana Center for Database Systems that studies issues of transaction management and consistency in the multidatabase area. The InterBase prototype has been built and it currently includes Sybase, Ingres, Guru, Dbase IV, and Oracle. In addition it also integrates various other non-database packages.

for Flex transactions. We also present a method for analyzing the correctness of the Flex transactions. In Section 6, we summarize this paper by comparing our work to other related work and outlining our on-going research.

2 Extending Transaction Models

2.1 Alternatives

Multidatabase users frequently find themselves with functionally equivalent alternatives to reach their objectives. A powerful transaction model must allow the user to express the various choices by which his request can be implemented. The presence of alternate ways of achieving a particular objective represent a state of non-determinancy.

An example of these alternatives can be found in a travel agent scenario.

Example 1:

Consider a travel agent (TA) information system[Gra81]; a transaction in this system may consist of the following tasks:

1. TA negotiates with airlines for flight tickets.
2. TA negotiates with car rental companies for car reservations.
3. TA negotiates with hotels to reserve rooms.

Let us assume, now, that for the purpose of this travel, two airline companies (Northwest and United), one car rental company (Hertz) and three hotels (Hilton, Sheraton and Ramada) can be involved in this trip. The travel agent can implement these tasks as

1. Order-a-ticket from either Northwest or United airlines.
2. Rent-a-car from Hertz.
3. Reserve-a-room in any one of the three hotels.

These three tasks can be decomposed respectively as the three sets $\{t_1,t_2\}$, $\{t_3\}$ and $\{t_4, t_5, t_6\}$, where the t_i's are defined as follows:

t_1	Order a ticket at Northwest Airlines;
t_2	Order a ticket at United Airlines;
t_3	Rent a car at Hertz;
t_4	Reserve a room at Hilton;
t_5	Reserve a room at Sheraton;
t_6	Reserve a room at Ramada.

In the above example, we use the term task to name the specific function that we want to perform. For example, buying a ticket is a task. Two subtransactions which are used to implement the same task are called *functionally equivalent.* In Example 1, t_1 and t_2 are two functionally equivalent subtransactions for the order-a-ticket task. We say that a task is performed successfully if one of its functionally equivalent subtransactions executes successfully. A transaction is said to be successful if all its tasks are successfully performed.

2.2 Dependencies

Let us consider a set T of subtransactions, say $T = \{t_1, t_2, \cdots, t_n\}$. The execution of a subtransaction t_i can depend on the failure or the success of the execution of another subtransaction. Furthermore, it can be dependent on some external parameters (such as time). More precisely, we define:

Positive dependency: A subtransaction t_i is positively dependent on subtransaction t_j if t_i can be executed only after t_j is successfully executed.

Negative dependency: A subtransaction t_i is negatively dependent on subtransaction t_j if t_i can be executed only after t_j is executed and failed.

external dependency: Let X be a set of parameters (X is disjoint from T). A subtransaction t_i is externally dependent on X if the execution of t_i is dependent on the truth of a predicate on X.

In the previous example, replace t_2, t_5 and t_6 respectively by subtransactions t'_2, t'_5 and t'_6 which are defined as follows:

t'_2 Order a ticket at United Airlines, if t_1 fails;
t'_5 Reserve a room at Sheraton, between 8AM and 5PM;
t'_6 Reserve a room at Ramada, if t_1 succeeds.

One can see that, in the set $T = \{t_1, t'_2, t_3, t_4, t'_5, t'_6\}$, t'_2 is negatively dependent on t_1, t'_5 is time dependent and t'_6 is positively dependent on t_1.

Another example of external dependency is given by subtransactions that have values associated with them. These values can be cost or time related.

2.3 Compensatability

As has been stated in the previous section, transactions in the multidatabase environment can be long lived. It has been shown by Gray et. al. [Gra81] that problems arise when enforcing strict isolation in long lived applications.

Therefore, the isolation granularity of the global transaction should be reduced. Gray [Gra81] proposed to associate with each subtransaction a *compensating subtransaction* which can semantically "undo" the effects of a committed subtransaction, if required. This concept allows the global transaction to reveal (partial) results to other transactions before it commits. By doing so, the isolation granularity of the global transaction is reduced to the subtransaction level instead of the global transaction level. A global transaction consisting only of subtransactions which can be compensated is called a *saga* [GMS87]. However, in the real world, not all subtransactions can be compensated. For example, subtransactions that are accompanied by real actions are typically non-compensatable.

To address the fact that some of the subtransactions may be compensatable, we introduce in our model the concept of *typed transactions*. A global transaction is *typed* if

some of its subtransactions are compensatable and some are not. In a typed transaction, the subtransactions which are compensatable may be allowed to commit before the global transaction commits, while the commitment of the non-compensatable subtransactions must wait for a global decision. When a decision is reached to abort a typed transaction, the subtransactions in progress and the non-compensatable subtransactions waiting for a global decision are aborted, while the committed compensatable subtransactions are compensated. In this sense, typed transactions are different from *s-transactions* [EVT88] or *sagas* [GMS87] which allow only compensatable subtransactions.

Hence, typed transactions fill the spectrum from sagas, (assuming the compensability of all subtransactions) to traditional distributed transactions (assuming that subtransactions are non-compensatable). Typed transactions are more flexible because they allow compensatable and non-compensatable subtransactions to coexist within a single global transaction.

3 Formal Model

3.1 Form of Flex Transactions

In order to capture the notion of compensatability of subtransactions, we use the concept of type: a subtransaction is said to be of type C if it is compensatable, it is of type NC if it is non-compensatable.

A Flex transaction model is formally defined as follows:

Definition 1 *A* Flex transaction T *is a 5-tuple (B, S, F,* Π *, f) where*

- $B = \{t_1, t_2, \cdots, t_n\}$ *is a set of typed subtransactions called the domain of* T
- S *is a partial order on* B *called the success order of* T
- F *is a partial order on* B *called the failure order of* T
- Π *is a set of external predicates on* B
- f *is an n-ary boolean function defined on the set* $\{1, 0\}$ *and is called the* acceptability function *of* T

We illustrate the above definition using the example of travel agent transaction introduced in the previous section.

Example 2: Consider the travel agent transaction introduced in Example 1. In addition, we assume the following: (1) the ticket ordering subtransactions are non-compensatable; (2) ticket ordering subtransactions must run within business hours from 8am to 5pm and t_2 will be executed only after t_1 is executed and fails (3) the rent-a-car subtransaction must be executed after the order-a-ticket subtransaction and the reserve-a-room subtransaction must be under the budget of \$100; (4) the transaction is successful if ordering-ticket, rent-a-car and reserve-a-room are all successful. We propose the following Flex transaction to formalize the travel agent transaction.

$$\mathrm{B} = \{t_1(NC), t_2(NC), t_3(C), t_4(C), t_5(C), t_6(C)\}$$

$$\mathrm{S} = \{t_1 \prec t_3, t_2 \prec t_3\}$$

$$\mathrm{F} = \{t_1 \prec t_2\}$$

Π = { P, Q } where P and Q are two predicates defined by
P = { $8 < \mathrm{time}(t_1) < 17$, $8 < \mathrm{time}(t_2) < 17$ }
Q = { $\mathrm{cost}(t_4) < \$100$, $\mathrm{cost}(t_5) < \$100$, $\mathrm{cost}(t_6) < \$100$ }

$$\begin{aligned} f(x_1, x_2, x_3, x_4, x_5, x_6) = {} & (x_1 \wedge x_3 \wedge x_4) \vee \\ & (x_1 \wedge x_3 \wedge x_5) \vee \\ & (x_1 \wedge x_3 \wedge x_6) \vee \\ & (x_2 \wedge x_3 \wedge x_4) \vee \\ & (x_2 \wedge x_3 \wedge x_5) \vee \\ & (x_2 \wedge x_3 \wedge x_6) \end{aligned}$$

3.2 Execution States

Definition 2 *For a Flex transaction T with m subtransactions, the* transaction execution state x *is an m-tuple $(x_1, x_2, ..., x_m)$ where*

$$x_i = \begin{cases} N & \textit{if subtransaction } t_i \textit{ has not been submitted for execution;} \\ E & \textit{if } t_i \textit{ is currently being executed;} \\ S & \textit{if } t_i \textit{ has successfully completed;} \\ F & \textit{if } t_i \textit{ has failed or completed without acheiveing its objective;} \end{cases}$$

While successfully completed for a compensatable subtransaction means that the subtransaction is committed, successfully completed for non-compensatable subtransaction means that the subtransaction is in a *prepared state* [Gra78]. The transaction execution state is used to keep track of the state of subtransaction executions of a Flex transaction. The acceptability function appears as a partial function defined on the set of execution states. It is computable whenever all x_is occurring in its expression are equal to either S or F. Hence, the acceptability function reflects the acceptability of an execution state. Whence the following definition

Definition 3 *Let T be a Flex transaction and X the set of its execution states. The acceptable state set, A, of the Flex transaction is the subset*

$$A = \{\, x \in X \mid f(x) = 1\}$$

In Example 2, the set of acceptable states is defined by

$$
\begin{aligned}
A = \{ &(S, \_, S, S, \_, \_) \} \cup \\
&\{ (S, \_, S, \_, S, \_) \} \cup \\
&\{ (S, \_, S, \_, \_, S) \} \cup \\
&\{ (\_, S, S, \_, S, \_) \} \cup \\
&\{ (\_, S, S, S, \_, \_) \} \cup \\
&\{ (\_, S, S, \_, \_, S) \}
\end{aligned}
$$

Definition 4 *Let T be a Flex transaction and x be an execution state of T. T succeeds if x is an acceptable state.*

3.3 Execution of Flex Transaction

Let T= (B, S, F, Π, f) be a Flex transaction and t_i be an element in B. The set $Pdep(t_i)$ (resp. $Ndep(t_i)$) is the subset of B constituted by all elements t of B such that t_i is positive dependent (resp. negative dependent) on t. Let x = $(x_1, x_2, \cdots, x_n)$ be an execution state of transaction T. We say that the subtransaction t_i is executable at state x if the following four assertions are satisfied.

1. $x_i = N$;
2. For all j such that $t_j \in Pdep(t_i)$, $x_j = S$;
3. For all j such that $t_j \in Ndep(t_i)$, $x_j = F$;
4. For each external predicate P, P(t_i) is true.

Intuitively speaking, a subtransaction t_i is executable at a given execution state if it is not executed and all conditions on which the execution of t_i depends are satisfied:

We can now formulate the execution rules of a Flex transaction as follows:

procedure Exec-Flex-transaction(in:T, out:R)
begin1. initialize x := (N, N, $\cdots$, N), $R := \emptyset$ and
compute the set EXEC of all executable subtransactions;
2. **while** *$R = \emptyset$ and EXEC $\neq \emptyset$*
execute concurrently all elements of EXEC, put the response in x
(x_i = the execution state of t_i) and compute $f(x)$, if $f(x) = 1$ then $R = \{x\}$;
3. if $R \neq \emptyset$ then commit the Flex transaction else get feedback from the user to determine whether to commit or abort the Flex transaction.
end

According to the above execution rules, concurrent execution of subtransactions is allowed if they are executable at the same time. When the result of the execution is known, we modify the transaction execution state accordingly. After the completion of a subtransaction, we check whether or not an acceptable state has been reached. If an acceptable state has been reached, we commit the Flex transaction. When a Flex transaction terminates

without reaching an acceptable state (i.e. $EXEC = \emptyset$), then a feedback is required from the user to decide whether to commit (with partial results) or to abort the Flex transaction. The feedback mechanism allows the user to save (and therefore commit) the useful partial results when the execution of the Flex transaction is not perfect. This is useful when functions that are not achieved can be performed later when it is more convenient.

To commit a Flex transaction, we send a "commit" message to all non-compensatable subtransactions which are waiting in their "prepared to commit" states (the compensatable subtransactions may have been committed earlier). To abort a Flex transaction, send an "abort" message to all sites in which a subtransaction of the Flex transaction is waiting in its prepared state, and issue compensating subtransactions to the sites in which a compensatable subtransaction of the Flex transaction has been committed.

4 Representing Flex Transaction by Predicate Transition Nets

Having defined the Flex transaction model, one has to implement this model. Such an implementation must provide control and analysis tools, and has to support the properties of Flex transaction (such as alternative, compensability and dependency). Our approach, in this sense, is to model the (dynamic) Flex transaction in terms of Predicate Transition Nets (PTN) introduced in [GL81].

We begin this section by presenting the Predicate Transition Nets, and then construct a mapping between Flex transactions and PTNs.

4.1 On Predicate Transition Nets

We simplify the definition of the Predicate Transition Nets as follows. A predicate transition net consists of

1. A set H called the set of places;
2. A set K called the set of transitions;
3. A set L of arcs;
4. A mapping $\mathcal{K}$ of the set of transitions into the set of formulae of the first order logic; and
5. A marking M_0 of the places: it is a mapping that assigns to each place h in H a set of symbolic characters (called tokens).

To clarify this definition, we propose the following example.

Example 3: Figure 1 represents the Predicate Transition Nets (H,K, L, $\mathcal{K}$, M_0) given by

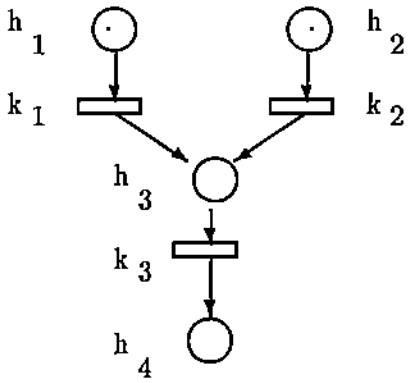

Figure 1: A Predicate Transition Net

$H = \{h_1, h_2, h_3, h_4\}$
$K = \{k_1, k_2, k_3\}$
$L = \{(h_1, k_1), (h_2, k_2), (h_3, k_3), (k_1, h_3), (k_2, h_3), (k_3, h_4)\}$
$\mathcal{K}$:
$k_1 \rightarrow (8 < time(k_1) < 17)$
$k_2 \rightarrow (8 < time(k_2) < 17) \wedge (x_1 = F)$
$k_3 \rightarrow true$
M_0 :
$h_1 \rightarrow \bullet$
$h_2 \rightarrow \bullet$

Where $\bullet$ is a given symbol.

4.2 Mapping the Flex transaction

Let T = (B, S, F, Π, f) be a given Flex transaction, we define the associated Predicate Transition Nets PTN(T)=(H, K, L, $\mathcal{K}$, M_0) as follows:

If B is the set $\{t_1, t_2, \cdots, t_n\}$, then:
H = $\{a_1, a_2, \cdots, a_n, b_1, b_2, \cdots, b_n\}$ where we assume that, for all i and j, the symbol a_i is equal to the symbol b_j iff t_i is positively dependent on t_j.
K = B
L = { (a_i, t_i) | i ≤ n } ∪ { (t_i, b_i) | i ≤ n }.
$\mathcal{K}$:

$$\mathcal{K}(t_i) = \left(\bigwedge_{t \in Ndep(t_i)} F(t, t_i) \right) \wedge \left(\bigwedge_{q \in \Pi} q(t_i) \right)$$

M_0 :
$a_i \rightarrow \bullet$, for each i such that t_i has no positive precedence

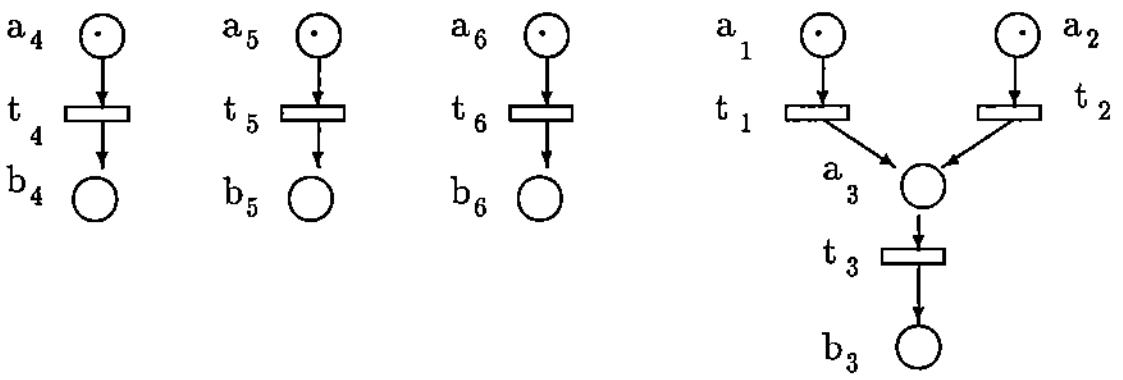

Figure 2: PTN for the travel agent transaction

Intuitively speaking, transitions of the associated PTN are the subtransactions of the Flex transaction linked with the condition that needs to be executed, places and arcs are physically present to maintain the connectivity. In the above mapping, $\mathcal{K}$ associates with each transition a set of predicates, which includes the negative dependence predicates $F(t, t_i)$ for any t in $Ndep(t_i)$ and the external predicates $q(t_i)$ for any q in Π.

Example 4: Consider the travel agent transaction defined in Example 2, the associated Predicate Transition Net is represented in Figure 2.

Looking for the properties of this mapping, the following can be observed:

- The construction of the associated PTN can be done automatically,
- The execution of the PTN models the execution of the Flex transaction,
- The number of places in the associated PTN is equal to the number of subtransactions plus the number of terminal subtransactions (i.e. with no subtransaction positively depending on it).

5 Execution and Analysis

In this section, we propose an algorithm for the execution control of Flex transactions using their Predicate Transition Net representation. We then give a method for performing the analysis of Flex transactions.

5.1 Execution Control of Flex Transactions

The execution of Flex transactions must satisfy the dependency relations and the external predicates. Because the associated Predicate Transition Net captures these dependency relations and gives tools to preserve them, our approach consists of executing the associated Predicate Transition Net.

The following algorithm implements, in a sequential manner, the procedure *Exec-Flex-Transaction* defined in Section 3.3, in the sense that the scheduling activity executes all executable subtransactions, stores its execution state, and then computes the acceptability function.

In procedure *evaluate_PTN*, $\mathcal{E}$ is the current enabled set; $\mathcal{U}$ is the current executable set derived from $\mathcal{E}$; and $\mathcal{G}$ is the scheduled set that contains the transitions corresponding to the submitted subtransactions. The algorithm starts from the initial execution state (with all state variables initialized to N), computes the enabled set $\mathcal{E}$, calculates $\mathcal{U}$ from $\mathcal{E}$, and submits all subtransactions whose corresponding transitions are in $\mathcal{U}$.

Whenever a subtransaction completes, a new executable set $\mathcal{U}$ is determined by the algorithm. The set $\mathcal{U}$ is partitioned into two sets, $\mathcal{G}$ and $\mathcal{G}^+$. The executable transitions which are not yet submitted are contained in $\mathcal{G}^+$, while transitions in $\mathcal{G}$ are the transitions that are already submitted.

To prevent the loss of responses from the local database systems, the responses are first buffered in queue Q. Whenever Q is not empty, the algorithm dequeues Q and computes the new execution state, fires the corresponding transition if the dequeueing response is a SUCCESS, and computes the executable set $\mathcal{U}$.

The scheduling activity is continued until the termination condition is met. When the execution terminates, if the final execution state x is acceptable (i.e. $f(x) = 1$) the Flex transaction is committed; otherwise, it is left to the user to decide whether to commit or to abort the Flex transaction depending on the current execution state. The feedback allows the user to commit the partial results of an imperfect execution.

To commit a Flex transaction, for each non-compensatable subtransaction t_i whose corresponding execution state variable x_i is S, send a "COMMIT" message to its local database system; for each non-compensatable subtransaction t_j (if any) in $\mathcal{G}$, send an "ABORT" message to its local database system; and then *compensate* each compensatable subtransaction in $\mathcal{G}$.

To abort a Flex transaction, each subtransaction t_i, whose corresponding execution state variable x_i is S, has to be aborted or compensated depending on its type.

5.2 Analysis of Flex Transactions

Before a Flex transaction is actually executed, we have to verify that the transaction will behave exactly as desired. We are especially interested in the following two aspects:

p1. Can each acceptable state be reached from the initial execution state?

p2. Which acceptable states can be reached when some specific failures occur?

To analyze these aspects of Flex transactions, we use the well known techniques of Petri Nets reachability. The reachability problem is perhaps the most basic Petri Net analysis problem. Deadlock and failure can be stated in term of the reachability problem.

In this section, we present the notion of reachability of an execution state in our Flex transaction model and show how to capture it when analyzing the associated Predicate Transition Net. For this purpose, we assume that we are given a Flex transaction T and its predicate transition net PTN. By pattern of failure x, we mean a state $(x_1, x_2, \cdots, x_n)$ where x_i is S if we assume that t_i successfully executes, and F otherwise. For the sake of

```
procedure evaluate_PTN(PTN, f, t0; R)
/* PTN - the Predicate Transition Net, t0 is the timeout value and f is the acceptability function*/
    Initialize timeout mechanism with timeout interval t0;
  begin
        x ← (N, N, N, ···, N) /*no subtransactions has been executed */
        on timeout flex_abort;
        E ← φ;     /* E – enabled set */
        U ← φ;     /* U – executable set */
        G ← φ;     /* G – scheduled set */
        Q ← empty;     /* Q – response Queue */
        compute_enabled_set E from PTN;/* enabled contains transitions which are enabled*/
        compute_executable_set U from the new enabled set E;
    repeat
        G+ ← U - G;
        /* tr represents both transition and subtransaction */
        For each transition tri ∈ G+ do
            begin
                submit subtransaction tri to the local database system;
                G ← G ∪ { tri};
                xi ← E /* E for executing state */
            end;
        on receiving response enqueue response in Q;
        while (Q = empty) do
            begin
                if (U = ∅) then
                    begin
                        flex_abort;
                        exit
                    end
            end;
        RESP← dequeue(Q);
       /*assume that RESP is from ti*/
        G ← G - { tri };
        if (RESP = SUCCESS)
        then
            begin
                xi ← S;
                fire(tri);
                compute_enabled_set E
            end
        else
            xi ← F
        endif;
        compute_executable_set U;
    until (check_terminate);
      if f(x) = 1 then flex_commit; R = x;
      else get feedback; if feedback is ABORT then flex_abort; else flex_commit;
  end.
```

Figure 3: The execution control algorithm

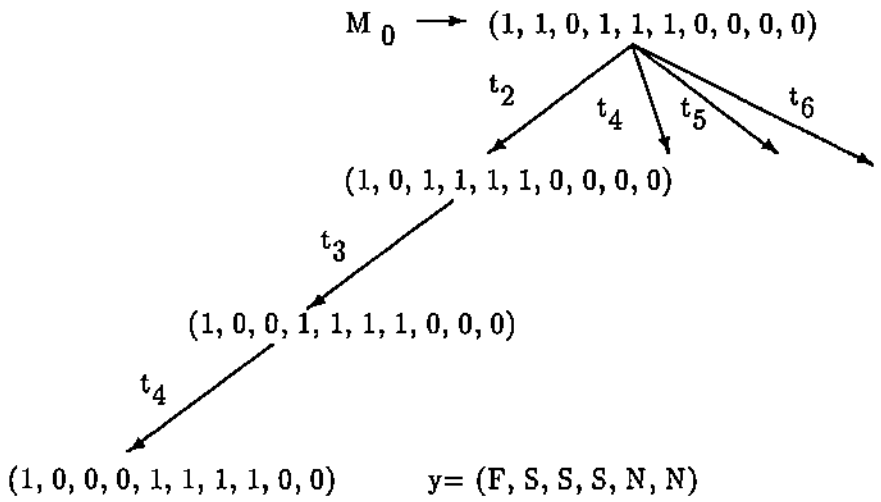

Figure 4: A reachable acceptable state

simplicity, we assume that the places of a PTN are ordered as $(a_1, a_2, \cdots, a_n)$. A marking of a PTN is a representation of the distribution of tokens in the places of the PTN. We denote a marking by $(\mu_1, \mu_2, \cdots, \mu_n)$ where μ_i is the number of tokens in place a_i. Firing a transition consists of deleting a token from each of its input places and adding one token to each of its output places.

We suppose in the following that a pattern of failure is fixed. The reachability tree *Rt* under the pattern of failure x is given as follows.

- Root of *Rt* is the initial marking of the associated PTN.
- A node, in *Rt*, is obtained from the initial marking by firing a sequence of transitions.
- An arc in *Rt* links a marking μ to marking μ' if μ' is obtained from μ by firing a transition in PTN.

Definition 5 *A reachable acceptable state y, under pattern of failure x, is an execution state such that $f(y) = 1$ and there is a sequence of transitions leading to it.*

To illustrate the notion of reachability, we consider the travel agent transaction defined in Example 3.

Example 5: Consider the PTN in Example 4. We show, in Figure 4, the reachability tree of the PTN with the pattern of failures (F, S, S, S, S, S) (i.e. t_1 will fails when it is executed). As shown in Figure 4, the state (F, S, S, S, N, N) can be reached by the ordered execution of t_2, t_3, t_4. Note that we order the marking in Figure 4 by $(a_1, a_2, a_3, a_4, a_5, a_6, b_3, b_4, b_5, b_6)$.

In performing the reachability tree analysis, we ignore the external dependency and fix the negative dependency by the use patterns of failure. This analysis allows us to predict

the behavior of a Flex transaction according to its specification, which may be useful when designing a complex Flex transaction.

6 Conclusions

6.1 Related Work

Several extensions to the initial transaction concept have recently been proposed. One significant extension is the *nested transaction* [Mos81]. A nested transaction allows a transaction to be dynamically decomposed into a hierarchy of subtransactions and ensures the transaction properties for individual subtransactions. The capability of decomposing a transaction into subtransactions, providing synchronized concurrent execution of subtransactions and encapsulating failures in the subtransactions makes the nested transactions suitable for composing transactions in distributed computing environments. However, the *atomicity* and *isolation* properties are still the same as in the classical transaction model.

Other extensions to the classical transaction model focus on solving the problem of long lived transactions. For example, a *CAD transaction model* is proposed [BKK85] to allow a group of cooperating designers to complete a design without having to wait. CAD transactions are further classified into five conceptual levels: project transactions, cooperating transactions, clients/subcontractors transactions, designer's transactions and short-duration transactions.

A transaction model for *software development environment*, (SDE) has been proposed to support the cooperation of a team of programmers in developing and maintaining software systems. Two salient features of this model are *split-transaction* operation and the *participation domain*. A *split-transaction* operation can be used to split a long lived transaction into two new transactions, one contains the completed part of the old transaction and the other one contains the ongoing part. The one containing the completed part can be committed and, therefore, releases useful results to other transactions. The *participation domain* concept allows a specific set of transactions to form a domain. Inside a domain, serializability is not required for the proper execution of transactions. As a result, transactions may view uncommitted updates from other transactions in the same domain. Transactions belong to different domains must be serialized.

Sagas are long lived transactions that consists of a sequence of relatively independent steps, where each step does not have to observe the same consistent database state. Each step is a subtransaction associated with a compensating transaction which can semantically "undo" the effects of this subtransaction [Gra81]. When a subtransaction completes, it can commit on its own without waiting for its parent transaction to commit, and therefore, reveals its partial results. When a failure occurs in the middle of an execution, a forward recovery by executing the missing subtransactions or a backward recovery by executing the compensating subtransactions for the committed subtransactions can be used. Sagas successfully reduce the isolation granularity to the subtransaction level. However, it may not be applicable to applications that consists of non-compensatable actions. In contrast, our approach can model sagas by letting all subtransactions be compensatable. while on the other extreme, our approach can model distributed transactions (the one proposed by Gligor) by letting all subtransactions be non-compensatable.

6.2 Summary

In this paper, an extension to traditional transactions is described and formalized. Flex transactions are described in the context of autonomous multidatabase systems. The Flex transaction model contains features which are esspecially useful for coping with problems caused by autonomy. A Flex transaction has alternate ways of committing. The external dependencies associated with subtransactions make the scheduling of Flex transactions more convenient. The Flex transaction model also supports mixing compensatable and non-compensatable subtransactions, thereby, reducing isolation granularity. The rationale for several of these extensions are described in the paper. The model has also been formalized, and several algorithms for controlling its execution has been proposed.

The InterBase prototype is currently being extended to support Flex transactions. We are also studying transaction management routines to support these new transactions in the context of both serializability [LE90] and quasi serializability [DE89]. Various other extensions are also now underway. Among others, we are working on using the logic programming paradigm to specify Flex transactions [KELB90], [LBEK], extending the notion of feedback described in this paper and relaxing the durability property of transactions.

References

[BKK85] F. Bancilhon, W. Kim, and H. Korth. A model of CAD transaction. In *Proceedings of the International Conference on Very Large Data Bases*, pages 25–33, 1985.

[DE89] W. Du and A. Elmagarmid. Quasi serializability: a correctness criterion for global concurrency control in InterBase. In *Proceedings of the International Conference on Very Large Data Bases*, Amsterdam, The Netherlands, August 1989.

[ELLR90] A. Elmagarmid, Y. Leu, W. Litwin, and M. E. Rusinkiewicz. A multidatabase transaction model for InterBase. In *Proceedings of 16th VLDB conference*, August 1990.

[EVT88] F. Eliassen, J. Veijalainen, and H. Tirri. Aspects of transaction modelling for interoperable information systems. In *Interim Report of the COST 11ter Project*, pages 39–55, 1988.

[GL81] H. J. Genrich and K. Lautenbach. System modeling with high level petri nets. *Theorical Computer Science*, 13:109–136, 1981.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM Conference on Management of Data*, pages 249–259, May 1987.

[Gra78] J. Gray. *Notes on database operating systems. Operating Systems: An Advanced Course.* Springer-Verlag, Berlin, 1978.

[Gra81] J. Gray. The transaction concepts: Virtues and limitations. In *Proceedings of the International Conference on Very Large Data Bases*, pages 144–154, 1981.

[KELB90] E. Kuhn, A. K. Elmagarmid, Y. Leu, and N. Boudriga. A parallel logic language for transaction specification in multidatabase systems. Technical Report CSD-TR-1031, Purdue University, October 1990.

[LBEK] Y. Leu, N. Boudriga, A. K. Elmagarmid, and E. Kuhn. Logic framework for the specification of complex transactions. In preparation.

[LE90] Y. Leu and A. Elmagarmid. A hierarchical approach to concurrency control for multidatabases. In *Second International Symposium on Databases in Parallel and Distributed Systems*, July 1990.

[Mos81] J.E. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing,* PhD thesis, Dept. of Electrical Engineering and Computer Science, MIT, April 1981.