CovaTM: A Transaction Model for Cooperative Applications

Jinlei Jiang* Guangxin(Gavin) Yang** Yan Wu* Meilin Shi*

*Department of Computer Sciences, Tsinghua University, Beijing, P. R. China 100084

{jjlei,wyan,shi}@csnet4.cs.tsinghua.edu.cn

**Bell-Labs Research, Lucent Technologies, 600 Mountain Avenue, Murray Hill, NJ 07974, USA

gxyang@acm.org

Abstract

It has been widely recognized that traditional transaction models with **ACID**(Atomicity, Consistency, Isolation and Durability) properties generally are not applicable to cooperative applications. Though many advanced transaction models have been proposed to address the problems, they are too database-centered or too rigid to be useful in real environments. This paper presents a new transaction model named **CovaTM**, which provides sophisticated but flexible control over cooperative process as well as support for error recovery and exception handling. The most distinguished feature of this model is that user intervention is explicitly introduced into transaction processing. This paper details the features and structural elements of this model. An example is also given to illustrate how it works in real world settings.

Keywords

Cooperation, advanced transaction model, ACID, recovery, Cova

1. INTRODUCTION

In modern organizations, especially those involving design and product manufacturing, close cooperation between co-workers is often needed in order to get the work done. New tools and applications have been developed to enable such cooperation. The most outstanding example would be workflow management where a group of people work together to achieve some common goal. These applications are often of long duration and consist of multiple steps that are executed over possibly heterogeneous, autonomous and distributed environment. As organizations evolve, it is widely accepted that effectiveness or performance is no longer the dominant factor to achieve their goals. Also, reliability has been shown to be crucial. Failure recovery and exception handling in such applications are attracting more and more attention.

In traditional database systems, the above issues have been widely addressed by the concept of transaction. Key to the success of the transaction model is the atomicity, consistency, isolation and durability properties. Atomicity ensures that either all operations of a transaction complete successfully or all of its effects are absent. Consistency ensures that a transaction when executed by itself, without interference from other transactions, maps the database from one consistent state to another consistent one. Isolation ensures that no transaction ever views the partial effects of other transactions even when transactions execute concurrently. Durability ensures the changes to the database are persistent even when systems crash. However, this once successful concept is unsuitable for cooperative applications. To view the whole cooperative process just as a transaction will produce many problems and some of them are listed below:

- The work done during a long transaction will be lost when failure occurs before the end of the transaction. Unfortunately, failures are common for long-running applications and lose of work is not acceptable for mission-critical applications.
- Message or control exchanges among transactions are not supported. However, it is a common case for cooperative applications to have some type of dependencies among them.
- Lock mechanism reduces the throughput or concurrency. The longer the duration, the larger the reduction.

To address the above problems, several extended transaction models [8,11] have been proposed, including Sagas [10], long-running activities [5,6], ASSET [3], multi-databases [9,16] and so on. Usually, they are called advanced transaction models (**ATM**) to distinguish from the traditional one. Many of these models are developed from a database point of view and are too database-centric to provide adequate flexibility. In this paper, a transaction model **CovaTM** is proposed to support cooperative applications.

In **CovaTM**, we provide a way to describe cooperative applications, where a transaction is treated as one execution of a cooperative process with its sub-transactions corresponding to activities. By cooperative applications we mean applications or tools developed to support cooperations between users such as workflow management systems(**WfMS**). An activity or sub-transaction in **CovaTM** may be reactivated after its submission. Therefore, activities of a transaction form a graph other than a tree like in long-running activities [5,6]. Based on the description of the application, the run-time system can guarantee the reliability of execution to its best.

The rest of the paper is organized as follows. Section 2 presents a brief review of **ATM**s, which provide a solid base for developing new models. Then **CovaTM** model is described in section 3 and its implementation is introduced in section 4. Finally, we conclude our work in section 5.

SAC 2002, Madrid, Spain

2. RELATED WORK

Till now, many advanced transaction models have been suggested. For reasons of space we will only present the ones of most interests. Similarly, for a specific model, literal description is used instead of the formal one. The interested readers can find the details in [8,11,13].

Nested Transactions As an important step in the evolution of a basic transaction model, nested transactions extend the flat transaction structure to a multi-level one. In a nested transaction, a child transaction may start after its parent has started and may commit locally. A parent transaction may terminate only after all its children terminate. However, if a child fails, the parent may choose its own way for recovery. If a parent transaction is aborted, all its children are aborted. Although permitting increased modularity and finer granularity of failure handling, nested transactions provide full isolation (The committed local result is released only when all of its parents up to the root have successfully terminated.) at a global level that is not acceptable in some cooperative applications such as co-design system.

Sagas Sagas were originally proposed as a way to solve the problems related to long lived transactions[10]. The basic idea of Saga is to allow a transaction to release resources before committing by using the concept of compensating transactions. A Saga is a transaction that consists of a sequence of **ACID** sub/transactions and associated compensating ones. Each sub-transaction is allowed to commit individually and its effect can be explicitly undone by its compensating transaction. By allowing sub-transactions to commit on their own, Sagas relax the full isolation requirements and increase the inter-transaction concurrency. However, Sagas demand every sub-transaction has its counterpart, which makes its hard to be useful in some cases. On the other hand, compensating a sub-transaction can be very expensive and sometimes it is unnecessary. An example can be found in [7] and other problems related to Saga can be found in [13]. Nevertheless, the idea has been implemented in a workflow management system [1].

Flexible Transactions This model [9] is suitable for multi-database environment where each local database acts independently from others. In such environment, it is not possible to enforce the commit semantics of a global transaction [16]. A flexible transaction uses functionally equivalent sub-transactions as its alternative execution paths. It commits if either the main sub-transaction or their alternatives commit. To relax the isolation requirement, a flexible transaction uses compensation and relaxes global atomicity requirement by allowing the transaction designer to specify the acceptable states for the termination of a flexible transaction, in which some sub-transactions may be aborted. Time factor is also taken into account. Sharing some features of workflow management, flexible transactions can be easily implemented within a workflow management system [1]. For this reason, we base our model on it. However, scalability and access control are not addressed in flexible transaction.

CoAct Cooperative Activity Model[14] provides the transactional properties applicable to cooperative scenarios. Each user in CoAct works in his/her own workspace (called private workspace) and they cooperate through the controlled information exchange and synchronization of their private workspaces. The model works like this: a certain parameterized CoAct is used to describe a particular activity and by instantiating it we get a concrete activity. Each participant of a cooperative activity has his/her own activity (called user activity). One final result is obtained by merging the result of each user activity. This model is well suited for building asynchronous cooperative applications. But because of the static description of cooperative activity, it is not flexible enough.

The above work has solved the problems faced by the cooperative applications more or less from different perspectives. For example, Sagas and Flexible transaction are around database whereas CoAct faces the specific applications. But still a lot of problems remain open. For example, the scalability and access control have not been addressed. One big problem with these models is that features required by one application might be unacceptable in another one. In other words, they are not as flexible as possible. Nevertheless, the ideas behind these models are very helpful for designing new models.

3. CovaTM MODEL

In this section, we will give a detailed description of **CovaTM**. Providing integrated exception handling and recovery as well as access control, **CovaTM** can support reliable and flexible process instance enactment, rollback and compensation.

3.1 Formal Definitions

Sub-transactions within a **CovaTM** transaction are tightly coupled. For instance, the equivalent sub-transaction can't be executed until the preferred one has aborted. To specify this execution dependency, we define the execution state of a transaction as follows:

Definition 1 Execution State of a Transaction

For a **CovaTM** transaction T with m sub-transactions, the transaction execution state x is an m-tuple $(x_1, x_2, \ldots, x_m)$ where

$$x_i = \begin{cases} N & \text{if } t_i \text{ has not been submitted for execution} \\ E & \text{if } t_i \text{ is being executed} \\ S & \text{if } t_i \text{ has successfully completed} \\ A & \text{if } t_i \text{ has aborted} \\ F & \text{if } t_i \text{ has failed} \end{cases}$$

Here **N** is called the initial state and **S**, **A**, the end state and **E**, **F**, the intermediate state. The state transition is shown in Figure 1.

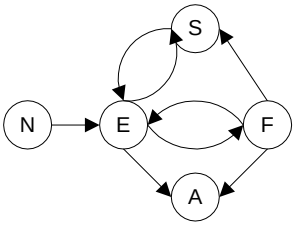

Figure 1 state transition diagram

Execution state of a transaction T is used to keep track of the execution of the sub-transactions. It is also used to determine whether the objectives of T have been achieved. At the beginning of T, all x_i's are set to **N**. The value of x_i is set to **E** when t_i is submitted. When t_i completes the execution state x_i is set to **S** if it has successfully completed (or achieved its objective), and to **A**, otherwise. During the execution, failures and exceptions may occur and then x_i is set to **F**. Afterwards, exceptions are handled and x_i can then be set to **E**, **S** and **A** respectively according to the context. In the end, a sub-transaction can only be in state **S** or **A**. One major difference between our

model and the others is that in our model, a successfully finished sub-transaction can be reactivated for further execution. This is reflected by the state transition from **S** to **E** in Figure 1. We denote by **X** the set of all possible execution states.

At a certain point of execution, the objectives of T may be achieved. In this case, T is considered to be successfully completed and can be committed. The corresponding execution state is called an acceptable state. Usually, there is more than one acceptable state for T because many ways can lead to the same goal. We call the collection of them as an acceptable state set which is defined as follows.

Definition 2 Acceptable State Set

The acceptable state set, denoted by **AS**, of a **CovaTM** transaction T is a subset of **X**, where

AS={x | x∈ **X**, and in state x, the objectives of T are achieved.}

To determine the legal execution order of sub-transactions, we need to specify the execution dependencies among them. Two basic dependencies are defined to express the general execution dependencies. The first is *positive dependency*. A positive dependency between sub-transaction t_1 and t_2 exists if t_1 can't be executed until t_2 succeeds. The second basic dependency is called *negative dependency*, which is used to specify the alternative sub-transactions. A sub-transaction t_1 negatively depends on t_2 if t_1 has to wait until t_2 has aborted before it can start. This happens when t_1 and t_2 implement the same task in a global transaction and t_2 is preferred to t_1. In order to express the execution dependencies, we associate with each sub-transaction t_i a precedence predicate pp_i defined as follows:

Definition 3 Precedence Predicate

A precedence predicate pp_i for a sub-transaction t_i is a boolean function defined on **X**, where

$$pp_i : \mathbf{X} \rightarrow \{true, false\}$$

To indicate that t_j is positively depends on t_i, we formulate the precedence predicate $pp_j := (x_i = S)$. We use the precedence predicate $pp_j := (x_i = A)$ to denote that t_j negatively depends on t_i. Having the basic dependencies, we can express any execution dependency in term of boolean combination of the basic dependencies. The value of the precedence predicate changes as the global transaction is executed and is used to determine whether the corresponding sub-transaction can be submitted for execution at the current time. Different execution strategy can be achieved by specifying different precedence predicate for each sub-transaction.

In cooperative work, some tasks may be time-critical. To express this requirement, we define temporal predicate.

Definition 4 Temporal Predicate

A temporal predicate tp_i for a sub-transaction t_i is a time requirement of t_i. It has the following format:

Operator Hour:Minute:Month:Day:Year

The Operator can be *before*, *after* or their combination, denoted by *between*. For not all of the sub-transactions have time requirements and not all of the time fields are needed, a wild card is also used to indicate "don't care" condition. For example, the predicate "before (17:*:*:*:*)" stands for the task should be finished before 5 p.m.

A coordinator plays a very important role in achieving the common goal of a cooperative application. A manager is also needed to manipulate the executing process. So we define transaction administrator as follows:

Definition 5 Transaction Administrator

A transaction administrator is an active entity, such as a person or a program that is responsible for the execution of the transaction, i.e. it is up to the administrator to determine what to do when exceptions occur.

Note that (1)Not all the exceptions are reported to the administrator. Instead, the administrator need only to handle such exceptions as the system knows little about what to do. (2)Transaction administrators are only responsible for their own transaction(s). For instance, a global transaction administrator is only responsible for the global transaction, while the administrator of sub-transaction t_i just cares for t_i.

Based on the above definitions, we can formulate a **CovaTM** transaction as:

Definition 6 CovaTM Transaction

A **CovaTM** transaction T is a 6-tuple (**ST**, **O**, **PP**, **TP**, **AS**, **M**) where

- **ST** is the set of all sub-transactions of T
- **O** is the partial order on **ST**
- **PP** is the set of all precedence predicates of **ST**
- **TP** is the set of all temporal predicates of **ST**
- **AS** is the set of all acceptable states of T
- **M** is the set of administrators of T

In order to specify a **CovaTM** transaction, we have to specify the set of sub-transactions. For each sub-transaction, we specify its type and participants. The sub-transaction type can be one of the followings:

- **CP** – if the sub-transaction is compensable
- **UC** – if the sub-transaction is uncompensable

The above two are called simple types.

- **CT** – if the sub-transaction is compound, i.e. it has its own sub-transactions, thus forming a transaction tree

We also specify the precedence predicate and the temporal predicates of the sub-transaction. At the global level, we specify the partial order **O**, the set of acceptable states **AS** and the transaction administrator. We can see from the above definitions that the minimum schedule unit of our model is a simple sub-transaction, which can be the composition of a series of traditional transactions consisting of read and write operations. Each simple sub-transaction can commit and release its resources before the global transaction successfully completes and commits, so the full isolation requirement is relaxed. When one sub-transaction fails, an equivalent one will be executed instead of aborting the global transaction, thus the atomicity property is also relaxed. Now let's look at an example to get an intuitive impression.

3.2 An Example

The example used here is the well-known example from transaction literature: planning a business trip. Consider "planning a business trip from Beijing to Shanghai" as a **CovaTM**, which consists of the following subtasks:

1. Customer reserves a business trip to an agency
2. Agent reserves a vehicle from Beijing to Shanghai
3. Agent reserves a hotel in Shanghai

Assume that we have two choices for task2 and three for task3. Let t_1 be the request from customer, say John, and t_2 ordering a

train ticket, t_3 ordering a plane ticket and t_4, t_5, t_6 reserving a room at hotel H1, H2 and H3 respectively. We say t_2 and t_3 are alternative/equivalent sub-transactions for ordering a ticket, so do with t_4, t_5 and t_6. Obviously, t_1 is uncompensable. Also assume t_2 and t_3 are uncompensable too. In addition, the ticket must be before January 17, 2001. Let Tom be the administrator for the whole transaction, John for t_1 and Jack for all the other sub-transactions. At last we assume that t_2 is the first choice for task2 and t_4 and t_6 are the first and last choice for task3 respectively. Then the example can be formally specified as follows:

ST = { t_1(UC,{John}), t_2(UC,{Jack}), t_3(UC,{Jack}),
t_4(CP, {Jack}), t_5(CP, {Jack}), t_6(CP,{Jack}) }

O: $t_1 < t_2$, $t_1 < t_3$, $t_2 < t_4$, $t_2 < t_5$, $t_2 < t_6$, $t_3 < t_4$, $t_3 < t_5$, $t_3 < t_6$

In fact, **O** gives out the process structure represented by directed graph.

$$PP: \begin{cases} pp_1 := true \\ pp_2 := (x_1 = S) \\ pp_3 := (x_1 = S) \wedge (x_2 = A) \\ pp_4 := (x_2 = S) \vee (x_3 = S) \\ pp_5 := (x_2 = S \vee x_3 = S) \wedge (x_4 = A) \\ pp_6 := (x_4 = A) \wedge (x_5 = A) \end{cases}$$

For x_4=A implies that either t_2 or t_3 has been successful (because there is no transition from N to A), we omit the condition (x_2=S$\vee$$x_3$=S) in pp_6. For the same reason, pp_6 can also be written as pp_6 := (x_5=A). So do with pp_3 and pp_5. It is up to **O** and **PP** to determine the execution order.

$$TP: \begin{cases} tp_1 := * \\ tp_2 := before(*:*:*:1:17:2001) \\ tp_3 := before(*:*:*:1:17:2001) \\ tp_4 := * \\ tp_5 := * \\ tp_6 := * \end{cases}$$

AS = { (S, S, N, S, N, N), (S, N, S, S, N, N), (S, S, N, N, S, N),
(S, N, S, N, S, N), (S, S, N, N, N, S), (S, N, S, N, N, S) }

M = {Tom}

3.3 Execution Rules

In this section, we will explain execution rules that a **CovaTM** transaction must abide by. At first, we will define the predecessor and successor of sub-transaction t_i.

Definition 7 predecessors of sub-transaction
Predecessors of a sub-transaction t_i, denoted by pred(t_i), are those sub-transactions which precede t_i in the partial order **O**, i.e.

$$pred(t_i) = \{t_j \mid t_j \in \mathbf{ST} \text{ and } t_j < t_i \text{ in } \mathbf{O}\}$$

Definition 8 successor of a subtransaction
Successors of a sub-transaction t_i, denoted by succ(t_i), are those sub-transactions which follow t_i in the partial order **O**, i.e.

$$succ(t_i) = \{t_j \mid t_j \in \mathbf{ST} \text{ and } t_i < t_j \text{ in } \mathbf{O}\}$$

A sub-transaction t_i is executable if
(1) t_i is not in state **E** or **F**; and
(2) $t_k \in$ pred(t_i)-succ(t_i), either t_k has been executed or the pp_k is false; and
(3) both the pp_i and the $tp_i(t)$ are true.

From the above, we can see that a terminated sub-transaction can be re-executed as long as the condition is satisfied.

We can now formulate the execution rules as an algorithm named **TransactionScheduler**. The exception handling process will be described in the next section. The execution of T terminates when any of the following conditions or events occurs:

- The current execution state is acceptable
- None of the sub-transactions is executable and no sub-transaction is in state **E** or **F**.
- Transaction administrator issues a termination command.

```
Algorithm TransactionScheduler
In: A CovaTM transaction T_t to be executed
Out: Execution Result of T_t
{
  foreach (t_i ∈ T_t)
    x_i←N;
  while (true) //Loop forever until terminated
  {
    foreach (t_i ∈ T_t)
      if (t_i is executable)
      {
        x_i ← E;
        start a new thread for t_i; //Execute t_i
      }
    //wait for the execution result
    R← WaitForExeResult(T_t);
    i← R.id; //The name of the returned sub-transaction
    switch(R.code)//The return code of sub-transaction
    {
      case SUCCESS: // Objective achieved
        x_i← S; break;
      case ABORT: //Objective not achieved
        x_i← A; break;
      case EXCEPTION: //Exception occurs
        x_i← F;
        call ExceptionHandler; //Handle it
        break;
    }
    //Check to see if termination condition is met
    if (terminated(T_t))
      return T_t.result; //return execution result of T
  }
}
```

Figure 2 Transaction Scheduling Algorithm

According to this algorithm, concurrent execution of sub-transactions is allowed if they are executable at the same time. When the result of the execution is known, we modify the transaction execution accordingly. After the completion of a sub-transaction, we check if the termination condition is satisfied. If it is not satisfied, we continue scheduling the sub-transactions. If the global transaction terminates and an acceptable state has been

reached, we can commit T; otherwise, it must be aborted. Transaction administrator can terminate the transaction at any time no matter what the execution state is.

3.4 Exceptions and Recovery

Exceptions are inevitable during the execution of cooperative applications. Usually, the longer the duration lasts, the more possible. Much work [2,4,12] has been done related to exception handling in workflow systems. As we all know, it is expensive for a transaction to be aborted on every failure of a step, because rolling back a transaction to its beginning could potentially undo a lot of work. As an alternative to aborting, our model supports exception handling and recovery.

In our model, two levels of exception handling are provided. The first one is for predictable exceptions and the second for unpredictable ones. By predictable, we mean this type of exception can be known before hand, e.g. trying to read a file that does not exist, communication errors such as loss of information, and hardware errors such as computer system crashes. For these errors the designer can tell the system what to do in advance. Unspecified exceptions are reported to transaction administrator to determine what to do. Once an exception is captured during the execution, the state of the corresponding sub-transaction is set to **F** and then it is handled as algorithm **ExceptionHandler**.

```
Algorithm ExceptionHandler
In: Transaction t; Exception e
Out: none
{
   look up a handling rule for e from RuleBase;
   if (found) //Handled by the program
   {
      Handle exception e according to the rule;
      Reset the state of t accordingly;
   }
   else //Handled by the transaction administrator
      Report exception e to administrator unit;
}
```

Figure 3 Exception Handling Algorithm

The process for above algorithm is illustrated in figure 4. When exception occurs, the normal execution of sub-transaction is suspended and the control logic is transferred to the exception handler unit. After the exception has been handled, the control logic is returned and the execution is resumed. The transaction administrator handles exceptions via administrator interface, which will be introduced in the next section.

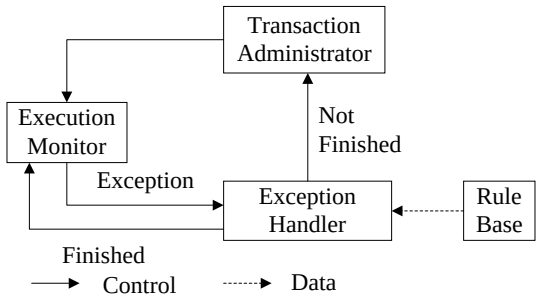

Figure 4 Exception Handling Process

The introduction of rule base promotes the system flexibility enormously. By defining different handling rules on the same process, we can implement various run-time controls. For example, in the trip process, if t_3 is aborted, the process will terminate unsuccessfully. However, if we treat such a situation as an exception, we can go on with the execution by simply adding a new rule to Rule Base that makes pp_4 true (the termination condition should be changed accordingly.). This is very useful for none mission-critical tasks, where even some steps fail, the others can still run on. By this means, we can generate new child/sibling transactions of the current one or even another global transaction. Thus the coordination between different transactions becomes possible. The transaction administrator can also alter the execution strategy through the administrator interface. Also take the trip process as an example. Task2 and task3 can be executed in parallel in order to increase the system throughput. To achieve this purpose, what the administrator need to do is merely to change pp_4 to pp_4:=(x_1=S). Thus, when t_1 finished successfully, according to **TransactionScheduler** algorithm, both t_2 and t_4 can be started for pp_2 and pp_4 are true.

Related to exception, recovery is also very important in a transaction model. In our model, we use mixed type of sub-transactions. For compensable sub-transaction, its execution result is saved immediately after it is submitted. While for uncompensable one, its results are saved to a temporal file and stored to the system permanently on success of the global transaction. To cancel the execution effects of a transaction (called **backward recovery**), we just execute the compensating sub-transactions corresponding to compensable ones and reject the temporal results of uncompensable one. For example, if task2 and task3 run in parallel, when task2 fails, the effect of successfully finished task3 can be semantically undone by executing its corresponding compensating activity (not presented in the process). To go on with the execution upon exceptions (called **forward recovery**), log file is used to keep the state of the global transaction and the operation of each sub-transaction. Once the system restarts after crash, the information is used to restore the system state and data of the executing transaction.

However, not all activities need compensation. How to determine the compensation scope is another interesting problem. Some work has been done in [7], but it is assumed that the activities of a workflow process are executed sequentially. In fact, many sub-transactions can be executed concurrently, this may result in unexpected states. To take this effect into account, a lot of work is left to do. In our model, we adopt an alternative way currently, i.e. transaction administrator is required to specify the compensation scope before the recovery process starts. First, the self-generated (according to the process definition) compensation process is presented to the administrator. Then the administrator determines which one to be compensated. At last the specified compensation process is executed and the recovery is finished.

4. IMPLEMENTATION

To facilitate the design of cooperative applications, we develop a high level language called *Cova* (***Co***operati***v***e ***a***pplication), which provides the facilities for describing both the coordination and the computation parts of a cooperative application. Grounded on the semantics of these descriptions, *Cova* runtime system provides the services needed at various stages of a cooperation process[15].

Exploiting the description ability provided by *Cova* language, we develop a transaction service in *Cova* runtime system to implement **CovaTM** transactions.

4.1 Process Design

To depict a cooperative application, we must adopt a proper coordination model, which describes the properties of all stages of a cooperation process as well as the control and data flow among these stages. In *Cova*, a cooperation process is modeled as a set of interrelated *activities*. An *activity* is a stage or a step in a cooperation process. It is a computing entity that has its lifecycle and the rules guiding its state transition. Thus, implementing **CovaTM** in *Cova* runtime system is straightforward. The whole cooperative process is treated as a **CovaTM** transaction with each sub-transaction represented as an activity.

Triggers are used in [5,6] to offer data- or event-driven specification of control flow, and thus provide a flexible framework. Due to its powerful functions, trigger is also adopted by *Cova*. We associate each activity a trigger list Tr, which describes which action should be triggered upon the occurrence of specific events. The element of Tr has the format of ⟨e, c, a⟩, which is similar to the well-known *event-condition-action* rule developed for active DBMSs. Figure 5 shows how the previous example can be described in *Cova*. Words in bold in figure 5 are the keywords of *Cova* language.

In this program, **startsat** and **startswhen** are used to describe precedence predicate with the former indicating the entrance of the process (i.e. pp_1) and the latter indicating the start condition of other activities (i.e. pp_2-pp_6). The keywords **trigger**, **when** and **where** are exploited to specify the action, event and condition respectively. The control flow of the process is also specified by ECA rule. The transaction administrator is defined by word **administrator** and the activity participants are defined by **receiver**. **Time** is employed to depict time-critical activity while **type** specifies the sub-transaction type.

We can get a process definition by compiling the program. During this process, the elements of the **CovaTM** are specified. Roughly speaking, **O** is constructed by analyzing the triggering events *complete* and *abort*. **PP**, **M** and **ST** are given by the program directly. **TP** can be obtained by analyzing the triggering time-event and **AS** is implied by the triggering event *complete* plus the corresponding conditions. For example, in the above program, the completion of t_4 activates no other activities, so it is also the completion of the cooperative process. In addition, execution of t_4 implies that t_2 or t_3 finished successfully for the start condition of t_4 is (t_2.state==S||t_3.state==S). Thus, we get two acceptable states, i.e. (S, S, N, S, N, N) and (S, N, S, S, N, N). Go on with the process, at last **AS** is obtained. Once the construction completes, the process is ready for execution in *Cova* runtime system. The **TransactionScheduler** and **ExceptionHandler** consist of the transaction service of *Cova* runtime system.

Once a cooperative process is defined, it is fixed and stored in the system permanently. Unfortunately, the environments or the commercial rules are changing as the time goes on, which makes it difficult for the predefined process to meet the changing needs. To solve this problem, we provide run-time modification of process by administrator interface. The system maintains a list of transactions for each administrator.

```
public process TripReservation startsat t1 administrator Tom;
{
  activity t1 type UC //transaction type
  {
    receiver John; //participant
    trigger t2.AcceptRequest(this) //action
      when after complete where true//event & condition
    trigger Exception.Handle(this.except)
      when after execute where t1.state=E
    . . .
  };
  activity t2 type UC startswhen (t1.state==S) //condition
  {
    receiver Jack;
    trigger t3.AcceptRequest(request)
      when after abort where true
    trigger t4.ReserveRoom()
      when after complete where true
    trigger Exception.Handle(TimeOut)
      when after time (*:*:*:1:17:2001) where true
    . . .
  };
  . . .
  activity t4 type CP startswhen (t2.state==S || t3.state==S)
  {
    receiver Jack;
    trigger t5. ReserveRoom()
      when after abort where true
    trigger process.Commit() //Commit the transaction
      when after complete where true
    . . .
  };
  . . .
};
```

Figure 5 An Example Process by Cova

4.2 Administrator's Interface

Transaction administrators interact with the system through an interface provided by *Cova* runtime system. Administrators can control the execution process of transactions so that they can be adapted to changing conditions. Commands provided by this interface are similar to those in ASSET[3] and listed below.

- **abort (t_i)**: abort sub-transaction t_i and set the state of t_i to A. If t_i has already committed, it does nothing.
- **commit(t_i)**: commit the operation of sub-transaction t_i and set the state of t_i to S. It does nothing if t_i has been submitted successfully.
- **form_dependency(type, t_i, t_j)**: form a dependency of the specified type between t_i and t_j. The dependency type should be positive or negative.
- **compensate(t)**: call the compensation process of transaction t. After this is done, the transaction may be aborted and its effects are canceled or it can be restarted from some point.
- **delegate(t_i, t_j)**: transaction t_i transfers to t_j the responsibility for all the operation.

5. DISCUSSIONS AND CONCLUSIONS

Cooperative applications, which are usually of long, uncertain duration and consist of multiple steps that are executed over possibly heterogeneous, autonomous and distributed environment, impose new requirements on transaction models, i.e. to support the cooperation of a group of people working together for a common goal. In this paper, a transaction model named CovaTM is proposed to guarantee the fulfillment of such applications and make the sharing and exchange of information among co-workers as natural as possible. Before giving remarks on the model, we should make some points clear here.

(1) User intervention may violate the transparency of transaction from the administrator point of view, but it doesn't hold for general user, who only knows his/her own work to do. From the experience of the real world, there are usually one or more chargers for a project who are aware of the project goal. Therefore, we argue that it is proper to define a transaction administrator to handle unspecified exceptions or failures for he/she is familiar with the common goal and can make correct decisions under such conditions.

(2) The introduction of compound sub-transaction type makes it possible to support subtasks. Subtasks are very useful in flexible cooperative process. Due to the enormous complexity of real-life application, it is impossible to identify all control and correction steps a priori. For the same reason, it is not possible to prescribe the sequence of control and correction steps in detail. By subtask we can decompose a complex process into smaller ones and design respectively. So it is necessary to introduce the compound sub-transaction type.

(3) The model works as a system-level tool rather than a user-level one. Although administrator interface is provided for administrators to interact with the system, it is invisible to most users. The main purpose of this interface is to improve the system adaptability.

Now we can conclude **CovaTM** with the following assertions:

- Sophisticated control of flow makes it possible to describe the wide spectrum of cooperative activities from structured to non-structured.
- User intervention enhances the system flexibility and ability.
- Time predicate can be used to depict time-critical and none time-critical activities uniformly.
- Mixed sub-transaction types make it more flexible.

All of the above make it appropriate for the cooperative applications.

6. ACKNOWLEDGMENTS

This work was done at Bell-Labs Research China. It is co-supported by National Natural Science Foundation of China under grant No. 60073011 and 985 Project of Tsinghua University.

7. REFERENCES

[1] G. Alonso, D. Agrawal, A. E. Abbadi, M. Kamath, R. Gunthor and C. Mohan, Advanced Transaction Models in Workflow Context, IBM Research Report RJ9970, IBM Almaden Research Center

[2] G. Alonso, C. Hagen, D. Agrawal, A. E. Abbadi and C. Mohan, Enhancing the Fault Tolerance of Workflow Management Systems, in IEEE Concurrency, July-September 2000

[3] A. Biliris, S. Dar, N. Gehani, H. V. Jagadish and K. Ramamritham, ASSET: A System for Supporting Extended Transactions, in proceedings of SIGMOD'94, 44-54

[4] A. Borgida and T. Murata, Tolerating exceptions in workflows: a unified framework for data and processes, in proceedings of WACC'99, 59-68

[5] U. Dayal, M. Hsu and R. Ladin. A Transactional Model for Long-Running Activities, in proceedings of VLDB'91(Barcelona, September 1991), 113-122

[6] U. Dayal, M. Hsu and R. Ladin. Organizing Long-running Activities with Triggers and Transactions, in proceedings of SIGMOD'90, 204-214

[7] W. Du, J. Davis and M. Shan, Flexible Specification of Workflow Compensation Scopes, in proceedings of GROUP'97, 309-316

[8] A. Elmagarmid (eds.), Database Transaction Models for Advanced Applications, Morgan Kaufmann Publishers, 1992

[9] A. K. Elmagarmid, Y. Leu, W. Litwin and M. Rusinkiewiczt, A Multidatabase Transaction Model for InterBase, in proceedings of VLDB'90 (Brisbane, Australia 1990), 507-518

[10] H. Garcia-Molina and K. Salem. SAGAS, in proceedings of SIGMOD'87, 249-259

[11] S. Jajodia and L. Kerschberg (eds.), Advanced Transaction Models and Architectures, Kluwer, 1997

[12] M. Klein and C. Dellarocas, A Knowledge-based Approach to Handling Exceptions in Workflow Systems, in Computer Supported Cooperative Work (CSCW): The Journal of Collaborative Computing, 2000, 9(3/4):399-412

[13] C. Mohan. Tutorial: Advanced Transaction Models – Survey and Critique, in proceedings of SIGMOD'94.

[14] M. Rusinkiewicz, W. Klas, T. Tesch, J. Wasch and P. Muth, Towards a Cooperative Transaction Model—The Cooperative Activity Model—, VLDB'95

[15] G. X. Yang and M. L. Shi, Cova: A Programming Language for Cooperative Applications, in Science in China Series F, 2001, 44(1):73-80

[16] A. Zhang, M. Nodine, B. Bhargava and O. Bukhres, Ensuring Relaxed Atomicity for Flexible Multidatabase Systems Transactions in Multidatabase Systems, in proceedings of SIGMOD'94, 67-78

Biographical note: Jinlei Jiang is a Ph.D student in Department of Computer Science and Technology at Tsinghua University, Beijing, China. His research interests include Computer Supported Cooperative Work(CSCW), Workflow Management Systems(WfMS), Programming Language and Systems.