Distributed and Parallel Databases, 8, 399–446, 2000

Flexible Transaction Dependencies in Database Systems

LUIGI V. MANCINI mancini@dsi.uniroma1.it
Dipartimento di Scienze dell'Informazione, Università di Roma "La Sapienza", Rome, Italy

INDRAJIT RAY indrajit@umich.edu
Department of Computer and Information Science, University of Michigan-Dearborn, Dearborn MI 48128, USA

SUSHIL JAJODIA jajodia@gmu.edu
Department of Information and Software Engineering, George Mason University, Fairfax, VA 22030-4444, USA

ELISA BERTINO bertino@dsi.unimi.it
Dipartimento di Scienze dell'Informazione, Università di Milano, Milano, Italy

Abstract. Numerous extended transaction models have been proposed in the literature to overcome the limitations of the traditional transaction model for advanced applications characterized by their long durations, cooperation between activities and access to multiple databases (like CAD/CAM and office automation). However, most of these extended models have been proposed with specific applications in mind and almost always fail to support applications with slightly different requirements.

We propose the Multiform Transaction model to overcome this limitation. The multiform transaction model supports a variety of other extended transaction models. A multiform transaction consists of a set of component transactions together with a set of coordinators which specify the transaction completion dependencies among the component transactions. A set of transaction primitives allow the programmer to define custom completion dependencies. We show how a wide range of extended transactions can be implemented as multiform transactions, including sagas, transactional workflows, nested transactions, and contingent transactions. We allow the programmers to define their own primitives—having very well-defined interfaces—so that application specific transaction models like distributed multilevel secure transactions can also be supported.

Keywords: database management systems, transaction processing, transaction dependencies, commit protocols, distributed systems, multilevel security

1. Introduction

The atomicity property of the classical transaction model has often been regarded as too restrictive for advanced database applications in distributed, cooperative and heterogeneous environments. For example long running activities when executed as atomic transactions, significantly delay the execution of shorter activities. In the case of multidatabase systems, the autonomy requirements of the component local databases are in direct conflict with the atomicity property of classical transactions. Consequently, in recent years, a number of extensions have been proposed to the traditional atomic transaction model with the goal of supporting more flexible transaction processing [9]. Examples of such extended transaction

models are SAGAS [11], Flex [6], Asset [5], DOMS [12] and ConTract [20]. These systems differ from each other in the way they model cooperating transactions.

The basic idea of those approaches is to provide transaction primitives and run-time environments, so that users can define their own transaction models. A theoretic foundation of this type of approach can be found in ACTA [7]. ACTA is a formal framework for specifying relationships among different cooperating transactions that can be used to formalize all these different extended models. ACTA classifies these relationships into two broad categories based on a transaction's effect on the commit and abort of other transactions (called completion dependencies) and on the transaction's effect on the data items it accesses (known as data dependencies). The developers of the ACTA framework have shown that several extended transaction models can be represented by using a number of basic dependency types. Though ACTA is open-ended, the framework as it is now does not model significant events in a system other than commit or abort of the transactions. Events, such as several error conditions that do not influence the commit or abort of transactions or the secure dependencies arising among subtransactions of a multi-level secure distributed transaction, are typical examples of dependencies that cannot be modeled within the ACTA framework.

We limit ourselves in this work, to exploring the category of transaction dependencies known as *completion dependencies*. Completion dependencies represent constraints on the order of termination of transaction. We feel that completion dependencies provide a convenient way to specify the behavior of concurrently executing and cooperating transactions although we agree that they are not enough to capture all types of relations among cooperating transactions. As we describe our work we will specify the assumptions we make to capture these other types of dependencies—namely the data dependencies.

We introduce the *multiform transaction* model as a tool-kit approach to transactional application development. With this approach the programmer can express any completion dependency as a program fragment. This is done by using the transaction processing primitives that we propose.

The goals of this work are:

1. to present the multiform transaction model that supports a variety of extended transaction models proposed in the literature;
2. to define a set of primitives which allows the programmer to specify various commit and abort dependencies among transactions and to realize relaxed correctness criteria;
3. to allow the programmer to define his/her own set of primitives for a flexible transaction coordination;
4. to reduce the programming effort to allow transaction code re-usability; and
5. to provide examples which show how the completion dependencies present in the different extended transaction models like transactional workflows, sagas, secure transactions etc., can be implemented as multiform transaction.

Our work focuses on managing completion dependencies at the programming language level, and proposes a linguistic construct that separates the coding of the transaction from

the definition of their dependencies. The reason behind this separation is to simplify the work of the programmer. Transactions can be thus coded without worrying about managing concurrent computations, communications etc.

If transactions in a group are related such that the termination of one member of the group determines the termination of other members or is determined by them, then these group of transactions are explicitly coordinated in our model. The programmer has the option of specifying a coordinate block that implements such coordination requirements. The coordinate block contains the specification of the completion dependencies in the form of a program. The coordinate block also allows the programmer to define his/her own primitives for further flexible coordination of transactions. A programmer defined primitive can be used from anywhere within a transaction and causes the execution of a specified code by the coordinator. This allows many different transaction models to be easily implemented.

The primitives, we introduce, can be used directly by the programmer as part of a programming language to design the program for coordinating transactions. Moreover, the compiler of a database programming language can also use those primitives to support higher level constructs for transactions. In this case, a pre-compiler can automatically generate the appropriate code needed for coordination of a set of transactions from the high level description of their dependencies. To demonstrate this idea we have developed a sample high level declarative language.

The remainder of this paper is organized as follows. Section 2 gives an overview of our approach by introducing the multiform transaction model and describing the execution model for a system that supports multiform transactions. It also presents an example of a multiform transaction and explains how such a transaction is executed in a distributed environment. Section 3 introduces the transaction primitives that we use to support multiform transactions. Section 4 discusses some of the major issues related to the multiform transaction model. Of particular importance is the issue of orphan transactions—transactions that remain active upon the termination of a multiform transaction. Subsection 4.1 introduces the notion of well-formed multiform transaction and describes how a compiler can determine statically whether there will be orphan transactions at the completion of a multiform transaction. Subsection 4.2 briefly discusses some recovery issues for multiform transactions. Section 5 deals with a particular instance of the deadlock problem that occurs when completion dependency are specified in addition to data dependencies. This problem is present in most extended transaction models that allow completion dependencies to be specified along with data dependencies although it has not been addressed. In particular, Section 5 gives examples of how deadlock can occur within a multiform transaction and how the deadlock can possibly be detected and re-solved. Section 6 gives examples of multiform transactions which implement the completion dependencies of the many different extended transaction models proposed in the literature—SAGAS, transactional workflows, nested transactions, multi-level secure distributed transactions, contingent transactions etc. We also show in Section 6 how the ACTA commit and abort dependencies can be represented in our model. Section 7 introduces the high level declarative language we have developed for completion dependencies. Section 8 discusses some of the different extended transaction models and compares our work with these, wherever applicable. Finally Section 9 concludes the paper.

2. Overview of our approach

2.1. The multiform transaction model

A *transaction* T_i in our model is defined as any sequence of operations on data items (both persistent and volatile) delimited by either the begin_trans(T_i) ... end_trans(T_i) pair or the begin_trans(T_i) ... abort_trans(T_i) pair. A transaction is written in a high level language supporting persistence, concurrency and the new transaction processing primitives that we introduce. Transactions implement the ACID properties—Atomicity, Concurrency, Isolation and Durability [13]—which ensure that concurrent transactions execute without interference from each other even though they operate on common database items.

A *Multiform Transaction* consists of a set T of such transactions and includes the definition of a set of termination dependencies among these transactions; the set of dependencies includes, but is not limited to, the commit or abort relationships among the component transactions. The multiform transaction is also written in the same high level language as its component transactions. It is organized as a set of *coordinate blocks* each of which consists of a subset, T', of the set of transactions T, such that there exists some completion dependency among the transactions in this subset. Each coordinate block has two distinct parts—the *transaction component* that contains either the definitions of the transactions in the coordinate block or a set of nested coordinate blocks, and the *protocol component*, that includes a definition of the dependencies among the transactions in the transaction component. The protocol component of a coordinate block can either be a program fragment in the high level language or a declarative description of the completion dependencies. A coordinate block in execution is called a *coordinator module* (CM), or simply *coordinator*, of the multiform transaction.

More formally a multiform transaction is defined as follows:

Definition 1 (Multiform Transaction). A multiform transaction is the pair $\langle T, C \rangle$ where

- T is a set of transactions on which two partial orders, R and S, are defined. R specifies the execution order among the transactions in T, and S specifies the completion dependencies in T.
- C is a sequence of coordinate blocks, each of which can be either a sequence of nested coordinate blocks or a flat coordinate block with the constraint that nested coordinate blocks cannot be concurrent.
- A coordinate block has two components—a transaction component and a protocol component. The transaction component groups a set of transactions $T' \subseteq T$ and/or a set of coordinate blocks $C' \subset C$. It implements the execution order of the transactions in T' as specified in R. The protocol component implements the completion dependencies among (a subset of) the transactions in T' specified by S.
- Every transaction $T_i \in T$ must belong to at least one coordinate block in C.

Note that we have defined two partial orders R and S on the set of transactions T. The partial order R is implicit within the structure of the program for the multiform transaction. Thus the transactions execute in the order they are listed within the multiform

transaction—sequentially or concurrently as the programmer specifies. Once a transaction has been executed, its termination (that is commit or abort) relative to other transactions within the multiform transaction, is determined by its position in the partial order S. Separating the execution order R from the completion dependency S has many advantages including:

1. more expressive power to the model with respect to specifying completion dependencies only, since the actual dependencies enforced by the system are obtained as a combination of R and S.
2. a simple implementation of complex dependencies as two separate components—transaction component and the protocol component.

A coordinate block can express different properties of transactions which are defined within its transaction component. For example if the type of a transaction (compensatable, retriable etc.) is known, then a coordinate block can be developed which will enforce this type specification. If some definition D for the successful or unsuccessful termination of the set of transactions T is given, C can be appropriately constructed so as to ensure that if the multiform transaction terminates in a state satisfying D, then success/failure is returned. Similarly if precedence/preference relations between the transaction in T are provided, then these can be enforced by the coordinate blocks. As shown later, a multiform transaction can specify the different commit or abort relationships among a set of transactions, or among different subsets of this set by defining one or more coordinators. To prevent conflicting decisions for the same transaction we do not allow coordinators to be concurrent. This is not a limitation because it does not prevent transactions from being concurrent.

2.2. Execution model

Basic transaction processing is achieved at a site by the cooperation of the Transaction Manager, the Log Manager, the Lock Manager and the Resource Manager. These components together form what is known as the Transaction Processing (TP) subsystem at the particular site and ensures the atomicity, consistency, isolation and durability (ACID) properties of the transactions executing at that site. The TP subsystem implements the basic transaction control operations like commit, abort, savework, rollback, begin-transaction, lock data items etc.

On top of the TP subsystem at every site, we assume that there is a *Transaction Management Adapter* (TMA) module that enhances the functionality of the underlying TP subsystem by implementing an extended interface of the TP system for our new transaction processing primitives. The notion of Transaction Management Adapter is borrowed from [2] where the authors propose also a lock adapter and a conflict adapter as add-on modules on top of an existing TP system to enhance the TP system's functionality. A discussion on these other adapters is beyond the scope of this work as we focus mostly on transaction termination dependencies. However we allow these other adapters in our architecture thus obtaining the same functionalities that the authors in [2] achieve.

Finally we assume that there is a TP Monitor at every site that is responsible for services such as support for multithreaded processes, interprocess communications, scheduling,

context management, resource administration etc. The TP Monitor gets requests for some functions to be executed and provides the run-time support needed to start and control the execution. Note that most of these services are provided by the operating system at the site the transaction is executing. Indeed the TP Monitor can very well be the operating system itself. However unlike conventional operating systems, the TP monitor needs to implement context management in a persistent manner. We do not discuss anything about the TP Monitor any further. In the subsequent discussions we will assume that the TP monitor is an integral part of the transaction processing subsystem and use the term TMA-TP module to indicate the extended transaction processing subsystem that include the various adapters.

A transaction T_i executing at some site interacts with the TMA-TP module at that site via a *coordinator module* (CM). This coordinator module acts as a transaction event handler and is implemented by the executable module corresponding to the coordinate block of which T_i is a part. Before transaction T_i gets executed its CM is started by the TP-Monitor as a set of concurrently executing threads (a daemon process). Execution of a transaction primitive by a transaction T_i is the transaction event that causes the CM corresponding to T_i to react and handle the event.[1] If a thread with the same name as the event is defined within the CM then the thread gets activated otherwise the CM lets the underlying TMA-TP module handle the event as appropriate. The executing threads can in turn invoke other primitives that are part of the TMA-TP module or other threads defined by the programmer, in order to actually handle the event. The coordinator module can be viewed as an extended form of the notion of *metatransaction* of [2] to include executing codes and a mechanism to specify and handle inter-transaction communication and synchronization.

Functionally a CM can be divided into two distinct parts: A required set of compiler-generated *event interceptors* and an optional set of programmer defined *event handlers*. The latter is essentially the programmer defined protocol component of the coordinate block. The set of event interceptors includes: (1) the mechanism to pass on relevant parameters from the run-time environment to the other system modules and vice versa, and (2) the information as to how a particular event is to be handled, i.e. whether by a programmer defined event handler or by the underlying TMA-TP module. In particular, the set of event interceptors contains mechanism to communicate with other CMs of the same multiform transaction and to pass on parameters to these CMs. An event interceptor is awakened by the occurrence of an event and then either invokes one of the threads in the CM or invokes an action exported by the TMA-TP module. Finally, the CM may contain a set of invariants for each component transaction. These invariants constitute the predicates that need to be satisfied before and after a transaction execution.

In a distributed setup the CM at the originating (coordinating) site of a multiform transaction is of the form just described. At remote sites where component transactions get executed, lightweight CMs are created. The lightweight CMs contain only the compiler generated event interceptors. Their function is to invoke the relevant threads executing at the CM of the coordinating site or at the TMA-TP module at the local site. They act as the interface between the TMA-TP at the remote site and the CM at the coordinating site. If the programmer does not explicitly specify any coordinate block (i.e. the programmer has not defined any thread to handle transaction events) then such lightweight CMs get loaded at every site and act as forwarding agents for transaction events to the TMA-TP

module at each site. The CM at the coordinating site executes the programmer defined code to perform the coordinating operations, commit some components of the coordinate block while aborting other components or taking some other action. If no component transaction of the coordinate block is executing at the coordinating site, the TMA-TP subsystem at this site is responsible only for the housekeeping functions (e.g., writing log records etc.) for the coordinate block as a whole and for its components, while the TMA-TP subsystems at the remote sites execute the component transactions as well as perform housekeeping (only for the component transaction executing at that site).

A multiform transaction submitted by the user to the transaction processing system at a particular site (the coordinating site) is executed as follows:

1. If the programmer has specified one or more coordinate blocks with the multiform transaction then
   (A) The transaction processing subsystem at the coordinating site executes one coordinate block after the other.
   (B) A coordinator module (CM) is created corresponding to a coordinate block. If there are nested coordinate blocks within this coordinate block, then the coordinator module spawns children coordinators at the same site. Such spawning can go to any depth as allowed by the resources available at the originating site.
   (C) A CM starts up its two components—the event interceptor and the event handler. It then loads a table—the *event mapper*—in the event interceptor that maps events to the respective handlers located in either the local TMA-TP system or in the programmer defined primitives.
   (D) If the coordinate block consists of component transactions that are to be executed at remote sites then the coordinator spawns a lightweight CM at each of these sites; these lightweight CMs contain only the mechanism to communicate with other sibling lightweight CMs and the parent CM, and also copy of the event mapper table.
2. If the programmer has not specified any coordinate block then
   (A) The transaction processing subsystem creates a lightweight CM at the coordinating site to pass on the event that has occurred to the underlying TMA-TP module.
   (B) If it is a distributed setup, the newly created CM spawns similar lightweight CMs at the remote sites.
3. When a transaction event occurs for some transaction T_i (that is T_i executed some transaction primitive), the event interceptor in the CM associated with T_i is awakened.
   (A) If the event has been the execution of some transaction primitive defined by the programmer then the event interceptor first checks the invariants, if any, that constitute the precondition for the primitive.

   If the precondition check fails the event handler part of the CM can do one of several things in order: (i) it checks if an error handling routine has been defined by the programmer and, if so, invokes it; (Note that if such an error handling routine has been defined by the programmer it will be part of the protocol component of the

multiform transaction and will be executing as one of the concurrently executing threads at the CM at the coordinating site) (ii) if no error handler has been defined by the programmer, then the event handler invokes an appropriate system defined error handler (on the assumption that one is available); (iii) it takes no action corresponding to the transaction event, just returns control to the TMA-TP subsystem for the execution of the next statement in the body of the transaction.

(B) If the precondition is satisfied then the thread with the same name at the coordinating site's CM gets activated, performs the actions defined by its code and returns control to the event interceptor of the CM associated with T_i. Note that prior to or along with invoking a thread at the coordinating site's CM, the event interceptor may also invoke actions at the local TMA-TP subsystems.

(C) When the initiating CM receives the result, it checks for postcondition satisfaction and, if the test is successful, returns the result of the thread execution to the transaction as if for a normal transaction primitive call. If the test is unsuccessful, a null value is returned. Note that before any result is returned, the CM may invoke any function at the local TMA-TP subsystem.

(D) If the event has been the execution of some primitive not defined by the programmer, then the event interceptor allows the local TMA-TP subsystem to handle the event appropriately.

4. The decision to end an executing coordinate block comes from the CM at the coordinating site. When this happens the different lightweight CMs at the various site all terminate and control gets returned to the TMA-TP subsystem at the coordinating site. If all coordinate blocks in the multiform transaction have been executed the multiform transaction terminates. Otherwise the next coordinate block is executed by the transaction processing subsystem as detailed in step 1 above.

Figure 1 gives a schematic diagram on how transaction events at a remote transaction are handled by the cooperation of the lightweight CM at the remote site, the CM at the coordinating site and the TMA-TP subsystems at both the sites. In the figure the begin_trans event is handled as a local TP system call by the TMA-TP subsystem at the remote site; the end_trans event is intercepted by the lightweight CM at the remote site and forwarded to the CM at the coordinating site. The latter in turn invokes a TP system call at its local TMA-TP subsystem.

Our model allows the programmer to define not only application specific transaction events (an example of this will be given later on in Section 6.3), but also to redefine the semantics of ordinary transaction events such as transaction completion or transaction begin, commit and abort. The programmer defined behavior get precedence over the default behavior and can thus be imposed on the latter.

In the following we give an example to illustrate the execution of a multiform transaction.

2.3. A multiform transaction implementing two sets of completion dependencies

We offer the programmer two methods for specifying completion dependencies. With the first approach the programmer directly uses the high level language and our new transaction

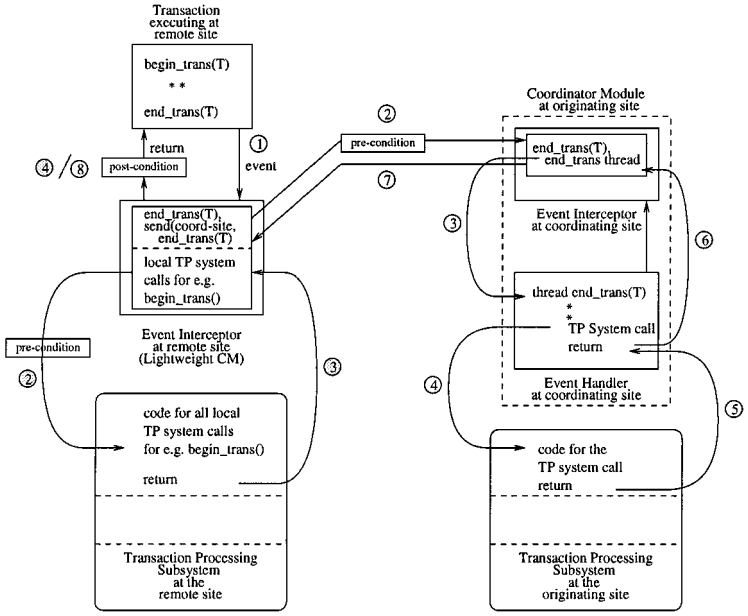


Figure 1. Event handling sequence for transaction events.

processing primitives to specify the dependencies. With the second approach the programmer uses a declarative language in a notation we have proposed to specify the dependencies among transactions. A pre-compiler then translates this higher level description into a corresponding code in the high level language we use to specify transactions. The first approach gives more expressive power to the programmer than the second one and hence we will concentrate mostly on this approach; on the other hand the second approach is easier to use. In the following we show by an example, how the programmer can express these dependencies in our model to define coordinate blocks for a group of transactions using the high level language.

We assume that the programmer wants to design a multiform transaction consisting of four component transactions T_1, T_2, T_3 and T_4, each of which will be executed at different sites. The application requires that at most one of T_1 or T_2 commits with T_1 being preferred to T_2 and either both T_3 and T_4 commit or none of them do so. In short, one and only one of the following sets of transactions commits: $\{\}$, $\{T_1\}$, $\{T_2\}$, $\{T_3, T_4\}$, $\{T_1, T_3, T_4\}$ or $\{T_2, T_3, T_4\}$. Moreover, transactions T_3 and T_4 can commence execution only after T_1 and T_2 completes.

For this scenario, the programmer will develop the program fragment shown in figure 2. Note that although we use some of our new language primitives before they have been presented in the paper, a detailed understanding of the primitives is not required at this stage.

From the program fragment, we find that the multiform transaction consists of two coordinate blocks specified by the two *coordinate . . . using* delimiters. Each block contains the

```
void example()
{
        coordinate
        initiate(T1,T2) ;
            begin_trans (T1)
                    .
                    .
                    .
            end_trans (T1);
            begin_trans (T2)
                    .
                    .
                    .
            end_trans (T2);
        using
                                        thread end_trans (M) {
                                            if M == T1 then
                                                { commit (T1); abort (T2); exit; }
                                            if M == T2 then
                                                { commit (T2); abort (T1); exit; }
                                        }
                                        thread abort_trans (M) {
                                            if M == T2 then
                                            { abort (T1,T2) exit; }
                                        }
        end;
        coordinate
        initiate(T3,T4) ;
        cobegin
            begin_trans (T3)
                    .
                    .
                    .
            end_trans (T3);
            begin_trans (T4)
                    .
                    .
                    .
            end_trans (T4);
        coend
        using default
}
```

Figure 2. Program in the high level language for a multiform transaction consisting of four transactions.

code that implements the dependencies between those transactions that are defined within the blocks. In the figure, the coordinate block *coordinate . . . using . . . end* implements the dependency between transactions T_1 and T_2 (namely only one of T_1 or T_2 can commit with T_1 being preferred) while the block *coordinate . . . using default* implements the dependency among T_3 and T_4 (namely either both commit or none do). Note that the latter commit dependency is the standard commit dependency implemented in the various commit protocols (like Early Prepare). We assume that each transaction processing system implements a default commit protocol. The second coordinate block in the example in figure 2 specifies "default" as the coordinator module for transactions T_3 and T_4. Also note that the order of the two coordinate blocks ensures that T_3 and T_4 can commence execution only after both T_1 and T_2 completes.

Of interest to this discussion is the coordinate block for transactions T_1 and T_2 specified by the programmer in the form of a program fragment within the sub-block *using . . . end*. This program fragment contains definitions of some of the primitives that the programmer uses within the transactions.

The transactions T_1 and T_2 are defined sequentially within the multiform transaction (and not within a *cobegin . . . coend* block which would have implied concurrent execution) with T_1 being defined before T_2. This sequential definition of transactions naturally entails a precedence relation between these two transactions. Each transaction must be initiated by the initiate primitive before being able to start its execution. After a transaction is initiated, it is assigned a transaction identifier in the system and an environment is set up for its execution.

After the multiform transaction in figure 2 is submitted to the system, the TMA-TP module at the coordinating site assigns local transaction identifiers to the multiform transaction as well as to its components and then loads the main CM. Lightweight CMs for T_1 and T_2 are spawned by this CM at the remote sites. At the time these lightweight CMs are started up, the TMA-TP modules at these sites are asked to assign local transaction identifiers for T_1 and T_2. The TMA-TP module at the coordinating site then submits transaction T_1 to the remote site's TMA-TP which in turn begins to execute T_1. Note that the three CMs as a unit represent the interface to the three cooperating TMA-TP systems for the multiform transaction.

Suppose T_1 executes an end_trans; the CM at T_1's site sends a prepare-to-commit T_1 message to its underlying TMA-TP module and then invokes end_trans at the coordinating site's CM. (In figure 2 end_trans has been defined by the programmer in the coordinate block.) The CM at the coordinating site asks the CM for T_1 to invoke commit(T_1) and asks the CM for T_2 to invoke abort(T_2) at the respective underlying TMA-TP. The respective TMA-TP modules consequently force a commit log record for T_1 and an abort log record for T_2 and acknowledge the corresponding CMs. Also the TMA-TP module at the coordinating site forces appropriate log records for T_1 and T_2. This causes T_1 and T_2 to terminate. Note that the TMA-TP module can force an abort log record for T_2 although T_2 was never submitted to a remote site for execution. This is because when the lightweight CM for T_2 was established at the remote site of T_2, the TMA-TP subsystem there established a local identifier for T_2.

T_1 may alternatively execute an abort_trans command during its execution. This abort_trans command may have been invoked explicitly by T_1 or it may have been invoked by the TMA-TP because T_1 could not successfully complete. If the TMA-TP module at the remote site aborts T_1, the CM at the remote site informs the CM at the coordinating site by sending an abort_trans message to the CM at the coordinating site that T_1 has aborted. On the other hand if T_1 invokes abort_trans, the CM at the remote site forwards the invocation of abort_trans by T_1 to the CM of the coordinating site. The CM at the coordinating site executes abort_trans according to the implementation specified by the programmer in the thread abort_trans. The thread returns without executing any explicit abort (or commit) command and the coordinating CM does not send any specific instructions back to the CM at the remote site (it merely returns). As a result T_1 stops its execution but remains alive in the system until an explicit abort comes from the coordinator module to terminate it.[2] The TMA-TP module at the coordinating site now submits T_2 for execution at the remote site. Subsequently invocation of end_trans or abort_trans by T_2 will be trapped by the CM at the remote site and forwarded to the CM at the coordinating site for execution. If T_2 executes end_trans, the corresponding thread will commit T_2 and abort T_1. If, on the other hand, T_2 executes abort_trans, both T_1 and T_2 will be aborted. This will terminate the CMs for T_1 and T_2.

Once the CM for T_1 and T_2 terminates, the control gets transferred to the runtime support at the coordinating site which in turn initiates the default coordinator in the system (i.e. the default commit protocol) for T_3 and T_4. T_3 and T_4 proceed concurrently in the system as they are within a *cobegin* ... *coend* block. When T_3 and T_4 commit or abort the execution of the multiform transaction is over.

Note that in any coordinate block, we can refer to only those transaction identifiers that are in the scope of the block. In figure 2, transactions T_1 and T_2 are scoped in the coordinate block defined by the programmer but not T_3 and T_4. Consequently we can define end_trans and abort_trans only for T_1 and T_2 within this block.

The above execution is represented pictorially in figures 3(a) and (b). Note that for the description of the implementation of the coordinator for T_1 and T_2 in figure 2 we have assumed that CM_u in figure 3(a) submits T_1 and T_2 sequentially to each of the respective TMAs. Moreover CM_u does not submit T_2 if T_1 commits.

3. Primitives for flexible commit

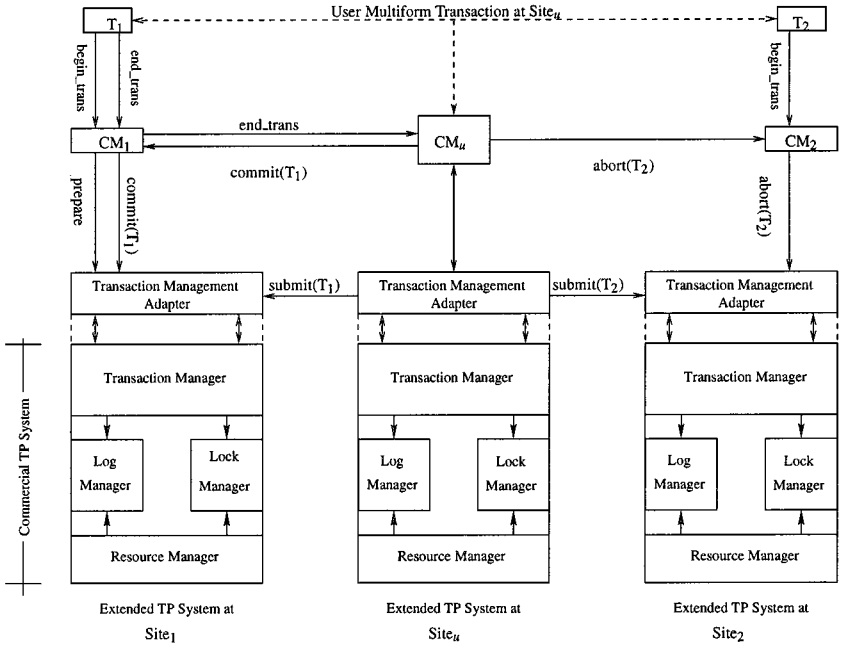
We now describe our transaction processing primitives. These primitives are essentially control primitives which modify the state of the transaction and are broadly classified into two types: basic primitives and new primitives. The basic primitives have been so named because quite frequently some kind of semantic counterparts to these are found in conventional transaction processing systems and even if these semantic counterparts do not match exactly with the definitions given here, they can be easily modified to support the functionality we desire here. The new primitives are the ones we define and that lend expressive power and flexibility to our model. Their default interface is provided by the transaction management adapters.

3.1. Basic primitives

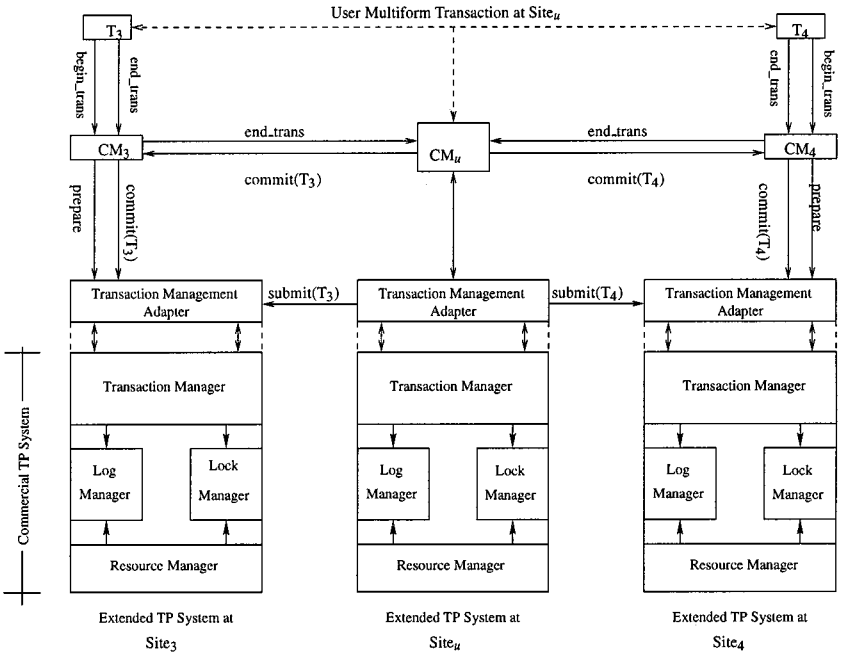
initiate($T_1, \ldots, T_n$) This primitive initiates the transactions $T_1, \ldots, T_n$. It returns new transaction identifiers in the variables $T_1, \ldots T_n$ and sets up the environment necessary for the execution of the transactions. The transactions are started by calling the begin_trans( ) primitive. The scope of the variables T_i used in an initiate primitive is the program block containing this initiate primitive. The initiate(T_i) primitive must precede all use of the variable T_i within a multiform transaction.

begin_trans(T_i) This primitive starts the execution of the transaction whose transaction identifier is T_i. This primitive can be redefined by the programmer.

sid = savework( ) The savework( ) primitive is used to establish a savepoint in the transaction execution. The invocation of this primitive causes the system to save the current state of processing. Each transaction manager writes a savepoint record on the local transaction log, while the current values of any local variables are saved on the stable storage. The savework call returns a handle which is assigned to the identifier sid (called a *savepoint identifier*). This identifier can be used subsequently to refer to that savepoint, and in particular to the state of the system when this savepoint was established. The scope of the binding between the savepoint and the identifier sid is the block in



(a)



(b)

Figure 3. Execution of a multiform transaction. (a) Execution of the programmer defined coordinator; (b) Execution of the default coordinator.

which the "sid = savework( )" is executed. Control can jump from inside of a block to a savepoint within an encompassing block, but not the other way round.

rollback(sid) This primitive takes as a parameter the identifier of a previously established savepoint and reestablishes (or returns to) the savepoint. More precisely, when the rollback(sid) function is invoked, it restores the state of the system to the state that existed when the savepoint denoted by the savepoint identifier sid, was established; the execution of the transaction then continues from the statements that follow the savepoint sid. The successful termination of the rollback primitive is indicated by the restoration of the savepoint denoted by sid. This primitive can only be invoked within a transaction code.

restart(T_i) This primitive is a part of the coordinator module and cannot be invoked by a transaction. When called, this primitive starts the execution of the transaction whose identifier is T_i. If the transaction T_i was previously executed (partially or fully) but not yet committed, then all changes effected by T_i are discarded before the transaction execution is restarted.

commit($T_1, \ldots, T_n$) This primitive is implemented in the TMA-TP module and is part of the commit protocol. It cannot be invoked directly by a component transaction. Rather, it has to be invoked by the coordinator module. This primitive commits the operations of the transactions which are its parameters, by first writing the log records and then communicating the commit decision to the transaction managers of these transactions. In other words, this command forms the final phase of any commit protocol between the coordinator and the transactions $T_1, \ldots, T_n$.

abort($T_1, \ldots, T_n$) This primitive aborts the transactions specified as parameters. If the primitive abort(T_i) is invoked before T_i has started its execution, then T_i never starts its execution and is discarded from the system. Like the commit primitive, abort is a part of the TMA-TP module and can be invoked only by the coordinator module.

cobegin . . . coend These two primitives act as bracketing constructs for specifying concurrently executing transactions. Control flow does not proceed beyond the cobegin . . . coend block until all of the transactions created by the block complete. Cobegin . . . coend can be nested.

3.2. New primitives

end_trans(T_i)⟨*support_code*⟩ The end_trans is a system defined primitive which can be redefined by the programmer as a coordinator module thread. Its execution signifies the successful completion of the transaction T_i, specified by a previous matching begin_trans(T_i), and indicates a willingness to commit the work of T_i. The programmer's definition of end_trans gets precedence over the default definition for the primitive.

If the programmer has not redefined the end_trans primitive, the default execution takes place. In this case the CM for T_i notes the completion of T_i; it asks the relevant TMA-TP module to force a prepare log record and sends vote messages relevant to the default commit protocol to the coordinator for the multiform transaction. The control flow does not proceed beyond the end_trans call, until the transaction manager for T_i receives either a commit or an abort decision. If the primitive is invoked without any parameter, then it commits the transaction within which it has been invoked.

As mentioned earlier, it is possible to overload this primitive to have a more flexible programmer-defined commit protocol. From transaction T_i's point of view the execution of the programmer-defined end_trans is the same as the default execution. That is the completion of the transaction T_i is recorded by a prepare log record and control is passed to the thread of the same name being executed at the coordinator. If the thread for end_trans does not contain an explicit invocation of the commit or abort primitives, the control proceeds beyond the transaction T_i after the thread completes execution and returns. However, the transaction T_i remains unterminated until an explicit invocation of commit or abort is eventually performed by the coordinator module for T_i.

Note that this primitive has two parts: end_trans(T_i) and *support_code*. The second part is an optional piece of program code which can be included by the programmer. This program code is not executed when the end_trans primitive is invoked. Rather, the coordinator module can direct the TMA-TP module for T_i to execute this program code by invoking the call_support primitive (explained next).

call_support(T_j, . . . ,T_m) This primitive can be invoked only as part of the programmer defined coordinator. With this primitive the coordinator module can direct the transaction managers of the transactions $T_j, \ldots, T_m$ to execute the support_codes specified as part of the corresponding end_trans primitives in these transactions. The program fragment for support_code of each T_k runs within the scope of T_k. If a support_code is invoked while the corresponding transaction is running, then the execution of the support_code is deferred until the transaction completes. The call_support returns to the invoking thread, when all the executing support_codes finish. This primitive along with the programmer specified support_code are useful in cases where the coordinator module wants to perform some task beyond merely committing/aborting after the transaction has executed an end_trans primitive. An example is shown later in Section 6.3.

abort_trans(T_i) The abort_trans is a system defined primitive which can also be redefined by the programmer as a coordinator module thread. In both cases, it signifies the unsuccessful completion of the transaction T_i, specified by a previous begin_trans(T_i), and indicates a decision to abort the work. However, the abort_trans does not actually abort the transaction. Rather the actual abort is performed by the abort primitive which is invoked by default and will always abort T_i. In case the abort_trans primitive is redefined by the programmer, the new definition gets preference over the default definition. If the thread for abort_trans does not contain an explicit invocation of the abort primitive, status of the transaction T_i remains unterminated until an explicit invocation of abort is eventually performed by the coordinator for T_i. The termination of the abort_trans primitive itself is similar to the end_trans primitive as explained above.

coordinate ⟨*transaction*⟩ **using** ⟨*protocol*⟩ This primitive defines the *coordinate block* whose partial syntax is described in Table 1. (Note that only the enhancements required in a standard programming language to support such a syntax is shown in the table. Anything not been defined is assumed to follow the syntax of the host language.) The coordinate block consists of two components: the *transaction* component and the *protocol* component. The protocol component defines the dependencies for the set of transactions specified in the transaction component. The protocol component can be the keyword default, in which case one of the traditional commit protocols like two-phase commit or early prepare is used. Or it can be a programmer specified dependency among the

Table 1. Partial syntax for a coordinate block.

Keyword		Syntax
coordinate-block	::=	**coordinate** transaction **using** protocol | **coordinate** coordinate-block **using** protocol
transaction	::=	**begin_trans**(trans-id) trans-command **end_trans(**trans-id**)** | **initiate**(trans-id) | **cobegin** transaction **coend** | transaction ; transaction
trans-command	::=	**abort_trans(**trans-id**)** | host-language-command
protocol	::=	**default** | declarativecode **end** | protocolcode **end**
declarativecode	::=	**dependency**{dependency expression}
dependency expression	::=	general dependency from Table 3
protocolcode	::=	protocol-command | protocolcode ; protocol-command
protocol-command	::=	**thread begin_trans**(parameter) thread-command | **thread end_trans**(parameter) thread-command | **thread abort_trans**(parameter) thread-command | **thread** identifier(formal-par-sequence) thread-command
thread-command	::=	**commit(**trans-ids**)** | **abort(**trans-ids**)** | **restart(**trans-ids**)** | **call_support(**trans-ids**)** | **exit** | thread-command;thread-command | host-language-command

transactions in the high level declarative language; or it can be a programmer defined code in which case it contains the code for each of the primitives that the programmer wants to define or redefine, including begin_trans(t), end_trans(t) and abort_trans(t). The scope of the redefined primitives is limited to the corresponding transaction component.

Within the coordinate block the programmer can define persistent variables which can live across the boundaries of transactions involved in the coordinate block. Persistent variables can also span multiple coordinate blocks of the same multiform transaction. However, the scope of such persistent variables is limited to the multiform transaction in which these have been defined. If the multiform transaction aborts or crashes, the persistent variables are no longer recovered. If the coordinate block, within which the persistent variable has been defined, crashes, then the persistent variable is not restored. Suppose that a persistent variable x is defined in some coordinate block C_1; it was modified in C_1, then in some subsequent coordinate block C_2 and currently is being used in a third coordinate block C_3. If now C_3 crashes then the persistent variable x will be restored to the state after the completion of C_2. These persistent variables are useful for flow control.

The control flow does not proceed beyond the coordinate block until either the transaction component completes or the coordinator module is terminated in a manner explained below.

thread *identifier* For efficiency and ease of implementation as daemons, the protocol component is programmed as a set of concurrently executing threads. The thread primitive allows the programmer to define a coordinator module thread which is activated by a transaction event. The identifier specifies the event which activates the thread. When a coordinate block is encountered, the coordinator module is created. It waits for any of the events named in its threads. When such an event occurs the corresponding thread is activated. If the thread encounters an **exit** command, the coordinator module terminates thereby causing the entire coordinate block to end. This is true even if there are transactions in the transaction component which are either yet to be executed or are currently executing concurrently. These transactions have to be taken care of by a subsequent coordinate block otherwise may lead to the problem of *orphan transactions*.[3]

On the other hand if an exit command is not encountered, the thread does not cause the coordinator module to terminate. Instead when the thread completes, it returns control to the transaction component. If an exit command is never encountered, the coordinator module terminates when all transactions have completed their execution and the coordinate block has terminated.

The execution of a thread is considered to be atomic.

exit Invocation of this command causes the termination of the coordinator module.

Table 2 summarizes the transaction primitives in our model. The first column of Table 2 lists the different primitives. The primitives have been grouped into three categories—(i) primitives that allow the structuring of the multiform transaction (ii) primitives that can be invoked from within the transaction component of a coordinate block and (iii) primitives that can be invoked from only within the protocol component of a coordinate block. The second column specifies where the programmer can use each primitive from viz., inside or outside a protocol component. The third column gives the system component that provides the interface to a particular primitive. When a primitive is invoked, this system component executes the primitive first. It in turn may invoke other system components in order to carry on the execution of the primitive.

The fourth column specifies which primitives provide a bracketing construct to specify nesting from the syntactic point of view. Finally the fifth column indicates whether a primitive can be redefined by the programmer in the coordinate block. Note that we allow only begin_trans, end_trans and abort_trans to be redefined in the current model of multiform transactions.

Of particular interest among the primitives in Table 2 are the eight primitives—initiate, begin_trans, end_trans, abort_trans, commit, abort, restart and call_support—all of which cause state transitions for a transaction. Figure 4 gives the state transition diagram for a transaction with respect to the above eight primitives.

When a multiform transaction is submitted by the user each component transaction enters the *declared* state. A declared transaction enters the *initiated* state when the initiate primitive is executed for the transaction. The transaction while executing its code, is in the *running* state. After it has executed all its code (either successfully or unsuccessfully), the transaction moves to the *completed* state. From the completed state, the transaction terminates by either moving to the *committed* state or to the *aborted* state.

Table 2. Summary of transaction primitives.

Primitive name	Invocation relative to protocol component	Interface exported by	Nested definition	Redefinition
coordinate . . . using	Outside	Language support	Yes	No
cobegin . . . coend	Outside	Language support	Yes	No
initiate	Outside	TMA-TP	–	No
sid = savework	Outside	TMA-TP	–	No
rollback	Outside	TMA-TP	–	No
begin_trans	Outside	CM or TMA-TP	Yes	Yes
end_trans	Outside	CM or TMA-TP	Yes	Yes
abort_trans	Outside	CM or TMA-TP	–	Yes
thread	Inside	CM	–	No
commit	Inside	TMA-TP	–	No
abort	Inside	TMA-TP	–	No
restart	Inside	TMA-TP	–	No
call_support	Inside	TMA-TP	–	No
exit	Inside	TMA-TP	–	No

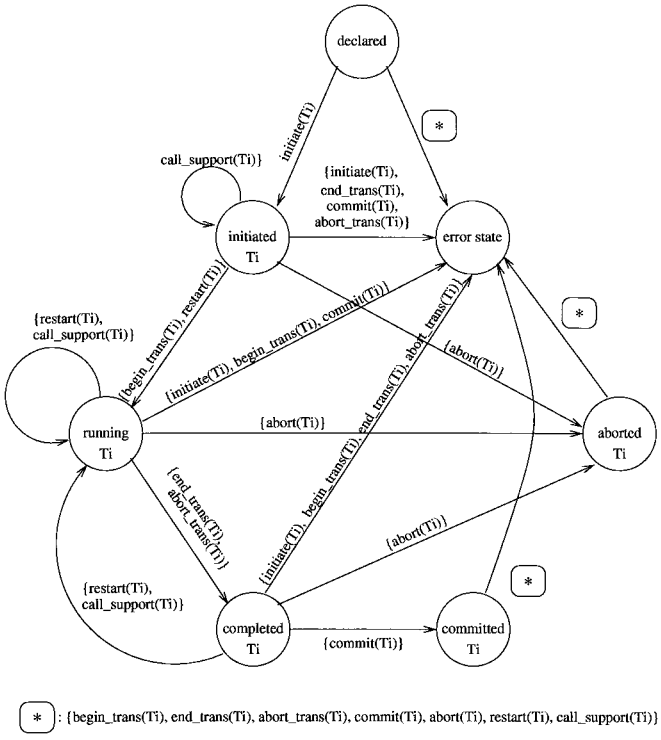


Figure 4. State transition diagram for a transaction.

Besides these six states there is an *error* state for a transaction. When the transaction manager detects that a transaction has moved into this state after invoking a primitive, the transaction manager ignores the primitive, and returns an error code to the transaction. The transaction can execute a particular error handler if provided by the programmer. Otherwise it can ignore the error code and continue the execution from the next instruction. Note that the circumstance behind a declared transaction T_i having to move to the error state can be detected at compile time by ensuring that the initiate primitive precedes the invocation of any other primitives by T_i.

4. Discussion

4.1. Orphan transactions

In our model, a transaction T_i, invoking a programmer defined end_trans or abort_trans thread at the coordinator, may not be terminated when the thread returns. This happens when the particular thread invoked by T_i does not in turn, invoke a commit or abort for T_i. Consequently, when the thread terminates its execution, T_i may remain active. Later on a second coordinator may execute a commit or abort command for T_i and terminate the transaction. Note that the second coordinator must be defined within the scope of the variable T_i to be able to commit or abort the transaction T_i and that only one coordinator can actually cause the termination of T_i.

The scope of every thread in a coordinator is limited to the corresponding transaction component. However, if the same thread is also defined in a nested coordinate block, the new binding overrides the previous binding. This scoping rule for the coordinate blocks ensures that every time a thread is invoked by a transaction, the definition of the thread closest to the invocation takes effect. For example, if a thread is defined twice—once within a coordinate block C and a second time within another coordinate block C' which is nested within C, then when the thread is invoked from within C', the definition of the thread within C' is in effect. After an end_trans or abort_trans has been executed for T_i, the very next coordinator that invokes a commit(T_i) or abort(T_i) will be able to commit or abort T_i (provided of course, that this invoking coordinator is within the scope of T_i). Moreover after T_i has terminated any subsequent invocation of commit or abort for T_i will result in an error condition. As part of the error handling, the relevant error handler will perform a null operation on the database and generate warning messages.

Assume now that a multiform transaction is not properly designed. Then a possibility remains that the transaction T_i is never explicitly terminated by any coordinator within the multiform transaction and thus remains active at the end of the multiform transaction. Such a transaction is called *orphan transaction*. When a transaction T_i is orphan the locks acquired by T_i are not released and the updates made by T_i are not made permanent. This may cause a number of problems like deadlock or unsatisfiable dependencies. For this reason a strategy is usually adopted in which all orphan transactions are aborted as soon as detected. However this strategy is not completely satisfactory as the work of the orphan transaction is wasted. This is more of a problem because often an orphan transaction is the result of a program error which could have been avoided by some compile time check.

In order to avoid the problem of orphan transactions we introduce the notion of a *well-formed* multiform transaction. A compiler can check a multiform transaction for well-formedness and can ensure that orphans do not result from a program error. Intuitively, a well-formed multiform transaction is one that does not leave any orphan transactions after its execution.

To introduce the notion of well-formed multiform transaction we use the concept of a control flow graph of a coordinate block. The control flow graph is a graph whose nodes represent events of the following types:

- Initiation—This node represents the fact that a begin_trans(T_i) is encountered in the code of the coordinate block and is labeled by T_i Begin;
- Thread invocation—This node represents a thread named in the transaction code and is labeled by the name of the thread followed by the transaction invoking the thread; for e.g. end_trans(T_i), abort_trans(T_i) etc.;
- Decision—This node represents the decisions taken by a particular thread and is labeled by the set of decisions taken viz., commit(T_j) or abort(T_j);
- Termination—This node represents the end of the coordinate block C. The end of C is specified either by the execution of an exit command in a thread defined in the protocol component or by the end of the last thread invoked by the last transaction in the transaction component of C. The node is labeled correspondingly, either by exit or by done.

There is an edge from one node N_i to another node N_j in the control flow graph if N_i may potentially cause N_j. For example if a transaction T_i contains conditional invocations of abort_trans(T_i) and end_trans(T_i) then the corresponding control flow graph contains one initiation node T_i Begin, with two children labeled end_trans(T_i) and abort_trans(T_i). Moreover, when a cobegin command is encountered after an event ε in the code of the coordinate block, a subgraph is appended to ε. This subgraph represents all the possible interleavings of the concurrent transactions within the corresponding cobegin ... coend block. This is sufficient to model concurrency among transactions that invoke the same coordinator. The reason for this is that the control flow graph does not describe the execution of the transactions but rather the coordinator's flow. Since concurrent transactions specified within a cobegin ... coend block invoke the same threads at the same coordinator, these invocations must be interleaved in some fashion.

Let a *control flow path* be any path in a control flow graph starting from a node of type initiation. Then a *termination path* is defined as follows:

Definition 2 (Termination Path). A *termination path* of a coordinate block C is one of the possible control flow paths ending in a termination node. Such a path describes a possible termination of the coordinator C.

Figure 5 is a control flow graph for the coordinate block involving transactions T_1 and T_2 of figure 2. It shows the three termination paths for the programmer defined coordinator for transaction T_1 and T_2.

Definition 3 (Cover). Let T and T' be two sets of transactions such that $T \subseteq T'$. A coordinate block C in the scope of the set of transactions T', *covers* T (alternatively covers

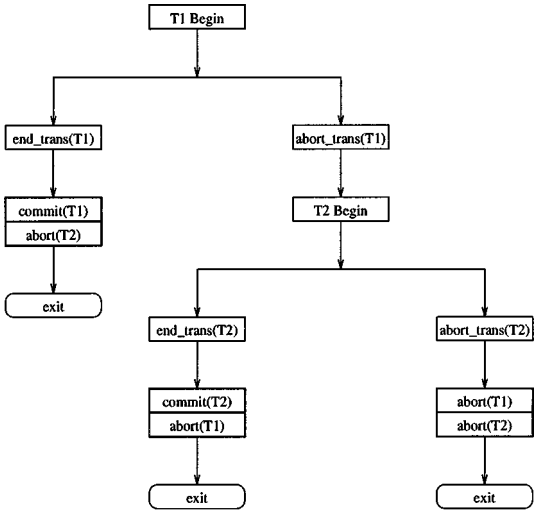


Figure 5. Control flow graph of coordinator for transactions T_1 and T_2 in figure 2.

all transactions in T) if every termination path of C contains either a commit command or an abort command for each $T_i \in T$. The set of transactions T covered by C is denoted by T^C.

At compilation time it is possible to determine whether a particular coordinate block covers a set of transactions T by performing a static analysis [18] of the coordinate block. Referring to figure 2 a compiler can generate a control flow graph of the coordinate block for transactions T_1 and T_2 similar to that shown in figure 5. Then the compiler can find out that the programmer defined coordinator covers transactions T_1 and T_2. This is because there are three termination paths ending with the exit command, each of which contains either a commit or an abort for both T_1 and T_2.

Note that all transactions $T_i \in T$ need not be defined within the transaction component of the coordinator C. C can cover transactions defined in the transaction component of another coordinator C'. C need only to be in the scope of the variable T_i in order to be able to cover the transaction T_i. Further a set of transactions may be covered by the system default coordinators. For example in figure 2 the system default coordinator covers transaction T_3 and T_4. Note that if a transaction is defined within a coordinate block but is not covered by any programmer's coordinator code, then it is not covered by the system default coordinator also.

Definition 4 (Branch). A *branch* of a multiform transaction is one of the possible execution sequences of the multiform transaction which ends with the multiform transaction's termination and along which the control flows in the multiform transaction.

Definition 5 (Well-formed Multiform Transaction). A multiform transaction is *well-formed* if in every branch B of the multiform transaction one or more coordinator blocks

$C_1, \ldots, C_q$ are encountered such that $T^{C_1} \cup T^{C_2} \cdots \cup T^{C_q}$ contains all the transactions defined in the blocks encountered in B.

Note that if a coordinator C does not cover a transaction T_i defined in the transaction component of C, then the multiform transaction can still be well-formed provided there is a second coordinator C' that covers T_i. Moreover, multiple invocation of commit or abort for a transaction T_i can be specified in the multiform transaction. Usually this occurs if T_i is to be conditionally aborted or committed. In such cases, the first execution at runtime of either primitive takes effect while the others, if executed subsequently, perform only null operations and generate warning messages.

It can be determined at compile time whether a multiform transaction is well-formed or not. To do this, the compiler can statically analyze a multiform transaction and determine whether the set of coordinators defined in the multiform transaction covers the full set of transactions defined in it. Referring to figure 2 we find that the multiform transaction is well-formed. T_1 and T_2 are covered by the programmer defined coordinator as shown in figure 5, while T_3 and T_4 are covered by the system default coordinator.

The multiform transaction shown in figure 6(a) is not well-formed as it results in the control flow graph shown in figure 6(b). In this control flow graph there is a termination path that does not contain abort(T_1) or commit(T_1) viz., T_1Begin $\rightarrow$ abort_trans(T_1) $\rightarrow$ T_2Begin $\rightarrow$ end_trans(T_2) $\rightarrow$ commit(T_2) $\rightarrow$ done.

4.2. Recovery for a multiform transaction

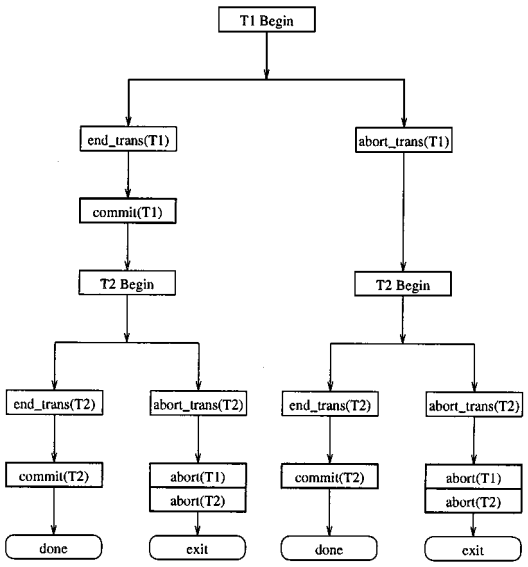
Recovery of a multiform transaction is different from that of a conventional transaction. In the classical transaction processing system if a crash destroys the code of a running transaction, the system log is sufficient to undo the effects of the transaction. However, in the case of multiform transactions, it is necessary to complete the remaining transactions including the one (if any) that was executing at the time of the crash. This requires the code of the multiform transaction to survive the crash.

Note that there is a distinction between the failure of the multiform transaction and the failure of one of its component transactions. In case of the failure of one of the component transactions T_i (which is the same as the transaction executing an abort_trans primitive), the corresponding coordinator performs the necessary recovery operations for T_i. Our concern is more with the failure of the multiform transaction in which the coordinator modules crash. This requires the code of the multiform transaction to be stored in the database.

To achieve this recovery process, we assume that the database system can store code as any other data item. When a multiform transaction T is submitted by the user an initiating transaction I_T enters into the database the code of all the component transactions and the coordinators specified in the multiform transaction and then commits. After I_T commits the rest of the multiform transaction is ready to start. When the multiform transaction ends its execution the system does not right away forget about T. At this point a terminating transaction E_T is initiated to remove the different codes of T that were entered earlier by I_T, from the database. If I_T aborts it is recovered using conventional recovery techniques. Since the multiform transaction does not start until after I_T commits, semantically the abort

```
void notwellformed()
{
        initiate(T1,T2) ;
        coordinate
            begin_trans (T1)
                    .
                    .
                    .
            end_trans (T1);
            begin_trans (T2)
                    .
                    .
                    .
            end_trans (T2);
        using
                                thread end_trans (M) {
                                        commit (M);
                                }
                                thread abort_trans (M) {
                                    if M = T2 then
                                        {abort (T1); abort (T2); exit }
                                }
        end
}
```

(a)



(b)

Figure 6. Example of (a) a not well-formed transaction; (b) its control flow graph.

of I_T is equivalent to the abort of the multiform transaction. The user gets a response similar to failure to start the multiform transaction and has to resubmit the multiform transaction if it is to be executed. On the other hand, if E_T gets aborted, there is no need to abort the multiform transaction T. However the system does not forget about T until after E_T commits. E_T is recovered and automatically restarted when the system comes up.

If there is a failure of the multiform transaction T, the recovery subsystem determines which of the component transactions have committed and which coordinator was running at the time of the crash. It undoes the transactions that were not committed at the time of the crash. All persistent variables defined within a coordinate block are also restored by the recovery subsystem. Since the code of T is stored in the database, the multiform transaction can then continue from the committed state just before the crash.

4.3. Multiple saveworks

In the course of a transaction execution, a sid = savework( ) primitive may be executed more than once. In such cases it is preferable to assign the new handle generated by the system, to different savepoint identifiers every time a savework primitive is invoked. Otherwise the transaction looses the ability to refer to the savepoints previously established but associated to the same sid.[4] For example suppose the programmer wants to undo the effects of a loop based on certain conditions established during the execution of the loop. If a savepoint is established within the loop and the same savepoint identifier is employed, the programmer can undo only the latest iteration of the loop. This is due to the fact that while re-using the same identifier for different handles, the programmer looses reference to the previous savepoint handles. If the programmer establishes a savepoint immediately before the loop, the effects of all the iterations of the loop can be fully undone. Recall that the scoping rules of the savepoint identifier is the whole block in which the sid = savework( ) primitive is executed and hence a rollback command within the loop can refer to a savepoint identifier outside the loop.

4.4. Restart and rollback

Semantically the restart(T_i) command is equivalent to executing a rollback to the beginning of the transaction. This rollback can be achieved by establishing a savepoint corresponding to the execution of the begin_trans(T_i). However, there is one important difference between the restart primitive and the rollback to the beginning of a transaction. The restart primitive can be executed only by the coordinator; a transaction cannot restart itself. The rollback primitive, on the other hand is invoked by the transaction itself. The coordinator does not have any idea about savepoints established by a transaction and hence is not allowed to execute a rollback primitive.

4.5. Other issues

In the multiform transaction model, a coordinator for a transaction T_i can multiply invoke commit or abort primitives for T_i. Usually this occurs if T_i is to be conditionally aborted or committed. In such cases, the first execution at runtime of either primitive takes effect while the others, if executed subsequently, performs only null operations and generates warning messages. Further, the initiate command can be invoked from both outside or inside a coordinate block. If it is invoked from outside a coordinate block then the scope

of the identifier T_i specified in the invocation of initiate is the entire program code for the multiform transaction; else the scope is limited only to the particular coordinate block from which initiate is invoked. In the former case we can have a number of coordinate blocks for a single transaction T_i defined within the scope of the identifier T_i; however, at most two coordinators can actually be involved for terminating T_i. The scoping rules ensure that every time an end_trans or an abort_trans is invoked, it gets bound to only one thread, viz. to the thread which is defined at point closest to the invocation. Hence, the closest coordinator will execute end_trans (or abort_trans) without committing or aborting T_i and a second will perform the actual commit or abort operation.

5. Deadlock detection within a multiform transaction

Deadlock can occur among the component transactions of a single multiform transaction, for a variety of reasons:

1. When the component transactions are concurrent and they share common data, the concurrency control mechanism used to synchronize access to the shared data can cause deadlock. For example strict two phase locking can cause deadlock.
2. If conflicting completion dependencies have been specified for a set of transactions, it can lead to deadlock. For example, for a set of transactions $\{T_1, T_2\}$, the completion dependency—T_1 can commit only if T_2 commits and T_2 can start execution only after T_1 commits—will cause a deadlock.
3. Deadlock can occur among the component transactions by the interplay of data dependency and completion dependency among these transactions.

Note that in the following discussion we are interested only with deadlock that can occur within a multiform transaction. Since there are no completion dependencies among component transactions of different multiform transactions, the only way deadlock can occur among different multiform transactions is by way of sharing data. We do not consider this issue since there are adequate techniques that take care of this matter. This is true for the deadlock that occurs by way of the method (1) above. Also deadlock as outlined in method (2) can be avoided by a simple static analysis of the control flow in the multiform transaction. Of greater concern is the deadlock occurring by method (3) which is more difficult to analyze and resolve.

Consider a well-formed multiform transaction composed of two concurrent transactions T_1 and T_2 such that

- there is a data dependency between T_1 and T_2 on a shared data item x such that T_1 writes x which is later on read by T_2 and
- there is the following completion dependency between T_1 and T_2—T_1 commits only if T_2 is committed.

If T_1 and T_2 are concurrent then the following deadlock situation may arise:

```
void deadlock()
{
        initiate(T1,T2,T3) ;
        coordinate
            begin_trans (T1)
                  w[x] ;
            end_trans (T1);
            cobegin
                begin_trans (T2)
                   r[x] ;
                end_trans (T2);
                begin_trans (T3)
                   ...
                end_trans (T3);
            coend
        using
                        completeSet = null ;    % persistent variable
                        thread end_trans (T) {
                              completeSet := union (completeSet,T) ;
                              if subseteq({T2,T3},completeSet) then
                                    { commit(T1,T2,T3) ; exit ; }
                        };
                        thread abort_trans (T) {
                              completeSet := union (completeSet,T) ;
                              abort (T) ;
                        }
        end }
```

Figure 7. Example in which a deadlock occurs.

1. T_1 receives write lock on x before T_2 and completes its execution (but T_1 is not committed and hence does not release its locks);
2. T_2 waits for T_1 to release the write lock on x before being able to proceed with its execution;
3. T_1 cannot commit and release its write lock on x as T_2 has not yet committed or aborted.

Conventional deadlock detection algorithms will fail to detect the deadlock. This is because the conventional algorithms consider only the data dependency and cannot cope with the interaction between data dependency and completion dependency. Moreover, the dependencies in a multiform transaction may be more complex than the one given in this example. Dependencies can exist not only between two transactions but also between two sets of transactions. This is shown in the example in figure 7.

The example in figure 7 illustrates a different completion dependency among three component transactions T_1, T_2 and T_3 which results in a deadlock. In this example, T_1 and T_2 share a common data item x while T_3 does not share any data item with any of these transactions. T_1 can terminate only after both T_2 and T_3 have committed.

This causes a deadlock. However, if the protocol component of this multiform transaction is modified to give a different completion dependency as that shown in figure 8, then we no longer have a deadlock. In figure 8 transaction T_1 can terminate after either T_2 or T_3 has committed. In those transaction examples and some other subsequent examples, a boolean function subseteq(a, b) has been used. This function takes two sets a and b as parameters and returns true if $a \subseteq b$. Moreover, the persistent variable completeSet has been declared within the protocol component; in the subsequent examples of multiform transaction all

```
void no-deadlock()
{                          ...
        using
                        thread end_trans (T) {
                            completeSet := union (completeSet,T) ;
                            if subseteq({T2},completeSet)
                            OR subseteq({T3},completeSet) then
                                commit(T1, T) ;
                        };
                        thread abort_trans (T) {
                            completeSet := union (completeSet,T) ;
                            abort (T) ;
                        }
        end; }
```

Figure 8. No deadlock.

variables used in a protocol component will be assumed to be persistent and not explicitly declared.

The reason for giving these examples is to emphasize the point that it is more difficult to detect deadlock when completion dependencies are also involved besides ordinary data dependencies. Moreover, completion dependencies are not specified explicitly in our model but in the form of program code.

Once a deadlock is detected, the usual strategy to break it is to abort any one of the involved transactions. With completion dependencies, a good strategy is to abort and restart the transaction which was supposed to commit last as this will allow the multiform transaction to terminate with success. For the example involving only T_1 and T_2, we can abort T_1 so that T_2 will be able to commit, then restart and complete T_1.

In the following, we address this problem of deadlock detection within a multiform transaction and show how the conventional waits-for-graph can be extended to detect a deadlock.

To detect possible deadlocks within a multiform transaction we extend the notion of a waits-for-graph to include completion dependencies. This extended waits-for-graph is an instance of what is known as an AND-OR graph. We assume that the completion dependencies among the components of the multiform transaction can be obtained by a static analysis of the coordinate blocks and the dependencies so obtained are specified in the form of an AND-OR graph. The exact description of how such a static analysis can be derived, is beyond the scope of this paper. For work related to program analysis the reader is referred to [18]. The AND-OR graph generated by the static analysis of the coordinate blocks forming the multiform transaction T, has the following characteristics:

- Each node of an AND-OR graph represents a component transaction T_i of T.
- If D, a subset of the set of transactions composing T, is such that T_i commits or aborts only after all $T_j \in D$ are committed or aborted, then there is an AND edge in the graph from T_i to every $T_j \in D$.
- If D', a subset of the set of transactions composing T, is such that T_i can commit or abort after any $T_j \in D'$ commits or aborts, then there is an OR edge in the graph from T_i to every $T_j \in D$.

An AND edge from T_i to every transaction T_j in D implies that T_i can commit or abort after all $T_j \in D$ have terminated. Alternatively, an OR edge from T_i to every T_j in D' implies that T_i can commit or abort after any $T_j \in D'$ has terminated. With reference to the example in figure 7, there is an AND edge from T_1 to T_2 and T_3, while in the example in figure 8 there is an OR edge from T_1 to T_2 and T_3.

During the execution of the multiform transaction T, the AND-OR graph is modified as follows to give the extended waits-for-graph:

- A data dependency edge is included in the AND-OR graph from transaction T_i to transaction T_j, if T_i is waiting for T_j to release a conflicting lock on a shared data item. When the lock is granted to T_i, this data dependency edge is removed from the graph.
- If a transaction T_j has terminated, then the node T_j is removed from the AND-OR graph together with the following edges:
  - If there is an OR edge from some T_i to T_j, then *all outgoing edges* of T_i are removed except the data dependency edges of T_i.
  - If there is an AND edge from T_i to T_j, then *only this edge* between T_i and T_j is removed.

A cycle in a AND-OR graph does not however, represent a deadlock. In fact as reported in [15], classical graph theory does not provide a construct to describe a deadlock situation in an AND-OR graph. Deadlock in the AND-OR graph can be detected by the repeated application of the deadlock detection algorithm for an OR graph (i.e. a graph with only OR edges and no AND edges), exploiting the fact that deadlock is a stable property, i.e. it does not go away by itself. If only OR edges are considered then a set of transactions $S \subseteq T$ is deadlocked if

1. all transactions in S are blocked, i.e. these transactions have outstanding OR edges and
2. the set of transactions on which every transaction in S is dependent is a subset of S;

Deadlock in an OR graph is represented by a *knot*. A node T_i is in a knot if for every node T_j such that T_j is reachable from T_i, T_i is reachable from T_j. Although finding a knot is a polynomial time problem, the repeated application of this algorithm to find a deadlock in an AND-OR graph is in general not very efficient. Recently in [3] a linear time algorithm to detect deadlock in an AND-OR graph is presented and proven correct by characterizing the above deadlock problem in a subclass of Petri nets equivalent to AND-OR graphs.

6. Realizing various transaction completion dependencies

We now examine the expressive power of the Multiform transaction model. This section shows how the different transaction completion dependencies in various extended transaction models can be specified as multiform transactions. We would like to emphasize that in the following discussions we capture only the transaction completion dependencies. We assume that we have transaction adapters similar to [2] that capture data dependencies. We begin with the SAGA model which is among the simpler models.

6.1. SAGAS

SAGA [11] is a transaction model that provides system support for the execution of long-lived transactions. In saga a long-lived transaction is executed as a number of shorter subtransactions without sacrificing the atomicity of the larger transaction.

A saga consists of a set of flat transactions $T_1, T_2, \ldots, T_n$ that execute sequentially within the context of the saga, but can interleave arbitrarily with component transactions of other sagas. For each transaction T_i $(1 \leq i < n)$ there is a compensating transaction CT_i, which, when executed, semantically undoes the effects of T_i. A compensating transaction CT_i is executed if and only if the transaction T_i has committed and the saga of which T_i is a part, has aborted. A saga commits if all the transactions T_i's successfully commit and aborts if any of the T_i's aborts. If a saga aborts, it compensates for the effects of all committed components T_j's, by executing their corresponding compensating CT_j's. The compensating transactions are executed in the reverse order of the commits of the corresponding T_i's. Note that there is no compensating transaction for the last component transaction T_n. This is because if T_n commits then the entire saga commits. The final outcome of a saga is either the sequence:

1. $T_1, T_2, \ldots, T_{n-1}, T_n$ if all T_i's commit or
2. $T_1, T_2, \ldots, \underbrace{T_i}_{abort}, \mathrm{CT}_{i-1}, \ldots, \mathrm{CT}_2, \mathrm{CT}_1$ if any T_i aborts.

Other transactions may see the effects of a partial saga execution.

In figure 9 we show how the semantics of a saga can be achieved with our primitives. The saga program consists of one coordinate block which controls the execution flow of the transactions $T_1, \ldots, T_n$ and the corresponding compensating transactions $\mathrm{CT}_{n-1}, \ldots, \mathrm{CT}_1$. In the transaction component of the coordinate block, the T_i's and the CT_j's (if so required) are executed sequentially. If T_n successfully completes, then the coordinator aborts $\mathrm{CT}_{n-1}, \ldots, \mathrm{CT}_1$ (as no compensation is required) and the saga terminates successfully. On the other hand, if any of the T_i's aborts the thread abort_trans(T_i) in the coordinator is executed, which aborts the transaction $T_i, \ldots, T_n$ as well as the compensating transactions $\mathrm{CT}_{n-1}, \ldots, \mathrm{CT}_i$. In this way the transactions remaining to be executed, viz., $\mathrm{CT}_{i-1}, \ldots, \mathrm{CT}_1$ become exactly those required to compensate the effects of the already committed transactions $T_1, \ldots, T_{i-1}$. If any of these CT_k's aborts, the thread abort_trans(CT_k) in the coordinator is executed, which in turn restarts the compensating transaction CT_k. In this way the effects of all the committed transactions are compensated for and the saga aborts.

Note that in this example the transaction execution order is very simple—being just a sequence of transactions and their corresponding compensating transactions. The specification of the completion dependency in the protocol component implements the actual saga model. This example illustrates the usefulness of separating the coding of the transactions from the definition of the transaction dependency. The programmers work can be simplified because transactions can be coded without worrying about managing dependencies among them.

```
void saga ()
{ initiate (T1,T2,...,Tn,CT1,CT2,...,CTn-1);
        coordinate
            begin_trans (T1)
                ...
            end_trans;
            begin_trans (T2)
                ...
            end_trans;
            .
            .
            .
            begin_trans (Tn)
                ...
            end_trans;
            begin_trans (CTn-1)
                ...
            end_trans;
            .
            .
            .
            begin_trans (CT2)
                ...
            end_trans;
            begin_trans (CT1)
                ...
            end_trans ;
        using
                        thread end_trans (M) {
                            commit (M) ;
                            if M == Tn then {abort(CT1,...,CTn-1); exit };
                        }
                        thread abort_trans (M) {
                            case (M) do
                            T1: { abort(T1,...,Tn, CT1,CT2,...,CTn-1 ); exit} ;
                            T2: abort(T2,...,Tn,CT2,...,CTn-1) ;
                            T3: abort (T3,...,Tn,CT3,...,CTn-1) ;
                            .
                            .
                            .
                            Tn: abort(Tn) ;
                            CT1: restart(CT1) ;
                            CT2: restart(CT2) ;
                            .
                            .
                            .
                            CTn-1: restart(CTn-1) ;
                        }
        end;
}
```

Figure 9. Implementation of a saga.

6.2. Workflows and long lived activities

The atomicity property of transactions prevents any internal structure to be perceived and referred to from outside of the transaction. Consequently if there is an activity that consists of multiple steps of processing with an explicit flow of control among these steps, it is next to impossible to model it as a transaction. Note that modeling any action as a transaction facilitates data sharing, persistence and failure recovery.

Workflows have been suggested as a way of implementing long-lived activities which have some kind of an internal structure, in terms of shorter transaction like components

[8, 20]. Workflows allow dependencies among transactions to be expressed and also allows correctness requirements among the component transactions that are less stringent than serializability or atomicity.

We show by an example how a workflow can be expressed by our primitives. The example workflow involves planning for a trip by John Doe. He plans to leave on the 3rd of June by either Delta or United or American in that preference order, stay at the hotel Ambassador from the 3rd till the 6th of June and rent a car from either National or Avis with no preference. If any of the reservations (i.e. flight, hotel or car) is unavailable, John Doe would like to cancel his trip.

In the example in figure 10 the different components flightReservation, hotelReservation, carReservation, cancelFlightReservation and cancelHotelReservation perform the actual reservation or cancelation operations. The single coordinate block for the workflow contains the transactions $T_1, \ldots, T_6$ and the compensating transactions CT_1, CT_2. CT_1 compensates for any committed flight reservation achieved by either T_1 or T_2 or T_3 in case either the hotel reservation or both of the car reservation do not fall through. CT_2 compensates for a committed hotel reservation if neither of the car reservation is successful.

Every time a transaction completes, it invokes the end_trans thread at the coordinator which then enforces the control flow of the activity. The successful completion of the workflow is indicated by the commit of either T_5 or T_6. In this case the coordinator ensures that CT_1 or CT_2 are aborted.

If any transaction decides to abort, it invokes the abort_trans thread at the coordinator. The execution of the abort_trans thread for a transaction T_i, aborts all transactions T_j that follow T_i in the workflow and compensates for the committed T_k's preceding T_i in the workflow. In case any compensating transaction CT_i is aborted, it has to be re-executed until it successfully completes.

6.2.1. Semiatomicity. A formalization of the workflow model is provided in [22]. In this paper a workflow is synonymous to a *flexible transaction*. The structure of a flexible transaction T, is viewed as a set of the so called *representative partial order* of subtransactions. The subtransactions within a representative partial order are related by the *precedence* relation. Each representative partial order gives an alternative for the execution of the flexible transaction. There is also a *preference* relation which defines the preferred order of the alternatives. Each subtransaction is categorized as either *retriable*, *compensatable* or *pivot*.

The execution of a flexible transaction T preserves the property of *semiatomicity* if one of the following conditions is satisfied:

1. All its subtransactions in one representative partial order commit and all attempted subtransactions not in the committed representative partial order are either aborted or have their effects undone.
2. No partial effects of its subtransactions remain permanent in local database.

In [22] the authors provide a commit protocol which dynamically commits subtransactions as soon as possible. An alternative representative partial order is executed if an attempted subtransaction aborts. In this case subtransactions which have already been committed in the failed representative partial order are compensated for.

```
void workflow ()
{
        initiate(T1,T2,T3,T4,T5,T6,CT1,CT2)
        coordinate
        airline* air;           % persistent variable
            begin_trans(T1)
                flightReservation(Delta, 6/3/96) ;
                air = Delta ;
            end_trans(T1) ;
            begin_trans(T2)
                flightReservation(United, 6/3/96) ;
                air = United ;
            end_trans(T2) ;
            begin_trans(T3)
                flightReservation(American, 6/3/96) ;
                air = American ;
            end_trans(T3) ;
            begin_trans(T4)
                hotelReservation(Ambassador, 6/3/96, 6/6/96) ;
            end_trans(T4) ;
            cobegin
                begin_trans(T5)
                    carReservation(National, 6/3/96, 6/6/96) ;
                end_trans(T5)
                begin_trans(T6)
                    carReservation(Avis, 6/3/96, 6/6/96) ;
                end_trans(T6)
            coend;
            begin_trans(CT1)
                cancelFlightReservation(air, 6/3/96) ;
            end_trans(CT1);
            begin_trans(CT2)
                cancelHotelReservation(Ambassador, 6/3/96, 6/6/96) ;
            end_trans(CT2);
        using
                        thread end_trans (M) do {
                            case M of {
                                T1 : { commit (T1); abort (T2,T3);} ;
                                T2 : { commit (T2); abort (T1,T3);} ;
                                T3 : { commit (T3); abort (T1,T2);} ;
                                T5 : { commit (T5); abort (T6,CT1,CT2); exit;} ;
                                T6 : { commit (T6); abort (T5,CT1,CT2); exit;} ;
                                default: commit (M);    %  commit T4 or CT1or CT2
                            };
                        };
                        thread abort_trans (M) do {
                            noSet = union (noSet,M) ;
                            if subseteq({T1,T2,T3},noSet) then
                                { abort (T1,T2,T3,T4,T5,T6,CT1,CT2); exit }
                            if subseteq(T4,noSet) then
                                abort (T4,T5,T6,CT2);
                            if subseteq({T5,T6},noSet) then
                                abort (T5,T6);
                            if M==CT1 or M==CT2 then
                                restart(M);
                        };
        end ;
}
```

Figure 10. Workflows: reservations.

Given an instance of a flexible transaction we can implement a coordinator for this flexible transaction in a manner similar to implementing a workflow, shown previously. In fact a compiler can be developed to generate the codes for such a coordinator, given an appropriate description of the flexible transaction with the different precedence and preference relations.

6.3. Secure distributed transactions

The classical Early Prepare commit protocol (EP), used in many commercial systems, is not suitable for use in multilevel secure distributed databases systems (MLS) that employ a locking protocol for concurrency control. A major problem of the locking protocols in MLS systems is that in order to prevent illegal information flow through a timing covert channel, any read locks acquired by a high level transaction on lower level data, must be released whenever a low transaction attempts to acquire write locks on the same data items. This may lead to a transaction no longer remaining two phased. A secure two phase locking protocol has been proposed in the literature [4] to cope up with this problem. However even when using a secure locking protocol, serializability can still be violated during early prepare commit of distributed MLS transactions. During EP commit read locks may have to be released within a transaction's *window of uncertainty*—the period after a participant has voted yes to commit, but before it receives the commit or abort decision from the coordinator—possibly resulting in nonserializable executions [14].

A solution to the above problem has been suggested in [1]. In this work, a secure EP commit protocol (SEP) has been proposed which implements the following *secure commit* dependency in addition to the conventional commit/abort dependencies for distributed systems.

Definition 6 (Secure Commit Dependency). Given a multilevel secure distributed transaction T, the secure commit dependency among all possible pairs $\langle T_i, T_j \rangle$ of subtransactions of T, is defined as follows:

1. if either T_i or T_j releases any of its low read locks within its window of uncertainty and before the other subtransaction completes, then both T_i and T_j abort.

The secure dependency between subtransactions T_i and T_j is denoted by $T_i \xrightarrow{s} T_j$.

In other words, to prevent nonserializable executions, a secure early prepare commit protocol aborts the distributed higher level transaction when any of the lower reading participants is compelled to release its lower level read-locks within its window of uncertainty before the other subtransactions complete. Further, either all the subtransactions abort or all of them commit.

The secure dependency, $T_i \xrightarrow{s} T_j$, between transactions T_i and T_j, is an example of a transaction dependency that cannot be implemented by any of the extended transaction models proposed in the literature. The reason for this is that these models do not have the expressive power to model significant events in the system other than the commit or abort of transactions.

The SEP protocol can be implemented as a multiform transaction by incorporating the primitive getSignal described in [19]. In that work, the authors introduce the notion of *signals* which are raised by the lock manager when it allows a write lock (requested by a low level transaction) to be placed on a low level data item even though the item is already locked in read mode by a higher level transaction. The getSignal primitive is used by the

programmer to specify how signals from lower level transactions are to be serviced by the higher level transaction. The default invocation for the getSignal primitive is:

getSignal[→ *handler*].

The handler represents a piece of code which is to be executed if there is any signal to be serviced.

By using the getSignal primitive it is possible to implement a commit protocol which enforces the secure dependency among subtransactions of a multilevel secure distributed transaction T. Figure 11 shows a multiform transaction that implements the SEP protocol for committing three concurrent subtransactions T_1, T_2 and T_3. These three subtransactions are related to each other by the secure dependency $T_i \xrightarrow{S} T_j$ $(1 \leq i, j \leq 3, i \neq j)$. In the example, transactions T_1 and T_3 read data at lower security levels, whereas T_2 does not. We assume there is a secure two phase locking protocol that provides concurrency control among the transactions.

```
void secure_distributed_commit ()
{
        initiate (T1,T2,T3) ;
        coordinate
        cobegin
            begin_trans (T1)
                r[x] ;
                sl2 = SaveWork() ;
                w[z] ;
                r[y] ;                          /* this is a low read */
                sl3 = SaveWork() ;
                r[q] ;                          /* another low read */
            end_trans (T1) { getSignal [ → abort_trans(T1) ];
                                noSignalServiced ; } ;
            begin_trans (T2)
                r[o] ;
                w[p] ;
            end_trans (T2) ;
            begin_trans (T3)
                r[s] ;                          /* this is a low read */
                sl2 = SaveWork() ;
                r[y] ;                          /* another low read */
                w[q] ;
            end_trans (T3) { getSignal [ → abort_trans(T3) ];
                                noSignalServiced ;} ;
        coend ;
        using
                        thread end_trans (M) {
                            completedSet := union(completedSet,M);
                            if completedSet = {T1,T2,T3}
                                then call_back(T1,T3);
                        }
                        thread noSignalServiced (M) {
                            commitSet := union(commitSet,M) ;
                            if commitSet = {T1,T3}
                                then { commit (T1,T2,T3); exit ;}
                        }
                        thread abort_trans (M) {
                            abort (T1,T2,T3); exit;
                        }
        end;
}
```

Figure 11. A secure distributed commit protocol.

As each of the subtransactions T_1, T_2 and T_3 completes, it invokes the end_trans thread at the programmer's coordinator code. When the last transaction has invoked the end_trans thread, it chains the call_support primitive for transactions T_1 and T_3. Note that call_support is not invoked for T_2 as this subtransaction does not read down. The call_support primitive in turn causes the support codes defined in T_1 and T_3 and containing the default invocation of the getSignal primitive, to be executed. Each of these support codes contains the default invocation of the getSignal primitive having abort_trans(T_i) as the handler:

$$getSignal[\rightarrow abort\_trans(T_i)].$$

Consequently if any of the two subtransactions have a signal to be serviced, the abort_trans thread is invoked at the programmer's coordinator code. This results in all the three subtransactions to be aborted (which should be the case according to the definition of secure dependency). On the other hand if none of the subtransactions T_1 and T_3 is required to service signals (indicating that neither T_1 nor T_3 has released its low read locks prematurely), then the support codes cause the programmer-defined thread noSignalServiced to be invoked at the coordinator. At this point it is assured that the multiform transaction T is two phased and hence all the three subtransactions are committed.

In the following, we describe a multiform transaction that implements the advanced secure early prepare commit protocol (ASEP) presented in [19]. ASEP performs partial rollback of subtransactions that have read lower data instead of aborting them, when signals are to be serviced. Though the ASEP protocol is more complex than the secure protocol we have presented above, it can still be modeled by a multiform transaction and is one more evidence of the flexibility of our multiform transaction model.

6.3.1. Multiform transaction implementing ASEP. To implement the ASEP protocol we require the complete semantics of the getSignal primitive together with the notion of signals and signal labels. As stated in Section 6.3, signals are raised by the lock manager when it allows a write lock to be placed on a data item even though a conflicting lock is already in place over the same data item. The transaction manager for the higher level transaction associates each of these signals with the savepoint identifier (also called signal label) immediately preceding the affected low read operation in the transaction body. The getSignal primitive is then used by the programmer to specify how signals from lower level transactions are to be serviced by the higher level transaction.

The syntax of the getSignal primitive is as follows.

$$getSignal[sl_1 \rightarrow handler_1; \ldots; sl_n \rightarrow handler_n]$$

Each signal label sl_i represents a savepoint established by the transaction and corresponds to our notion of a savepoint identifier. The $handler_i$ represents a piece of program code that is to be executed when the signal sl_i is to be serviced. If the handler is a RollBack(sid) command then it rolls back the transaction to the savepoint sid.

By using this getSignal primitive it is possible to implement the ASEP protocol in a multiform transaction. This protocol iterates until it can guarantee that none of the low reading participants has released any of their low read locks within their window of uncertainty. The programmer can specify the maximum number of allowable iterations of the protocol, in order to prevent starvation. The ASEP protocol is implemented by the multiform transaction

```
void advanced_secure_early_prepare(){
        initiate (T1,T2,T3)
        coordinate
        cobegin
            begin_trans (T1)
                r[x]
                w[z]
                sl2 = SaveWork()
                r[y]                          /* this is a low read */
                sl3 = SaveWork()
                r[q]                          /* another low read */
            end_trans (T1) {getSignal [
                                sl2 → RollBack(sl2)
                                sl3 → RollBack(sl3)] ;
                                NoSignalServiced } ;

            begin_trans (T2)
                r[o]
                w[p]
            end_trans (T2) ;

            begin_trans (T3)
                r[s]                          /* this is a low read */
                r[x]
                w[z]
                sl2 = SaveWork()
                r[y]                          /* another low read */
                w[q]
            end_trans (T3){ getSignal[
                                    → RollBack ];
                                    NoSignalServiced }
        coend
        using
                    redone = 0; first = true ;
                        thread end_trans (M) {
                                if first then {
                                    completedSet = union(completedSet,M);
                                    If completedSet == {T1,T2,T3} then
                                        { first = false; call_support(T1,T3) }}
                                else { commitSet = null;
                                    redone = redone + 1;
                                        call_support(T1,T3) }
                            }
                        thread noSignalServiced (M) {
                                commitSet = union(commitSet,M) ;
                                If commitSet == {T1,T3} then
                                    { commit (T1,T2,T3); exit }
                            }
                        thread abort_trans (M) {
                                abort (T1,T2,T3); exit
                            }
                        exit when (redone == 5)
                        If redone == 5 then abort (T1,T2,T3);
        end;
}
```

Figure 12. Multiform transaction implementing ASEP.

shown in figure 12. The protocol proceeds as follows:

1. Subtransactions T_1, T_2, and T_3 are generated and distributed to the participating sites.
2. The transaction manager TM_i at each participant site executes the subtransaction T_i. If the subtransaction is able to complete successfully, TM_i forces a prepare log record, and executes an end_trans command. This causes the execution of the thread end_trans(T_i)

in the programmer defined code of the coordinator. The thread records the transaction identifier T_i in the persistent variable "completedSet" as this the first time T_i has caused this thread to be executed. If on the other hand the transaction is unable to complete successfully, the thread abort_trans(T_i) is executed which aborts all the three transactions T_1, T_2 and T_3 and terminates the program.

3. If each of the TM_i's is able to execute an end_trans command, the programmer defined code of the coordinator executes the call_support(T_1, T_3) command. Note that T_1 and T_3 are the transactions that have read lower level data and thus may have released the lower read locks, prematurely. From now onwards only T_1 and T_3 needs to be checked for reaching the commit decision.
4. As a result of the call_support primitive, control once again returns to TM_1 and TM_3. TM_2 on the other hand waits for a response from the coordinator. TM_1 and TM_3 executes the support_code associated with the end_trans primitive in T_1 and T_3 respectively.

   (A) Consider transaction T_1. If the low read lock for data item q was released prematurely, then the lock manager had issued a signal for this data item. TM_1 associated this signal with the savepoint sl_3 in T_1. Similarly if low read lock on data item y was released prematurely, the signal generated by the lock manager would be associated with savepoint sl_2. When the handler is executed, the getSignal primitive services any signal that is there. The semantics of the getSignal primitive is such that if low data item q is signaled, $sl_3 \rightarrow$ RollBack(sl_3) is executed, if either y or both q and y are signaled, $sl_2 \rightarrow$ RollBack(sl_2) is executed. If T_1 executes a RollBack, it will eventually terminate at which point TM_1 will again execute the end_trans primitive. This will cause as before, the execution of the thread end_trans(T_1) in the coordinator.

   (B) If none of the low data items is signaled, then NoSignalServiced is executed. In this case T_1 executes a NoSignalServiced primitive, and the thread with the same name is invoked at the coordinator.

   (C) The actions of T_3 corresponding to the call_support command is the same as those of T_1.

5. If both T_1 and T_3 have executed NoSignalServiced primitives, the persistent variable "commitSet" contains T_1 and T_3 and consequently the thread noSignalServiced in the coordinator executes the system primitive commit which commits the three transactions. The program then ends.
6. If any of T_1 or T_3 or both executes end_trans and neither executes an abort_trans, the thread end_trans(T_i) in the coordinator is executed. It increments the persistent variable "redone" and then the steps from (4) above are re-executed. This goes on until all the transactions are either committed or aborted or the variable redone is more than five. The use of the persistent variable "redone" prevents the indefinite delay in the commit decision.

6.4. Contingent transactions

A contingent transaction [9] is a set of two or more component transactions $T_1, T_2, \ldots, T_n$ with the property that at most one of the transactions, T_i, commits. A contingent transaction

```
void contingent()
{
        initiate(T1,T2,T3) ;
        coordinate
            begin_trans (T1)
                    .
                    .
                    .
            end_trans (T1);
            begin_trans (T2)
                    .
                    .
                    .
            end_trans (T2);
            begin_trans (T3)
                    .
                    .
                    .
            end_trans (T3);
        using
                                    thread end_trans (M) {
                                        if M == T1 then
                                            { commit (T1); abort (T2,T3); exit; }
                                        if M == T2 then
                                            { commit (T2); abort (T1,T3); exit; }
                                        if M == T3 then
                                            { commit (T3); abort (T1,T2); exit; }
                                    }
                                    thread abort_trans (M) {
                                        abortSet = union(M,abortSet) ;
                                        if subseteq({T1,T2,T3},abortSet) then
                                        { abort (T1,T2,T3) exit; }
                                    }
        end ;
}
```

Figure 13. Example of a contingent transaction.

$T = \{T_1, T_2, \ldots, T_n\}$ is executed as follows: T_1 is executed first. If it commits, then the transaction T commits and ends. If T_1 aborts, T_2 is executed and if commits, the entire contingent transaction T commits and ends and so on.

The program fragment in figure 13 shows how a contingent transaction can be implemented within our framework. In the example the contingent transaction consists of three component transactions T_1, T_2 and T_3.

6.5. Nested transactions

A nested transaction is a transaction that is executed from inside the dynamic scope of another transaction. Nested transactions can further create nested transactions and the nesting can proceed to arbitrary depths. The transaction at the root of this tree of transactions is called the root transaction and the transactions at the interior nodes (called parents) or leaves of this tree are jointly called subtransactions. Subtransactions execute atomically with respect to their siblings.

Each of the parent transactions is suspended until all its nested transactions terminates (i.e. commits or aborts). However, the semantics of commit for the nested transactions are different from that for the root transaction. When a nested transaction (parent or leaf) commits, the changes that it made to the database are made accessible to its parent, but are not made permanent. Instead the changes are made permanent only when the root

```
void nested-transactions ()
{
        initiate(T1,T2,T3,T4);
        coordinate
        begin_trans(T1)
          .
          .
          .
            begin_trans(T2)
                flightReservation(United, 6/3/96)
            end_trans(T2) ;
          .
          .
          .
            begin_trans(T3)
                hotelReservation(Ambassador, 6/3/96, 6/6/96)
            end_trans(T3) ;
          .
          .
          .
            begin_trans(T4)
                carReservation(Avis, 6/3/96, 6/6/96)
            end_trans(T4) ;
          .
          .
          .
        end_trans(T1) ;
        using
                        thread end_trans (M) do {
                            if M == T1 then commit (T1,T2,T3,T4); exit ;
                        };
                        thread abort_trans (M) do {
                                abort (T1,T2,T3,T4); exit ;
                        };
        end;
}
```

Figure 14. Nested transactions.

transaction commits. Abort semantics for both root and subtransactions are similar to the abort semantics for the classical transaction. Furthermore, a subtransaction can access any data item that is currently accessed by one of its ancestors without forming a conflict.

We illustrate the implementation of a nested transaction in our model by an example. The example shown in figure 14 involves a nested transaction with two nesting levels, which makes travel arrangements. If at any stage a reservation cannot be made, the trip is canceled. At any stage thus, if the trip is to be canceled, any previous reservation has to be canceled. Note that unlike in the workflow model where previous reservations are canceled by explicitly executing compensating transactions, in the nested transaction we do not require any compensating transaction. This is because of the fact that the effects of subtransactions are made permanent only at the commit of the root transaction.

6.6. The ACTA framework

ACTA identifies two broad classes of dependencies—viz. direct dependency among a pair of transactions and indirect dependency between a pair of transactions arising from their actions on common data items. These two classes can be studied separately. As we are interested only in allowing flexible direct dependency to transactions, we examine just this class here. For this discusssion we assume that the only indirect dependency within the

multiform transaction model is the conflict dependency arising from component transactions manipulating shared data items.

The ACTA framework defines two major types of direct dependencies among pairs of transactions [7]. These are:

1. Commit dependency—If a transaction T_1 develops a commit dependency on transaction T_2, then T_1 cannot commit until T_2 either commits or aborts. This does not imply that if T_2 aborts then T_1 should abort.
2. Abort dependency—If a transaction T_1 develops an abort dependency on another transaction T_2 and T_2 aborts, then T_1 should also abort. This neither implies that if transaction T_2 commits, T_1 should commit, nor that if T_1 aborts, T_2 should abort.

Note that the abort dependency implies the commit dependency. If T_1 develops an abort dependency on T_2 then T_1 must wait for the commit decision of T_2; hence T_1 cannot commit before T_2 i.e. there is a commit dependency between T_1 and T_2. In figure 15 we provide a multiform transaction that implements a commit dependency and in figure 16 we give an example of a multiform transaction which implements an abort dependency.

Refer to figure 15. Suppose that T_1 wants to commit. It executes an end_trans primitive which causes the end_trans thread at the coordinator to be executed. Since T_2 has not yet executed the end_trans (or abort_trans) primitive the variable doneT_2 is false. Consequently, the end_trans thread sets the variable completedT1 to true and returns. As no commit or abort decision has been taken for T_2 by the coordinator, T_1 cannot terminate at this time by committing. On the other hand if T_1 had decided to abort, it would have executed the

```
void commit_dependency ()
{       initiate (T1,T2) ;
        coordinate ;
        cobegin
            begin_trans (T1)
            ...
            end_trans (T1) ;
            begin_trans (T2)
            ...
            end_trans (T2) ;
        coend ;
        using
                    completedT1 := false ; doneT2 := false ;
                        thread end_trans (M) {
                            if doneT2 then {commit(T1); exit ;}
                            else if M == T2 then {
                                    commit(T2); doneT2 = true ;
                                    if completedT1 then { commit(T1); exit ;}}
                                else completedT1 = true ; }
                        thread abort_trans (M) {
                            if M == T2 then {abort(T2) ; doneT2=true;
                                if completedT1 then { commit(T1); exit ;}}
                            else abort(T1) ;
                        }
        end ;
        }
```

Figure 15. ACTA commit dependency.

```
void abort_dependency ()
{       initiate (T1,T2) ;
        coordinate ;
        cobegin
            begin_trans (T1)
            ...
            end_trans (T1) ;
            begin_trans (T2)
            ...
            end_trans (T2) ;
        coend ;
        using
                    completedT1 := false ; doneT2 := false ;
                        thread end_trans (M) {
                            if doneT2 then {commit(T1); exit ;}
                            else if M == T2 then {commit(T2); doneT2 = true ;
                                    if completedT1 then { commit(T1); exit ;}}
                                else completedT1 = true ;
                        }
                        thread abort_trans (M) {
                            if M == T2 then { abort(T2,T1); exit; }
                            else abort(T1) ;
                        }
        end ;
        }
```

Figure 16. ACTA abort dependency.

abort_trans primitive, which in turn would have caused the abort_trans thread to be executed at the coordinator. This would abort T_1 irrespective of whether T_2 commits or aborts.

When T_2 decides to commit or abort, the variable doneT_2 will be set to true by one of the threads end_trans or abort_trans. If T_2 executes an abort_trans primitive, the corresponding thread aborts T_2. The abort_trans thread then finds that the variable completedT_1 is set to true (which indicates that T_1 is waiting to commit) and hence commits T_1. If, on the other hand, T_2 executes the end_trans primitive (indicating that it wants to commit), the end_trans thread commits T_2 first and then, noticing that completed T_1 is set to true, commits T_1. At this point the program terminates.

From the above discussion it is clear that the program in figure 15 implements the ACTA commit dependency between T_1 and T_2. The next figure (figure 16) implements an abort_dependency between T_1 and T_2. The discussion for this program fragment is similar to the above with the only difference being that if T_2 executes an abort_trans primitive, the corresponding abort_trans thread in the coordinator aborts both T_2 and T_1, even if T_1 has previously decided to commit. Moreover, if T_1 is yet to reach a decision when T_2 has decided to abort, T_1 is aborted.

7. A high level declarative language for specifying transaction dependencies

The following definitions establish the declarative language used to specify completion dependencies within a multiform transaction. Commit and abort dependencies can be specified and we permit a transaction to depend on a set of transactions for commit or abort.

Definition 7. Let T_i, T_j be two transactions. The commit and abort dependencies among T_i and T_j defined in Section 6.6 are expressed in our declarative language as follows:

Commit dependency If transaction T_i develops a commit dependency on transaction T_j then this is denoted by $T_i \rightarrow T_j$.

Abort dependency If transaction T_i develops an abort dependency on transaction T_j then this is denoted by $T_i \rightsquigarrow T_j$.

Recall from Section 6.6 that the enforcement of these dependencies requires that the transaction on the left-hand side of the dependency (that is, T_i in the above definition) wait for the termination of the transaction of the right-hand side of the dependency (that is, T_j in the above definition). Indeed, the outcome of T_j (abort vs. commit) may determine the outcome of T_i. For example, the abort dependency $T_i \rightsquigarrow T_j$ specifies that the abort of T_j is a sufficient condition for the abort of T_i. Therefore T_i must still wait for the outcome of T_j upon completing its normal execution.

We consider a more general form of completion dependencies and assume that dependencies can exist not only between two transactions but also between a transaction and a set of transactions. In addition, completion dependencies can be combined using the AND, OR and NOT logical operators. Therefore, an expression obtained as a Boolean combination of transactions may appear on the right-hand side of a dependency. We make the assumption that logical expressions are given in some minimal form.

Definition 8. Let $\mathcal{T}$ be a set of transactions and let T be a transaction in $\mathcal{T}$. Let also $\mathcal{TD}$ be a subset of $\mathcal{T}$ not including T. A *General Dependency* between T and transaction set $\mathcal{TD}$ is defined as a commit or abort dependency between T and elements of $\mathcal{TD}$ obeying the rules given is Table 3.

Here ⟨gendep⟩, ⟨cdep⟩ and ⟨adep⟩ indicates general, commit and abort dependencies respectively. We assume that the AND composition specified by rule R5 takes precedence over the OR composition specified by rule R6.2. The reverse precedence can be enforced

Table 3. Syntax for declarative language for general dependencies.

R1	⟨gendep⟩	::=	⟨cdep⟩ ⟨adep⟩
R2.1	⟨cdep⟩	::=	empty
R2.2	⟨cdep⟩	::=	$T \rightarrow$ ⟨term⟩
R2.3	⟨cdep⟩	::=	$T \rightarrow$ NOT⟨term⟩
R3.1	⟨adep⟩	::=	empty
R3.2	⟨adep⟩	::=	$T \rightsquigarrow$ ⟨term⟩
R3.3	⟨adep⟩	::=	$T \rightsquigarrow$ NOT⟨term⟩
R4.1	⟨term⟩	::=	$T' : T' \in \mathcal{TD}$
R4.2	⟨term⟩	::=	⟨andterm⟩
R4.3	⟨term⟩	::=	(⟨orterm⟩)
R5	⟨andterm⟩	::=	$T' : T' \in \mathcal{TD}$ AND ⟨term⟩
R6.1	⟨orterm⟩	::=	$T' : T' \in \mathcal{TD}$ OR ⟨orterm⟩
R6.2	⟨orterm⟩	::=	$T' : T' \in \mathcal{TD}$ OR ⟨term⟩ □

by the use of parenthesized OR expression, as specified by rule R4.3. There are two basic predicates:

$$\textit{Terminate}(T) = \text{true if } T \text{ commits or aborts, false otherwise;}$$
$$\textit{Abort}(T) = \text{true if } T \text{ aborts, false otherwise.}$$

The semantics of the notation in Table 3 is defined in terms of the basic predicates: A generalized commit dependency specifies that transaction T cannot commit until the right hand side Boolean expression evaluates to true when each transaction $T' \in \mathcal{TD}$ is substituted by its corresponding *Terminate*(T') predicate. Thus a generalized abort dependency forces transaction T to wait until the Boolean expression, evaluates to true when each transaction T' is substituted by its corresponding *Terminate*(T') predicate. At that point, transaction T is forced to abort if the Boolean expression specified by the right-hand side of the dependency evaluates to true with each transaction T' substituted by its corresponding *Abort*(T') predicate.

Pairwise commit or abort dependencies are obtained as a special case of our general notion of dependencies.

Figure 17 shows the specification, in this high level declarative language, of the example dependency given earlier in figure 2. Recall that in figure 2 the dependency was either

```
void example()
{
        coordinate
        initiate(T1,T2) ;
            begin_trans (T1)
                    .
                    .
                    .
            end_trans (T1);
            begin_trans (T2)
                    .
                    .
                    .
            end_trans (T2);
        using
            T2 ⇝ NOT(Abort(T1))
        end;
        coordinate
        initiate(T3,T4) ;
        cobegin
            begin_trans (T3)
                    .
                    .
                    .
            end_trans (T3);
            begin_trans (T4)
                    .
                    .
                    .
            end_trans (T4);
        coend
        using dependency
            T3 → (T3 → T4) AND (T4 → T3)
            T4 → (T3 → T4) AND (T4 → T3)
        end;
}
```

Figure 17. High level description of dependency from figure 2.

transaction T_1 commits or T_2 commits, but not both, with T_1 getting precedence over T_2. Both T_1 and T_2 may abort. For transactions T_3 and T_4 the dependency was that either both commit or none do so. Further T_3 and T_4 can commence execution only after T_1 and T_2 terminates.

As in figure 2, the sequential structuring of the coordinate blocks for T_1 and T_2 and for T_3 and T_4 ensures that T_3 and T_4 gets executed only after T_1 and T_2 terminates. The cycle in the dependency expressions for T_3 and T_4 (viz. $(T_3 \rightarrow T_4)$ AND $(T_4 \rightarrow T_3)$) ensures that either both commit or none do so.

For transactions T_1 and T_2, their sequential definitions ensure that T_1 gets precedence over T_2 (just as in figure 2). The termination of T_1 is not dependent on anything but that of T_2 is. Hence we have a dependency expression for T_2. The term $(T_2 \rightsquigarrow \text{NOT}(\text{Abort}(T_1))$ ensures that if T_1 commits (in which case $\text{NOT}(\text{Abort}(T_1))$ evaluates to true), then T_2 aborts.

8. Related work

The work in the area of extended transaction models can be broadly classified into two categories. The first category (which incidentally happens to be the earlier models) requires rigid organizational structures of related transactions. Examples of such categories are nested transactions (transaction organized in a tree-like structure) and SAGAS (a linear structure). The second category allows more flexible transaction structuring such as the workflow model. A number of extended transaction models of the second category provide system primitives that allow the user to specify different order of commit for cooperating transactions. Examples of such models are ASSET [5], Flex [6], DOMS [12]. In the following we discuss some of the extended transaction models described in the literature. For a comprehensive discussion on extended transaction models, the reader is referred to [9].

The nested transaction model [17] organizes transactions in a tree-like hierarchy to localize failures within a transaction and to exploit parallelism within transactions. Argus [16] supports this model and is one of the earliest efforts at providing linguistic support for extended transactions. Camelot [10] also implements the nested transaction model. It allows the programmer to choose from a set of three predefined commit protocols—two-phase commit, non-blocking commit and lazy commit. However, the programmer does not have the option of extending these protocols to provide more flexibility in transaction completion.

The SAGAS model [11] offers a linear structure for organizing a set of transactional activities. It is useful only when the subtransactions in a Saga are relatively independent of each other and each subtransaction can be successfully compensated. Moreover due to the strict linear structure, a Saga is not as flexible as the multiform transaction model.

The Flex transaction model [6] is based on the nested transaction model. It allows relaxation of the atomicity and isolation requirements of transactions. A Flex transaction is resilient to failure in the sense that it may proceed and commit even if some of its subtransactions fail. The model allows the user to control the isolation granularity of a transaction through the use of compensating subtransactions. The model offers flexibility in transaction processing by providing primitives that allow the specification of dependencies between subtransactions of a Flex transaction. The specifiable dependencies can be broadly categorized into two types: those that define the execution order on the subtransactions of a

Flex transaction—called *failure-dependencies* and *success-dependencies*—and those that define the dependencies of subtransactions on events that do not belong to the transaction—called *external-dependencies*. The ability to express external dependencies makes the Flex transaction model more powerful than the previous models; however the unique feature of the Multiform transaction model, namely the ability of programmers to redefine existing transaction primitives, makes this model more flexible than the Flex model.

The DOMS transaction model [12] was developed for the Distributed Object Management project at the GTE Laboratories. The goal of this project was to support application development in a distributed environment that integrates various component systems. DOMS allows both classical nested transactions [17] and open nested transactions—an extension of the classical nested transactions that do not enforce the atomicity of the root level transaction—as well as a combination of the two. DOMS provides a specification language similar to ACTA, to express the properties of extended transactions and dependencies among them. It provides a mechanism that configure the run-time transaction facility to realize the extended transaction. The programmer can construct *certain* extended transactions using expressions in the specification language, which are then mapped to certain *pre-built* configurations in the transaction management mechanism. The primitives of DOMS are limited to the specification level, while the primitives in the multiform transaction model are at the programming language level. Moreover the specification language suffers from similar shortcomings as ACTA.

ConTract [20] is a programming model that provides a basis for defining and controlling long-lived complex computations, composed of transaction-like components. In ConTract, the coding of the steps of the components is kept separated from managing asynchronous or parallel components, communications and synchronization between these components and failure recovery. These latter aspects are implemented by a separate flow control script which allows different dependencies among transactions to be specified. ConTract provides a database environment for specifying and executing workflows whereas the multiform transaction model embeds the specification of flexible transaction dependencies at the programming language level. ConTract scripts introduce their own control flow syntax while we use the control flow syntax of the host language.

ASSET provides language primitives for specifying the cooperation and dependency between a set of related transactions. This is achieved by intermingling within the same program, both the coding of the transactions and the transaction's control flow. Unfortunately, this approach appears to reduce the transaction code re-usability since the application program has to deal with both the transaction code and the transaction cooperation and dependencies. Furthermore, Asset does not allow proper coordination among concurrently running but cooperating transactions. The synchronization primitive proposed is too weak and difficult to use to provide flexible coordination among concurrent transactions.

The Reflective Transaction Framework of [2] is different from all these models and is the closest to the multiform transaction model. Indeed our model incorporates some of the ideas of [2]. The reflective transaction framework, is a practical modular method to implement extended transaction models and is able to support more than one model in the same framework. The framework builds on top of an existing commercial transaction processing monitor—the Encina toolkit [21]. The transaction processing monitor provides the general

framework that binds together the many software components of a transaction processing system through services like multithreaded processes, interprocess communication, queue management etc. The authors propose the notion of transaction adapters that are add on modules over the TP monitor to extend its basic services. The framework provides both—the flexibility of language primitives, enabling a developer to construct extended transactions from scratch, and the high level interface and functionality of a specialized transaction facility, enabling an application programmer to construct extended transactions from existing component. The multiform transaction model is an improvement of the reflective transaction framework in that it allows application specific completion dependencies that do not fit into any general model and also allows dependencies arising out of any type of events—not necessarily internal to the transaction. Also the multiform transaction is suitable for distributed transaction processing unlike the reflective framework which essentially is a centralized system.

Summarizing, the Multiform transaction model incorporates all the powerful features of the majority of the exisiting extended transaction models. We can have a single system that implements the multiform model and use it for many different applications, each with its own set of requirements. This feature we cannot have from a system that implements, for example, only the SAGAS model or the Nested transaction model or the DOMS model.

9. Conclusions and future work

This paper presents the *multiform transaction* model that allows the programmer to specify various transaction completion dependencies including, in particular, those proposed in the most well-known extended transaction models.

A multiform transaction consists of a set of cooperating transactions together with a sequence of coordinators implementing the completion dependencies. The programmer is provided with a small set of transaction primitives by which he/she can develop application specific coordinators. Additionally, the programmer can redefine some of these primitives for additional flexibility by providing code for the implementation of the new definitions. Moreover, the compiler of a database programming language can also use these primitives to support higher level constructs for transactions. In this case, the compiler can automatically generate the appropriate codes needed for coordination of a set of transactions from a high level description of their dependencies.

Not only can the programmer re-define some of the existing primitives, he/she can also define additional primitives to satisfy his/her own requirement. These new primitives can be defined as new threads of a coordinator. An example of such a new application specific primitive has been shown in Section 6.3, where it has been used to support the secure dependencies among transactions. This feature seems useful for supporting some other extended transaction models not discussed in this work, like the split-join transaction model. For example in the case of split-join transactions, the programmer can define two new threads *split and delegate* in the protocol component. The split thread starts a new transaction and the delegate thread delegates a set of data from one transaction to another. When a transaction T_i is to be split, T_i first invokes delegate to transfer a subset of its data items to another initiated transaction T_j and then invokes the split thread. The latter thread starts the execution of T_j.

When T_j needs to be joined with T_i, T_j invokes delegate to transfer back the data items to T_i and then terminates.

As we mentioned in the introduction, we have limited ourselves to exploring the category of transaction completion dependencies. This has been done to simplify the presentation of the basic ideas behind the multiform transaction model. Nevertheless data dependency can be incorporated in our model by defining a new set of primitives that allow flexible concurrency control. The getSignal primitive introduced in Section 6.3 appears to be a good candidate.

It appears that the multiform transaction model is a practical way to implement extended transaction models in a distributed setup following the approaches chosen by [2] for a centralized system. In [2] the authors extend Transarc's Encina TP system [13, 21] by developing transaction management adapters on top of Encina. Our transaction management adapters offer the same functionality as that in [2], while the notion of coordinator module can be viewed as an extension of *meta-transactions* of [2] to include executing codes and communication mechanisms. Thus it seems that the multiform transaction model is an elegant way to support distributed extended transaction models on conventional transaction processing systems.

We plan to incorporate these ideas within the framework of an ongoing project on multilevel secure transaction processing system. As part of future work we plan to complement this current set of primitives with another set for flexible concurrency control so as to have a complete flexible transaction processing system that supports both classes of dependencies—viz. completion dependency and data dependency between a pair of transactions arising from their actions on common data items.

Acknowledgments

The work of S. Jajodia was partially supported by National Science Foundation under grants IRI-9303416, IRI-9633541, and INT-9412507 and by National Security Agency under grants MDA904-96-1-0103 and MDA904-96-1-0104. The work of Luigi V. Mancini was partially supported by the Italian M.U.R.S.T. and the Italian C.N.R.

Notes

1. Note that a process can be made to react to an event in many different ways: The event can generate an interrupt to the process; the event can send a message to a port at which the process listens or the event can invoke a RPC at the process. We choose not to specify the exact mechanism so as to keep the model as much implementation independent as possible.
2. In most commit protocols, if any subtransaction aborts, the coordinator always sends an abort decision to all participants. However, in our protocol the coordinator may not send an abort decision. Instead the coordinator can ask the uncommitted transaction to restart its execution. This can be useful in many situations. For example, suppose the subtransaction could not complete because of a site crash. Then when the site comes up, the subtransaction can be restarted instead of being aborted.
3. A transaction T_i is an orphan if it is never explicitly terminated by any coordinator module within the multiform transaction. When a transaction T_i is orphan the locks acquired by T_i are not released and the updates made by T_i are not made permanent. This may cause a number of problems like deadlock or unsatisfiable dependencies. A complete discussion is outside the scope of this paper.
4. We assume here that the programmer does not save the contents of the savepoint identifier before re-using it.

References

1. V. Atluri, E. Bertino, and S. Jajodia, "Degrees of isolation, concurrency control protocols and commit protocols," in Database Security, VIII: Status and Prospects, J. Biskup, M. Morgenstern and C. Landwehr (Eds.), North-Holland: Amsterdam, 1994, pp. 259–274.
2. R. Barga and C. Pu, "A practical and modular method to implement extended transaction models," in Proceedings of the 21st International Conference on Very Large Data Bases, Zürich, Switzerland, 1995, pp. 206–217.
3. E. Bertino, G. Chiola, and L.V. Mancini, "Deadlock detection in the face of transaction and data dependencies in advanced transaction models," in Proceedings of the 19th International Conference on Application and Theory of Petri Nets, Lisbon, Portugal, June 1998, pp. 266–285.
4. E. Bertino, S. Jajodia, L.V. Mancini, and I. Ray, "Advanced transaction processing in multilevel secure file stores," IEEE Transactions on Knowledge and Data Engineering, vol. 10, no. 1, 1998, pp. 120–135.
5. A. Biliris, S. Dar, N. Gehani, H.V. Jagadish, and K. Ramamritham, "ASSET: A system for supporting extended transactions," in Proceedings of the ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 1994, pp. 44–54.
6. O. Bukhres, A. Elmagarmid, and E. Kuhn, "Implementation of the flex transaction model," Bulletin of the IEEE Technical Committee on Data Engineering, vol. 12, no. 2, pp. 28–32, 1993.
7. P.K. Chrysanthis and K. Ramamritham, "Synthesis of extended transaction models using ACTA," ACM Transactions on Database Systems, vol. 19, no. 3, pp. 450–491, 1994.
8. U. Dayal, M. Hsu, and R. Ladin, "Organizing long-running activities with triggers and transactions," in Proceedings of the ACM SIGMOD International Conference on Management of Data, Atlantic City, May 1990, pp. 204–214.
9. A.K. Elmagarmid, (Ed.), Database Transaction Models for Advanced Applications, Morgan Kaufmann Publishers, Inc.: San Mateo, CA, 1991.
10. J.L. Eppinger, L.B. Mummert, and A.Z. Spector (Eds.), Camelot and Avalon. A Distributed Transaction Facility, Morgan Kaufmann Publishers, Inc.: San Mateo, CA, 1991.
11. H. Garcia-Molina and K. Salem, "SAGAS," in Proceedings of the ACM SIGMOD International Conference on Management of Data, May 1987, pp. 249–259.
12. D. Georgakopoulos, M. Hornick, P. Krychniak, and F. Manola, "Specification and management of extended transactions in a programmable transaction environment," in Proceedings of the IEEE 10th International Conference on Data Engineering, 1994, pp. 462–477.
13. J. Gray and A. Reuter, Transaction Processing: Concept and Techniques, Morgan Kaufmann Publishers: San Mateo, CA, 1993.
14. S. Jajodia, C.D. McCollum, and B.T. Blaustein, "Integrating concurrency control and commit algorithms in distributed multilevel secure databases," in Database Security, VII: Status and Prospects, T.F. Keefe and C.E. Landwehr (Eds.), North-Holland: Amsterdam, 1994, pp. 109–121.
15. E. Knapp, "Deadlock detection in distributed databases," ACM Computing Surveys, vol. 19, no. 4, pp. 303–328, 1987.
16. B. Liskov and R. Scheifler, "Guardians and actions," ACM Transactions on Programming Languages and Systems, vol. 8. no. 4, pp. 484–502, 1983.
17. J. Eliot B. Moss, "Nested transactions: An approach to reliable distributed computing," PhD Thesis, EECS Department, M.I.T., 1981.
18. S.S. Muchnick and N.D. Jones (Eds.), Program Flow Analysis: Theory and Applications, Prentice Hall: Englewood Cliffs, N.J., 1981.
19. I. Ray, S. Jajodia, E. Bertino, and L. Mancini, "An advanced commit protocol for MLS distributed database systems," in Proceedings of the 3rd ACM Conference on Computer And Communications Security, New Delhi, India, March 1996, pp. 119–128.
20. A. Reuter, "ConTracts: A means for extending control beyond transaction boundaries," in Proceedings of the 3rd International Workshop on High Performance Transaction Systems, Asilomar, September 1989.
21. Transarc Corporation, Encina Toolkit Server Core Programmer's Reference, Pittsburgh, PA 15219, 1993.
22. A. Zhang, M. Nodine, B. Bhargava, and O. Bukhres, "Ensuring relaxed atomicity for flexible transactions in multidatabase systems," in Proceedings of the ACM SIGMOD International Conference on Management of Data, May 1994, pp. 67–78.