Detection for Conflicts of Dependencies in Advanced Transaction Models

Tai Xin Indrakshi Ray

Colorado State University
Fort Collins, CO 80523-1873
E-mail: `xin,iray@cs.colostate.edu`

Abstract

Transactional dependencies play an important role in coordinating the execution of subtransactions in advanced transaction models, such as, nested transactions and workflow transactions. The correct execution of the advanced transactions depends on ensuring the satisfaction of all the dependencies, which are specified by the application developer. Incorrect specification of transaction dependencies might lead to information integrity problems and unavailability of resources. An example of incorrect specification of dependencies is the presence of conflicts – the satisfaction of constraints imposed by one dependency may violate the constraints imposed by another dependency. Algorithms that can analyze and detect dependency conflicts are necessary. Although a lot of research appears on advanced transactions, no previous work has been done on analysis of dependency conflicts. In this work we analyze different kinds of dependency conflicts, propose algorithms to detect and remove the conflicts of dependencies in advanced transaction specifications. This will enable the application developer to get assurance about the correctness of the dependency specification and the correct behavior of the underlying advanced transaction model.

1 Introduction

Although the traditional transaction processing model has proved very successful for typical applications, it is inadequate for processing specialized transactions having long-duration and cooperative activities. To address this shortcoming, researchers have proposed various advanced transaction processing models [1, 6, 7, 9, 11, 12, 13, 17]. These advanced transaction models, differing in forms and applicable environments, have two common properties: they are made up of long running activities, and, they normally contain highly cooperative activities. We will refer to these component activities in an advanced transaction as *subtransactions* in this paper. The subtransactions are not independent entities; they need to be coordinated to accomplish a specific task. Such co-ordination is achieved through *dependencies*.

Application developers are responsible for specifying the dependencies in an advanced transaction. The correct execution of the advanced transactions depends on ensuring the satisfaction of all the specified dependencies. However, dependencies in an advanced transaction can be specified incorrectly; the constraints imposed by these dependencies may conflict with each other and they can never be satisfied simultaneously in execution. Incorrect specification of dependencies might lead to undesirable outcomes, such as unavailability of services and information integrity problems. For example, consider the two dependencies $T_i \rightarrow_{bc} T_j$ and $T_i \rightarrow_{fba} T_j$ specified over a pair of transactions T_i and T_j. The *begin on commit* dependency $T_i \rightarrow_{bc} T_j$ imposes the restriction that T_j cannot begin until T_i commits. The *force begin on abort* dependency $T_i \rightarrow_{fba} T_j$ requires that T_j must begin when T_i aborts. Since their constraints cannot be satisfied together, these dependencies are conflicting.

Existing research work, like ACTA [6, 7] and ASSET [5], have described and formalized dependencies in advanced transactions. However, not much work appears in dependency analysis. Our goal is to fill this gap. We begin by describing the kinds of dependencies that can exist in advanced transaction models. We discuss how these dependencies can be classified and combined. We illustrate the kinds of conflicts that can occur in these dependencies. Finally, we propose algorithms to automatically detect the conflicts resulting from the interaction of these dependencies.

The rest of the paper is organized as follows. Section 2 presents some related work in this area. Section 3 describes our transaction processing model and different kinds of dependencies. Section 4 lists the characteristics of dependencies. Section 5 illustrates different kinds of dependency conflicts. Section 6 formalizes the algorithms needed for conflict detection. Section 7 concludes the paper.

2 Related Work

Chrysanthis and Ramamrithan in [8] introduce ACTA, a formal framework for specifying extended transaction models. ACTA allows intuitive and precise specification of extended transaction models by characterizing the semantics of interactions between subtransactions in terms of different dependencies between subtransactions, and in terms of subtransaction's effects on data objects. During the past few years, ACTA has been used for specifying and reasoning about advanced transaction models.

Biliris et al. described "ASSET" as A System for Supporting Extended Transactions in [5]. They provided a set of subtransaction primitives extending a programming language. Beyond the traditional transaction primitives (e.g., begin, commit, abort, get_parent) it introduces new primitives allowing creation dependencies between subtransactions, resource delegation, and giving permissions for an access to acquired resources. None of the provided examples uses dependencies; instead, dependencies are implicitly modeled by flow control constructs.

Mancini, Ray, Jajodia and Bertino have proposed the notion of multiform transactions [10]. A multiform transaction consists of a set of transactions and a set of termination dependencies specified among these transactions. The set of dependencies specifies the commit, abort relationship among the component transactions. The multiform transaction is organized as a set of coordinate blocks. The coordinate block, along with the corresponding coordinator module (CM) can manage the execution of the transactions.

In Bertino, Chiola, and Mancini's recent paper [4], the authors discussed the deadlock detection issues in advanced transaction models, in face of transactional and data dependencies. In the paper, the authors show that in the face of these dependencies, deadlocks may arise that the conventional deadlock detection algorithms are not able to detect. They discussed transaction waiting states characterized by AND-OR graphs, and proposed an algorithm for detecting deadlocks in these graphs. The authors claimed that their algorithm has a computational complexity linear in the number of nodes and edges of the AND-OR graphs.

Singh has discussed the semantical inter-task dependencies on workflows [16]. The author uses algebra to express the dependencies and analyze their properties in workflow systems. Attie at el. [3] discussed means to specify and enforce intertask dependencies. They illustrate each task as a set of significant events (start, commit, rollback, abort). Intertask dependencies connect these events of various tasks and limit the occurrence and temporal order of them. None of these papers focus on analyzing conflicts in dependencies.

Kerstin, Turker and Saake have discussed some features of the execution dependency in transaction closure [14, 15]. They also did some formalization analysis for the dependencies. The authors discussed the execution dependency. The execution dependencies between transactions can be expressed in terms of *begin* and *end* events associated with the corresponding transactions, and they restrict the temporal occurrence of the events. The authors also discuss transitive properties of execution dependencies. However, they do not address the problem of dependency conflicts.

3 Generalized Advanced Transaction Model

In this work, we do not propose a new transaction model. Instead, we propose the concept of a generalized advanced transaction. The generalized advanced transaction can be customized for different kinds of advanced transaction models and workflow systems as well, by restricting the type of dependencies that can exist between the component subtransactions. We begin by the definition of subtransactions and dependencies.

Definition 1 [Subtransaction] Subtransactions are low-level coordinated activities in an advanced transaction. A subtransaction T_i contains a set of related operations of a subtransaction, and forms a smallest unit in an advanced transaction. The begin, abort and commit operations of subtransaction T_i are denoted by b_i, a_i and c_i respectively.

Definition 2 [Dependency] In advanced transaction model, a dependency specifies how the execution of primitives (begin, commit, and abort) of a subtransaction T_i causes (or relates to) the execution of the primitives (begin, commit and abort) of another subtransaction T_j.

Next, we define a generalized advanced transaction.

Definition 3 [Generalized advanced transaction] A *Generalized Advanced Transaction* $T = < S, D >$ is defined by S, which is the set of subtransactions in T, and D, which is the set of dependencies between the subtransactions of T.

In the rest of the paper we use the term *advanced transaction* instead of generalized advanced transaction.

A set of dependencies has been defined in the work of ACTA [7]. A comprehensive list of transaction dependency definitions can be found in [2, 5, 7]. Summarizing all these dependencies in previous work, we collect a total of fifteen different types of dependencies.

[Commit dependency] ($T_i \rightarrow_c T_j$): If both T_i and T_j commit then the commitment of T_i precedes the commitment of T_j. Formally, $c_i \Rightarrow (c_j \Rightarrow (c_i \prec c_j))$.

[Strong commit dependency] ($T_i \rightarrow_{sc} T_j$): If T_i commits then T_j also commits. Formally, $c_i \Rightarrow c_j$.

[Abort dependency] ($T_i \rightarrow_a T_j$): If T_i aborts then T_j aborts. Formally, $a_i \Rightarrow a_j$.

[Weak abort dependency] ($T_i \rightarrow_{wa} T_j$): If T_i aborts and T_j has not been committed then T_j aborts. Formally, $a_i \Rightarrow (\neg (c_j \prec a_i) \Rightarrow a_j)$

[Termination dependency] ($T_i \rightarrow_t T_j$): Subtransaction T_j cannot commit or abort until T_i either commits or aborts. Formally, $e_j \Rightarrow e_i \prec e_j$, where $e_i \in \{c_i, a_i\}$, $e_j \in \{c_j, a_j\}$.

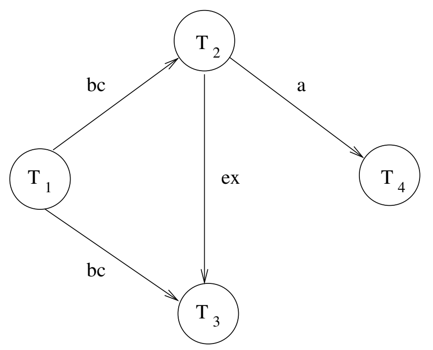

Figure 1. Example Advanced Transaction

[Exclusion dependency] ($T_i \rightarrow_{ex} T_j$): If T_i commits and T_j has begun executing, then T_j aborts. Formally, $c_i \Rightarrow (b_j \Rightarrow a_j)$.

[Force-commit-on-abort dependency] ($T_i \rightarrow_{fca} T_j$): If T_i aborts, T_j commits. Formally, $a_i \Rightarrow c_j$.

[Force-begin-on-commit/abort/begin/termination dependency] ($T_i \rightarrow_{fbc/fba/fbb/fbt} T_j$): Subtransaction T_j must begin if T_i commits(aborts/begins/terminates). Formally, $c_i(a_i/b_i/T_i) \Rightarrow b_j$.

[Begin dependency] ($T_i \rightarrow_b T_j$): Subtransaction T_j cannot begin execution until T_i has begun. Formally, $b_j \Rightarrow (b_i \prec b_j)$.

[Serial dependency] ($T_i \rightarrow_s T_j$): Subtransaction T_j cannot begin execution until T_i either commits or aborts. Formally, $b_j \Rightarrow (e_i \prec b_j)$ where $e_i \in \{c_i, a_i\}$.

[Begin-on-commit dependency] ($T_i \rightarrow_{bc} T_j$): Subtransaction T_j cannot begin until T_i commits. Formally, $b_j \Rightarrow (c_i \prec b_j)$.

[Begin-on-abort dependency] ($T_i \rightarrow_{ba} T_j$): Subtransaction T_j cannot begin until T_i aborts. Formally, $b_j \Rightarrow (a_i \prec b_j)$.

Let's see an example of an advanced transaction below.

Example 1 Consider an advanced application consisting of the following set of subtransactions $\{T_1, T_2, T_3, T_4\}$: [subtransaction T_1] – Reserve a ticket on Airlines A; [subtransaction T_2] – Purchasing the Airlines A ticket; [subtransaction T_3] – Canceling the reservation; and, [subtransaction T_4] – Reserving a room in Resort C. The various kinds of dependencies that exists among the subtransactions are shown in Figure 1. There is a *begin-on-commit* dependency between T_1 and T_2 and also between T_1 and T_3. This means that neither T_2 or T_3 can start before T_1 has committed. There is an *exclusion* dependency between T_2 and T_3. This means that either T_2 can commit or T_3 can commit but not both. Finally, there is an *abort* dependency between T_4 and T_2. This means that if T_2 aborts then T_4 must abort. We can model this application as an advanced transaction $AT = < S, D >$, where $S = \{T_1, T_2, T_3, T_4\}$, $D = \{T_1 \rightarrow_{bc} T_2, T_1 \rightarrow_{bc} T_3, T_2 \rightarrow_{ex} T_3, T_2 \rightarrow_a T_4\}$.

An advanced transaction AT can be represented in the form of a labeled directed graph $G_T = < V, E >$. We refer to this graph as the dependency graph. The subtransactions T_1, T_2, ..., T_n correspond to the different nodes of the graph. Each dependency between subtransactions T_i and T_j is indicated by a directed edge (T_i, T_j) that is labeled with the name of the dependency. The dependency graph of the previous example is shown in Figure 1.

4 Properties of Dependencies

In this section we categorize and discuss the characteristics of dependencies that are needed for conflict analysis.

4.1 Classification of Dependencies

The dependencies mentioned in the previous section can be classified into two types: *event ordering* and *event enforcement*. The event ordering dependencies, as the name implies, order the execution of events in different subtransactions. The event enforcement dependencies, on the other hand, enforce the execution of certain events.

Definition 4 [Event Ordering] An *event ordering dependency* $T_i \rightarrow_d T_j$ is one in which the execution of some event in subtransaction T_i must precede the execution of some event in subtransaction T_j.

The commit, termination, begin, serial, begin-on-commit and begin-on-abort dependencies are event-ordering dependencies. The event ordering dependencies can be classified into ones that order the begin events and others that order the termination event.

Definition 5 [Event Enforcement] An *event enforcement dependency* $T_i \rightarrow_d T_j$ is one in which the execution of some event (begin, commit or abort) in subtransaction T_i requires the execution of some event in subtransaction T_j.

The strong commit, abort, weak abort, exclusive, force-commit-on-abort, force-begin-on-commit, force-begin-on-abort, force-begin-on-begin, and force-begin-on-terminate are examples of such dependencies. The event enforcement dependencies can be further classified into enforcing commit events, enforcing abort events and enforcing begin events. These again can be classified as commit-to-commit enforcing and abort-to-commit enforcing.

4.2 Implied Dependencies

In a generalized advanced transaction, the set of dependencies given by the user may not reflect the total set of

$T_i \rightarrow_? T_k$	$T_j \rightarrow_{sc} T_k$	$T_j \rightarrow_a T_k$	$T_j \rightarrow_{fca} T_k$
$T_i \rightarrow_{sc} T_j$	$T_i \rightarrow_{sc} T_k$	NA	NA
$T_i \rightarrow_a T_j$	NA	$T_i \rightarrow_a T_k$	$T_i \rightarrow_{fca} T_k$
$T_i \rightarrow_{fca} T_j$	$T_i \rightarrow_{fca} T_k$	NA	NA
$T_i \rightarrow_{ex} T_j$	NA	$T_i \rightarrow_{ex} T_k$	$T_i \rightarrow_{sc} T_k$

Table 1. Dependency Interaction Example

dependencies. Sometimes one or more dependencies specified by the user logical imply the existence of other dependencies. Sometimes some dependencies necessitate the existence of other dependencies for reasons of implementation.

4.2.1 Logically Implied Dependencies

First, we consider the case where dependencies have the transitive properties. A dependency of type x is said to be *transitive* if $T_i \rightarrow_x T_j$ and $T_j \rightarrow_x T_k$ implies that $T_i \rightarrow_x T_k$. The dependencies having the transitive property are strong commit, abort, termination, force-begin-on-begin, force-begin-on-terminate, begin, serial, begin-on-commit and begin-on-abort.

Implicit dependencies can also exist due to the interaction of a number of dependencies. Dependencies specified between different pair of subtransactions may interact with each and they may imply another dependency. The dependencies $T_i \rightarrow_{d_1} T_j$ and $T_j \rightarrow_{d_2} T_k$ may imply $T_i \rightarrow_{d_3} T_k$. For instance, the dependencies $T_i \rightarrow_{ex} T_j$ and $T_j \rightarrow_{fca} T_k$ imply the existence of $T_i \rightarrow_{sc} T_k$. Examples of such dependencies are shown in Table 1.

Other type of interactions are also possible. If there are dependencies $T_i \rightarrow_{d_x} T_j$ and $T_k \rightarrow_{d_y} T_j$, then there may exist relationships between $T_i \rightarrow_{d_z} T_k$. For instance, $T_i \rightarrow_{bc} T_j$ and $T_i \rightarrow_{ex} T_k$ imply $T_j \rightarrow_{ex} T_k$. For lack of space, we do not show all such possibilities.

4.2.2 Implementation Specific Implied Dependencies

In the preceding discussion we have classified the dependencies into event enforcement and event ordering types. One point needs to be mentioned. Sometimes, an event enforcement dependency imposes an execution order on subtransactions. In this case, an event ordering dependency is implied, and we consider it as an implicit dependency. For instance, consider the strong commit dependency $T_i \rightarrow_{sc} T_j$. This dependency requires T_j to commit if T_i commits, and it does not specify an order of execution. If T_i commits and subsequently T_j aborts, the dependency will be violated. Since the commitment of a subtransaction cannot be guaranteed, this possibility exists. Thus, in order to ensure the satisfaction of the dependency, we require T_j to commit before T_i. Moreover, if T_j aborts for any reason, then to maintain the dependency T_i should not be allowed to commit. In other words, to satisfy this dependency we need to enforce an execution order. Thus, whenever there is a dependency of the form $T_i \rightarrow_{sc} T_j$, there are implicit dependencies of the form $T_j \rightarrow_c T_i$ and $T_j \rightarrow_a T_i$.

Similar problems exist with the abort and force-commit-on-abort dependencies as well. The event enforcement dependency $T_i \rightarrow_a T_j$, requires T_j to abort if T_i aborts. If an execution order is not specified T_j may commit before T_i aborts, and the dependency will not be satisfied. Here also, we assume there is an implicit commit dependency of the form $T_i \rightarrow_c T_j$. We can make similar arguments and show that for the event enforcement dependency $T_i \rightarrow_{fca} T_j$, we need an implicit dependency of the form $T_j \rightarrow_{bc} T_i$.

Conversely, sometimes event ordering dependencies require that some event must occur or some event must not occur. In other words, they may imply the presence or absence of some event enforcement dependencies. An example will help illustrate this point. Consider, for instance, the begin-on-abort dependency $T_i \rightarrow_{ba} T_j$. This requires that T_j cannot begin until T_i aborts. In other words, if T_i commits, T_j should not begin. This is not unexpected because $c_i \Rightarrow \neg b_j$ can be derived from $b_j \Rightarrow (a_i \prec b_j)$. Similarly, the event ordering dependency $T_i \rightarrow_{bc} T_j$ requires T_i to commit before T_j can begin. In other words $a_i \Rightarrow \neg b_j$. The event ordering dependency $T_i \rightarrow_t T_j$ does not allow T_j to terminate unless T_i has already done so. In other words $\neg e_i \Rightarrow \neg e_j$ where e_i, e_j denote the termination event of T_i and T_j respectively.

4.3 Composite Dependencies

Our model allows us to specify multiple kinds of dependencies between any pair of subtransaction in an advanced transaction. Such dependencies are called composite dependencies. Composite dependencies are often needed to express more powerful coordination controls over subtransactions.

Definition 6 [Composite Dependency] A composite dependency between a pair of subtransactions T_i, T_j in a generalized advanced transaction model, denoted by $T_i \rightarrow_{d_1,d_2,\ldots,d_n} T_j$, is obtained by combining two or more dependencies $d_1, d_2, \ldots, d_n$. The effect of the composite dependency is the conjunction of the constraints imposed by the individual dependencies $d_1, d_2, \ldots, d_n$.

The constraints of one dependency could be completely covered by the constraints of another dependency. We refer this relationship as inclusion.

Definition 7 [Inclusion Relation] Let $T_i \rightarrow_{d_x} T_j$ and $T_i \rightarrow_{d_y} T_j$ be a pair of dependencies defined over T_i and T_j. The dependency d_x is said to include dependency d_y, denoted by $d_y \subseteq d_x$ if the satisfaction of dependency d_x implies the satisfaction of dependency d_y. The dependency d_y in this case is referred to as included dependency and d_x is the including dependency. The inclusion relationship is reflexive, transitive and anti-symmetric.

Dependency	Includes
$T_i \longrightarrow_a T_j$	wa
$T_i \longrightarrow_{fbb} T_j$	fbc, fba, fbt
$T_i \longrightarrow_{fbt} T_j$	fba, fbc
$T_i \longrightarrow_t T_j$	c
$T_i \longrightarrow_s T_j$	b, t
$T_i \longrightarrow_{bc} T_j$	t, b, s, c
$T_i \longrightarrow_{ba} T_j$	t, b, s

Table 2. Inclusion Relationship

For instance, the abort dependency includes the weak abort dependency. This is because whenever the abort dependency is satisfied, the weak abort dependency will also be satisfied. Similarly, we can show that the serial dependency includes the begin dependency and the termination dependency. A complete list of inclusion relations is given in Table 2.

5 Conflicts of Dependencies

We now inspect different kinds of dependency conflicts that are possible in our generalized advanced transactions.

Definition 8 [Conflict] Two dependencies d_x and d_y are said to conflict if the constraints imposed by the dependency d_x makes it impossible to satisfy the constraints of d_y.

In the following subsections, we describe the different kinds of conflicts.

5.1 Conflicts in Composite Dependencies

Conflicts can occur in composite dependencies. For instance $T_i \rightarrow_{bc} T_j$ and $T_i \rightarrow_{fba} T_j$ conflict. First, we discuss how conflicts can occur in composite dependencies. We later discuss how conflicts may occur in a set of dependencies. For instance, a set of dependencies $T_i \rightarrow_b T_j$, $T_j \rightarrow_b T_k$, and $T_k \rightarrow_b T_i$ conflict since the constraints of these dependencies can never be satisfied together.

Example 2 For instance, consider the dependencies $T_i \rightarrow_{sc} T_j$ and $T_i \rightarrow_{ex} T_j$. The strong commit dependency requires that if T_i commits, then T_j should commit. The exclusion dependency says that if T_i commits, then T_j should abort. When T_i commits, it is impossible to satisfy both dependencies.

The possible conflicts can be found out by using a conjunction on the conditions imposed by the dependencies, and evaluating the conjunct to find whether the conjunct will be satisfied for all possible executions. The set of all composite dependency conflicts is given in Table 3.

Dependency	Conflicting dependency
$T_i \longrightarrow_{sc} T_j$	ex, bc, s, c, a, t, ba
$T_i \longrightarrow_a T_j$	fca, sc
$T_i \longrightarrow_{wa} T_j$	fca
$T_i \longrightarrow_{ex} T_j$	sc
$T_i \longrightarrow_{fca} T_j$	a, b, ba, bc, wa, s
$T_i \longrightarrow_{fbc} T_j$	ba
$T_i \longrightarrow_{fbb} T_j$	ba, bc
$T_i \longrightarrow_{fba} T_j$	bc
$T_i \longrightarrow_{fbt} T_j$	ba, bc
$T_i \longrightarrow_c T_j$	sc
$T_i \longrightarrow_t T_j$	sc
$T_i \longrightarrow_b T_j$	fca
$T_i \longrightarrow_s T_j$	sc, fca
$T_i \longrightarrow_{bc} T_j$	$sc, fca, ba, fbt, fbb, fba$
$T_i \longrightarrow_{ba} T_j$	$bc, fca, fbt, fbb, fbc, sc$

Table 3. Composite Dependency Conflict

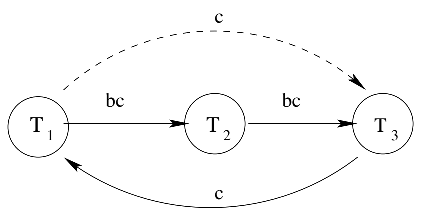

Figure 2. Conflicts with Event Ordering Type Dependencies

5.2 Conflicts between Event Ordering Dependencies

The event ordering dependencies, as the name implies, imposes an execution order on the events. It is possible for multiple event ordering dependencies to require an execution order that is not feasible in practice. To detect such infeasible requirements, we draw the graph for the advanced transaction model and check for the presence of some cycles.

Example 3 For instance, consider the example in figure 2 (a), $T_1 \rightarrow_{bc} T_2, T_2 \rightarrow_{bc} T_3, T_3 \rightarrow_c T_1$. This set of event ordering dependencies will lead to a conflict. To execute this advanced transaction, the events will be put in the following order: $c_1 \prec b_2 \prec c_2 \prec b_3 \prec c_3 \prec c_1$. The situation $c_1 \prec c_1$ is not possible because an event cannot precede itself.

Such conflicts can be detected by checking for the presence of cycles. The edges in the cycles are labeled with some event ordering dependencies. These event ordering dependencies must be related by an inclusion relationship.

5.3 Conflicts Between Event Enforcement Dependencies

An event enforcement dependency requires a certain event to happen. The events of interest are commit, abort and begin. Since event enforcement dependencies do not prohibit events from happening, the only conflicts possible are those in which one dependency requires some subtransaction T_i to commit and another requires T_i to abort. For illustration, consider the node T_k of a generalized advanced transaction model that has multiple incoming edges, namely (T_i,T_k) and (T_j,T_k).

Consider the case where $T_i \rightarrow_{sc} T_k$ and $T_j \rightarrow_{ex} T_k$ exist in the graph. If both T_i and T_j commit, then T_k is required to commit and abort which is not possible. In such a case we have a conflict, unless some other dependencies exist between T_i and T_j which prevent the simultaneous commitment of T_i and T_j. One way to check this is to see if T_i and T_j are connected by these kinds of edges. If there is an edge connecting T_i and T_j that implies $c_i \Rightarrow \neg c_j$, for instance, if either $T_i \rightarrow_{ex} T_j$ or $T_i \rightarrow_{ba} T_j$ or $T_j \rightarrow_{ba} T_i$ exists, then T_i and T_j do not commit together. If any one of these dependencies exists, the $T_i \rightarrow_{sc} T_k$ and $T_j \rightarrow_{ex} T_k$ dependencies are not conflicting. However, if none of these dependencies exists between T_i and T_j, we have a problem.

The next problematic case is when $T_i \rightarrow_{fca} T_k$ and $T_j \rightarrow_a T_k$. When T_i and T_j abort, we have a problem because T_k is required to both commit and abort. We do not have a problem if dependencies between T_i and T_j ensure that T_i and T_j do not abort at the same time. This is possible if either $T_i \rightarrow_{fca} T_j$, $T_j \rightarrow_{fca} T_i$, $T_i \rightarrow_{bc} T_j$, or $T_j \rightarrow_{bc} T_j$ exist. Each of these dependencies ensure that $a_i \Rightarrow \neg a_j$ and both a_i and a_j do not occur together.

The last case is when $T_i \rightarrow_{fca} T_k$ and $T_j \rightarrow_{ex} T_k$. Here again, we have a problem if T_i aborts and T_j commits. The presence of any of the following dependencies $T_i \rightarrow_a T_j$, $T_j \rightarrow_{ba} T_i$, or $T_j \rightarrow_{sc} T_i$ ensure that a_i and c_j do not occur at the same time. This is because $a_i \Rightarrow c_j$ can be inferred from any of the above dependencies.

One might think that there is a problem with the dependencies $T_i \rightarrow_{sc} T_k$ and $T_j \rightarrow_a T_k$. If T_i commits and T_j aborts, then the two dependencies may require T_k to commit and abort. This is clearly not possible. But the very nature of dependencies do not allow T_i to commit and T_j to abort. This is because the dependency $T_i \rightarrow_{sc} T_k$ implies the existence of the dependency $T_k \rightarrow_a T_i$. Since abort dependency is transitive in nature, there is an implicit dependency $T_j \rightarrow_a T_i$. The possibility of T_i committing when T_j aborts does not arise in this case.

5.4 Conflicts between Event Ordering and Event Enforcement Dependencies

There are two reasons why conflicts may occur between event enforcement dependencies and event ordering dependencies. The first reason is because some event enforcement dependencies impose an execution order (please refer to Section 4.2.2). Such execution orders may conflict with other event ordering dependencies. For instance, the strong commit dependency $T_i \rightarrow_{sc} T_j$ implies the dependency $T_j \rightarrow_c T_i$. This additional implied dependency might give rise to conflicts. To detect such conflicts, we insert edges for such implicit dependencies. As outlined in Section 5.2, we check for the presence of cycles to detect such conflicts.

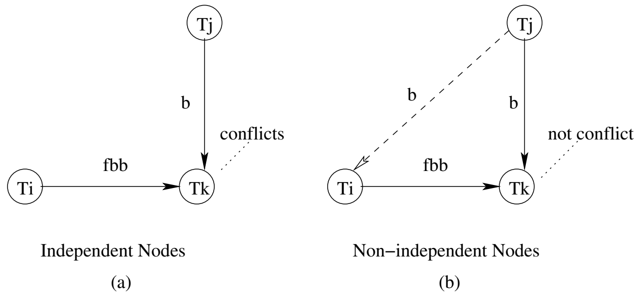

Figure 3. Conflicts with a Force-begin-on-begin Dependency and a Begin Dependency

The second reason why conflicts can occur is because sometimes event ordering dependencies require the prohibition of some events. Specifically, there are some event ordering dependencies $T_i \rightarrow_x T_j$ that do not allow T_j to begin until T_i has completed some event e_i. In other words, in the absence of occurrence of e_i, b_j cannot take place.

First, let us consider the case $T_i \rightarrow_{fbb} T_k$ and $T_j \rightarrow_b T_k$. Now if T_i begins and T_j does not begin, we have a problem: T_k is required to begin and not begin. Please refer to the figure 3. If the begin operations of T_i and T_j occur independently, then the two given dependencies are impossible to satisfy. However, if there exist some dependency between T_i and T_j that ensure that b_i and b_j will never occur together, that is, $b_i \Rightarrow b_j$, then we do not have a problem. The dependency $T_i \rightarrow_{fbb} T_j$ ensure this and so does $T_j \rightarrow_b T_i$. If either of these two dependencies do not exist between T_i and T_j, then the dependencies $T_i \rightarrow_{fbb} T_k$ and $T_j \rightarrow_b T_k$ cause a conflict.

Similarly, consider the case of dependency $T_i \rightarrow_{fbb} T_k$ and $T_j \rightarrow_{ba} T_k$. Suppose T_i has begun and T_j has not aborted. In such a scenario, T_k is required to begin as well as not begin. This is not possible. The only way this situation can be avoided is if there is some dependency between T_i and T_j that prevents the simultaneous occurrence of b_i and a_j. In other words, if $b_i \Rightarrow a_j$ can be inferred from the dependency that exists between T_i and T_j, then we do not have a problem. This happens when $T_j \rightarrow_{ba} T_i$ exists. When this dependency exist, then we may have a situation that is impossible to satisfy. We have a similar problem for the case when $T_i \rightarrow_{fbb} T_j$ and $T_j \rightarrow_{bc} T_k$. For this case, we need the presence of the dependency $T_j \rightarrow_{bc} T_i$ to avoid the conflict.

The dependencies fba, fbc and fbt can create similar problems. To detect such conflicts, we check the incoming edges for each node. If the incoming edges impose conflicting requirements, we must ensure that the conflicting requirements should not be simultaneously enforceable. In other words, we must ensure that the subtransactions from which the conflicting edges originate are related by dependencies such that the conflicting requirements are never simultaneously satisfied.

6 Algorithms for Detecting Conflicts

Since the behavior of an advanced transaction having dependency conflicts is unpredictable, it is important to analyze the transaction before it can be executed. Such analysis can be done statically. In this section, we present algorithms for automatically detecting conflicts.

To detect the conflicts, the structure of *dependency graph* described in Section 3 is useful. Recall that a dependency graph is composed of nodes and edges, where nodes are the subtransactions and edges are the dependencies in an advanced transaction.

The conflict detection algorithm is organized in three major steps - (i) generate initial dependency set, (ii) generate derived dependency set, and (iii) detect conflicts of dependencies. First, an initial dependency set could be built using the specification of the advanced transaction. Then, the derived dependency set will be generated, based on explicit edges in the initial dependency set. For conflict analysis, we need to insert all the implicit dependencies in the derived dependency set. The procedure of building up the derived dependency set is organized in a loop. In each iteration, new implicit dependencies, which could be derived from the current dependency set, are inserted into the derived dependencies set. The loop of operations will continue until no more new edge can be derived.

The rest is to detect the different kinds of conflicts. All these detection procedures are based on the derived dependency set. For each type of dependency conflicts discussed in Section 5, a corresponding procedure is needed to detect this type of conflict. These procedures could be performed in any order, since detecting for each type of conflicts is independent.

The algorithms for dependency conflict detection and the implementation details are described following.

Step 1: Generate the initial dependency set (IDS) for the advanced transaction. This is based on the specifications of the advanced transaction, which are provided by the application developer. The initial dependency set records every dependency $T_i \rightarrow_{d_x} T_j$ in the specification as an edges of the form (T_i, T_j, d_x), where T_i is the source node, T_j is the destination node, and d_x in the type of dependency associated between T_i and T_j.

Algorithm 1 Generate the Initial Dependency Set
Input: the GAT specification $AT_t = < S, D, C >$
Output: an initial dependency set IDS
Procedure GenerateIDS(AT_t)

```
STEP 1: generate initial dependency set IDS
Begin
   //Build initial dependency set based on the specification
   $IDS = \{\}$; // set of edges
   $INS = \{\}$; // set of nodes
   For every dependency $(T_i \rightarrow_{d_x} T_j) \in AT_t(D)$
        // insert a dependency of type $x$ from $T_i$ to $T_j$
        generate an edge $(T_i, T_j, d_x)$;
        $IDS = IDS + (T_i, T_j, d_x)$;
   For every subtransaction $(T_i) \in AT_t(S)$
        generate a node $(T_i)$;
        $INS = INS + (T_i)$;
End
```

Step 2: Generate the derived dependency set (DDS). The derived dependency set will be built based on the initial dependency set generated in step 1. This procedure is organized in loops. In every iteration, the algorithm scans all the dependencies currently in the DDS and check whether new edges could be implied by the existing edges. (i) for each dependency implying an implicit dependency (like sc, fca, a), insert the corresponding edge; (ii) for each pair of dependencies with interaction relationships, insert the corresponding implied edge. The loops are repeated until no more new edges can be derived.

In each round, newly generated implicit edges will be put into the set and they will be marked as *unchecked*. These edges newly inserted could possibly imply more new implicit edges, and their relationship will be checked in the next round. The algorithm keeps checking new edges round by round, until in certain round there is no new edge can be derived. Now we have the derived dependency set built up.

Algorithm 2 Generate the Derived Dependency Set
Input: the IDS of advanced transaction
Output: a derived dependency set (DDS) of the advanced transaction
Procedure GenerateDDS

```
STEP 2: generate the derived dependency set DDS
Begin
  // initiate the derived dependency set
  done == false;
  initiate $DDS == IDS$;
  mark every edge in $DDS$ as unchecked
  While (done = false)
  Begin
     For each edge $(T_i, T_j, d_x) \in DDS$
     Begin
        // If this dependency implies a relationship,
        // insert the implicit edge
        If this edge is unchecked
```

```
          If $d_x = sc$
            generate an edge $(T_j, T_i, c)$;
          else If $d_x = a$
              generate an edge $(T_i, T_j, c)$;
          else If $d_x = fca$
              generate an edge $(T_j, T_i, bc)$;
        // insert the implicit edges implied by
        // dependency interaction
        For each $(T_j, T_k, d_p) \in DDS$ and is unchecked
            If $(T_i, T_j, d_x)$ and $(T_j, T_k, d_p)$
            imply an interaction dependency $d_t$
              generate an edge $(T_i, T_k, d_t)$;
        For each $(T_i, T_k, d_q) \in DDS$ and is unchecked
            If $(T_i, T_j, d_x)$ and $(T_i, T_k, d_q)$
            imply an interaction dependency $d_s$
              generate an edge $(T_j, T_k, d_s)$;
      End
      // mark existing edges as checked;
      For every edge $\in DDS$
          set edge as checked
      // insert newly generated edges,
      // and make them as unchecked;
      For every newly generated edge $e_{new}$ in this round
          set edge $e_{new}$ as unchecked
          $DDS = DDS + e_{new}$
      If there is no new edge inserted in this round
        done == true;
  End
End
```

Step 3. Detect conflicts with the composite dependencies, using the DDS obtained in step 2. When multiple dependencies are specified between a pair of transactions, we need to ensure that this composite dependency does not impose conflicting requirements. The table of conflicting dependency pairs is shown in Table 3. In this step, we visit all the edges one by one, and find each pair of edges with the same source node and destination node; then, we check whether the two dependencies are conflicting or not. If they are conflicting, report error.

Algorithm 3 DetectingCompositeDependencyConflicts
Input: the derived dependency set DDS
Output: message identifying whether there are conflicts
Procedure DetectCompositeDependency

```
STEP 3: detect conflicts with composite dependencies
Begin
  For each edge $(T_i, T_j, d_x) \in DDS$
  Begin
    For each edge $(T_m, T_n, d_y)$ other than $(T_i, T_j, d_x)$
    Begin
      // check if they are a composite dependency
      If $T_i = T_m$ AND $T_j = T_n$
      Begin
        check the conflicting table for $d_x$
        $\{Conflict\_Set\}$ = all conflict dependencies
        // check whether they are conflicting
        If $d_y \in \{Conflict\_Set\}$
          return "conflict"
      end
    end
    return "no conflict";
End
```

Step 4. Detect conflicts with the event ordering type of dependencies, using the DDS set. We detect whether there is a cycle of a common type of event ordering dependencies existing in the DDS graph. If there is such a cycle, report conflicts. For every pair of connected edges (T_i, T_j, d_1) and (T_j, T_k, d_2), if d_1 and d_2 are of same ordering type, we insert an imaginary edge (T_i, T_k, d_1); else, if d_1 and d_2 are ordering type and they are of inclusion relationship, (assuming the included dependency is d_z) we insert an imaginary edge (T_i, T_k, d_1). With these imaginary edges inserted, if there exists a cycle of a common type d_z of ordering dependencies, we will eventually meet a pair of edges (T_p, T_q, d_z) and (T_q, T_p, d_z), and it means a conflict. If there is no such cycles, we report there is no conflict.

Algorithm 4 DetectingOrderingDependencyConflicts
Input: the derived dependency set DDS
Output: message identifying whether there are conflicts
Procedure DetectOrderingConflicts

```
STEP 4: detect conflicts with event ordering dependencies
Begin
  $\{ImaginarySet\} = DDS$;
  //Check cycles in event ordering dependencies
  For each ordering type edge $(T_i, T_j, d_1) \in \{ImaginarySet\}$,
  where $d_1 \in \{c, t, s, b, ba, bc\}$
  Begin
    // search for cycle $T_i \rightarrow_d T_j$ and $T_j \rightarrow_d T_i$
    If ( (($T_j, T_i, d_2) \in \{ImaginarySet\}$) AND
    ( ($d_1 = d_2$) OR ($d_1 \subseteq d_2$) OR ($d_2 \subseteq d_1$) ) )
      return "conflict"
    // insert imaginary edges of transitive relationship
    elseIf $(T_j, T_k, d_2) \in \{ImaginarySet\}$ AND
    ( ($d_1 = d_2$) OR ($d_1 \subseteq d_2$) OR ($d_2 \subseteq d_1$) )
        If ($d_1 = d_2$) OR ($d_1 \subseteq d_2$)
          $d_z == d_1$
        else If $d_2 \subseteq d_1$
            $d_z == d_2$
        $\{ImaginarySet\} = \{ImaginarySet\} + (T_i, T_k, d_z)$
  End
  return "no conflict"
  remove $\{ImaginarySet\}$
End
```

Step 5. Detect conflicts between pairs of event enforcement dependencies who are incident on the same node. If multiple edges are incident on a same node, check whether the edges cause any problem. Report conflicts if there are problems.

The major challenge is how to identify whether the two source nodes, with dependencies incident on the same destination node, are independent or not. By generating all the derived edges, the above algorithm reduces the complexity of this checking. It does this by checking whether certain kinds of desired dependencies (either explicit or implicit) exist between the two source nodes T_i and T_j, if these dependencies are present, then T_i and T_j are not independent, and the dependencies are not conflicting; otherwise the dependencies are conflicting. For example, for dependencies $T_i \rightarrow_{sc} T_k$ and $T_j \rightarrow_{ex} T_k$, if there exists a derived edge $T_k \rightarrow_{ex} T_i$, this dependency specification does not cause a conflict.

Algorithm 5 DetectingEnforcementDependencyConflicts
Input: the derived dependency set DDS
Output: message identifying whether there are conflicts
Procedure DetectEnforcementConflict

```
STEP 5: detect conflicts of event enforcement dependencies
Begin
  //Check conflicts with event enforcement dependencies
  For every node $T_k \in NS$
  Begin
    If exists edges $(T_i, T_k, sc)$ and $(T_j, T_k, ex) \in DDS$
      If there exists no
      $((T_i, T_j, ex)$ OR $(T_i, T_j, ba)$ OR $(T_j, T_i, ba))$
        return "conflict";
    If exists edges $(T_i, T_k, fca)$ and $(T_j, T_k, ex) \in DDS$
      If there exists no
      $((T_i, T_j, a)$ OR $(T_j, T_i, ba)$ OR $(T_j, T_i, sc))$
        return "conflict";
    If exists edges $(T_i, T_k, fca)$ and $(T_j, T_k, a) \in DDS$
      If there exists no $((T_i, T_j, fca)$ OR $(T_i, T_j, bc)$
      OR $(T_j, T_i, fca))$ OR $(T_j, T_i, bc))$
        return "conflict";
  End
  return "no conflict"
End
```

Step 6. Detect conflicts between pairs of event enforcement dependency and event ordering dependency, who are incident on the same node. Please refer to the discussion in section 5 and Figure 3 for this kind of dependency conflicts. The structure of this procedure is similar to step 5. We skip the details of code here due to page limit.

7 Conclusion

In this work, we address the problem of dependency conflicts existing in advanced transaction models. We have discussed how dependencies could affect the execution of advanced transactions, and present the means how to analyze conflicts of dependencies. In this paper, we describe the properties of dependencies, and study different kind of dependency conflicts, we also propose algorithms to automatically detect the conflicts. The conflict free dependencies ensures that the dependencies are physically realizable in execution. It ensures that the advanced applications are free from deadlocks and unavailability issues.

A lot of work still remains. In future, we would like to study the impact of the dependencies on the transaction scheduler and recovery manager. The subtransaction scheduler must take into account the nature of dependencies before scheduling the operations of a subtransaction. The recovery manager of the traditional subtransaction processing system focuses on restoring consistency when a crash occurs. The recovery manager for advanced transaction must not only restore consistency but must also ensure that the dependencies are not violated in the recovery process.

References

[1] M. Ansari, L. Ness, M. Rusinkiewicz, and A. Sheth. Using Flexible Transactions to Support Multi-System Telecommunication Applications. In *Proceeding of the 18th International Conference on Very Large DataBases*, August 1992.

[2] V. Atluri, W.-K. Huang, and E. Bertino. An Execution Model for Multilevel Secure Workflows. In *11th IFIP Working Conference on Database Security and Database Security, XI: Status and Prospects*, pages 151–165, August 1997.

[3] P. C. Attie, M. P. Singh, A. P. Sheth, and M. Rusinkiewicz. Specifying and enforcing intertask dependencies. In *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, pages 134–145. Morgan Kaufmann, 1993.

[4] E. Bertino, G. Chiola, and L. V. Mancini. Deadlock detection in the face of transaction and data dependencies. *Lecture Notes in Computer Science: 19th Int. Conf. on Application and Theory of Petri Nets, ICATPN'98, Lisbon, Portugal, June 1998*, 1420:266–285, June 1998.

[5] A. Biliris, S. Dar, N. Gehani, H. Jagadish, and K. Ramamritham. ASSET: A System for Supporting Extended Transactions. In *Proceedings of ACM SIGMOD International Coference on Management of Data*, May 1994.

[6] P. Chrysanthis and K. Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 194–203, May 1990.

[7] P. K. Chrysanthis. ACTA, A Framework for Modeling and Reasoning about Extended Transactions Models. Ph.D. Thesis, September 1991.

[8] P. K. Chrysanthis and K. Ramamritham. Synthesis of Extended Transaction Models Using ACTA. *ACM Transactions on Database Systems*, 19:450–491, September 1994.

[9] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book.* Prentice-Hall, 2002.

[10] L. V. Mancini, I. Ray, S. Jajodia, and E. Bertino. Flexible transaction dependencies in database systems. *Distributed and Parallel Databases*, 8:399–446, 2000.

[11] J. E. Moss. Nested Transactions: an approach to reliable distributed computing. PhD Thesis 260, MIT, Cambridge, MA, April 1981.

[12] M. Prochazka. Extending transactions in enterprise javabeans. Tech. Report No. 2000/3, Dep. of SW Engineering, Charles University, Prague, January 2000.

[13] M. Rusinkiewicz and A. P. Sheth. Specification and execution of transactional workflows. In *Modern Database Systems 1995*, pages 592–620, 1995.

[14] K. Schwarz, C. Türker, and G. Saake. Analyzing and formalizing dependencies in generalized transaction structures. In *Proceedings of the Third International Conference on Integrated Design and Process Technology*, pages 55–62, 1998.

[15] K. Schwarz, C. Türker, and G. Saake. Integrating execution dependencies into the transaction closure framework. *International Journal on Cooperative Information Systems*, 8(2-3):111–138, 1999.

[16] M. P. Singh. Semantical considerations on workflows: An algebra for intertask dependencies. In *Proceedings of the Fifth International Workshop on Database Programming Languages*, Electronic Workshops in Computing. Springer, 1995.

[17] H. Wuchter and A. Reuter. The ConTract Model. In *Database Transaction Models for Advanced Applications, A. K. Elmagarmid Ed., Morgan Kaufmann Publishers*, pages 219–263, 1992.