Analysis of Dependencies in Advanced Transaction Models

Abstract

Transactional dependencies play an important role in coordinating and executing the subtransactions in advanced transaction processing models, such as, nested transactions and workflow transactions. Researchers have formalized the notion of transactional dependencies and have shown how various advanced transaction models can be expressed using different kinds of dependencies. Incorrect specification of dependencies can result in unpredictable behavior of the advanced transaction, which, in turn, can lead to unavailability of resources and information integrity problems. In this work, we focus on how to correctly specify dependencies in an advanced transaction. We enumerate the different kinds of dependencies that may be present in an advanced transaction and classify them into two broad categories: event ordering and event enforcement dependencies. Different event ordering and event enforcement dependencies in an advanced transaction often interact in subtle ways resulting in conflicts and redundancies. We describe the different types of conflicts that can arise due to the presence of multiple dependencies and describe how one can detect such conflicts. An advanced transaction may also contain redundant dependencies – these are dependencies that can be logically derived from other dependencies. We show how such extraneous dependencies can be eliminated to get an equivalent set of dependencies that has the same effect as the original set. Our dependency analysis is done in the context of a generalized advanced transaction model that is capable of expressing different kinds of advanced transactions.

1 Introduction

Although the traditional transaction processing model [6] has proved very successful for typical applications, it is inadequate for processing specialized transactions. Examples include applications having long-duration transactions, or applications having a set of activities that must be properly coordinated to ensure correct behavior. To address this shortcoming, researchers have proposed various advanced transaction processing models [3, 10, 11, 12, 14, 15, 19, 21, 24, 28, 29, 33] that are suitable for executing such specialized applications. These advanced transaction processing models coordinate their activities using different kinds of dependencies. The ACTA model [11] was one of the earliest works to formalize the dependencies. However, research on the analysis and interaction of dependencies has received less attention.

Once the interaction of dependencies and the constraints they impose on transaction processing is well understood, new kinds of transaction models can be developed. Another promising area in which dependency analysis is needed is web services transactions [2, 26]. A web services transaction specification describes an extensible coordination framework for coordinating tasks. The web services transaction is often a long duration activity and the tasks are executed asynchronously by different parties. Dependencies can be used to coordinate the different tasks of a web services transaction. Our analysis techniques can be applied to the web services transactions to ensure that the dependencies used to specify the coordination do not conflict and it is indeed possible to complete such a transaction.

The goal in this work is to formalize the correctness of dependency specifications. To illustrate our analysis, we specify dependencies in the context of a generalized advanced transaction. A generalized advanced transaction is composed of subtransactions which have dependencies specified among them. Dependencies coordinate the execution of the subtransactions in the generalized advanced transaction. A generalized advanced transaction is also associated with one or more completion sets. Completion sets specify the subtransactions that need to commit and the order in which they must commit for the successful execution of the transaction.

The generalized advanced transaction model can be customized to specify different kinds of advanced transactions. This customization is done by limiting the types of dependencies that can exist between various subtransactions. We do not limit the kinds of dependencies that can exist between the different subtransactions because our purpose is to analyze the interaction of all the different kinds of dependencies that are present in advanced transaction processing models.

We begin by describing the kinds of dependencies that can exist in advanced transaction models. The dependencies define the relationships that may exist between different events, such as, begin, commit, or abort, of two subtransactions in a generalized advanced transaction. One class of dependencies specify ordering relationships among the events of two subtransactions; we call such dependencies *event ordering dependencies*. The other class of dependencies require that some event must happen when a certain event has occurred in the other subtransaction; we call such dependencies *event enforcement dependencies*. We also show that in practical systems how certain event enforcement dependencies imply an execution ordering.

Our formal specification of dependencies allows us to combine two or more dependencies between a pair of subtransactions in the generalized advanced transaction. To formalize, what dependencies can be combined we introduce the notion of *inclusion relation* and *conflict relation* that can exist between dependencies specified on a pair of subtransactions. A dependency d_x is said to include another dependency d_y if the constraints imposed by d_y can be logically derived from the constraints imposed by d_x. A dependency d_x is said to conflict with another dependency d_y if the constraints imposed by d_x violate the constraints imposed by d_y. Dependencies between a pair of subtransactions can be combined if they do not conflict with each other, that is, they do not impose constraints on the subtransaction execution that contradict each other. Moreover, if two dependencies related by inclusion relationship is combined, the dependency that is included by the other dependency is eliminated from the combined dependency. The combined dependency so obtained is termed as *composite dependency*.

Even though composite dependencies do not have dependencies related by the inclusion or the conflict relationships, it does not guarantee the correct specification for a given set of dependencies. In this work, the correctness of dependency specification is formalized in terms of two properties: *conflict-free property* and *minimality property*. The conflict-free property ensures that a given set of dependencies is physically realizable. That is, the dependencies do not pose constraints on the execution that contradict each other and are therefore impossible to satisfy in a given execution. If a given set of dependencies has conflicts, we cannot give any assurance about the correctness of the behavior of the generalized advanced transaction. Thus, we must ensure that the dependencies do not conflict with each other. We show how dependency conflicts occur and thereby help make the dependency specification conflict-free.

Enforcement of dependencies in advanced transaction models incurs additional overhead.

This is because the database management system must perform additional checks and carry out operations to ensure the satisfaction of the dependencies. Thus, the presence of extraneous dependencies in a set of dependencies unnecessarily slows down performance. To address this issue, we propose the minimality property. The minimality property ensures that no dependency specified in a set of dependencies is extraneous. We show how to check for extraneous dependencies and how to minimize a given set of dependencies.

The rest of the paper is organized as follows. Section 2 describes some related work in this area. Section 3 describes our generalized transaction processing model that can be adapted to express different kinds of transactions. Section 4 categorizes the different kinds of dependencies that may be present in our generalized transaction. Section 5 shows what kinds of dependencies can be combined together between a pair of subtransactions. Section 6 focuses on how to eliminate extraneous dependencies from a given set of dependencies and formalizes the notion of equivalence of dependencies. Section 7 illustrates the kinds of conflicts present in generalized advanced transactions. Section 8 describes the effect of dependencies on the completion sets of a generalized advanced transaction. Section 9 discusses what it means for a generalized advanced transaction to be well-structured. Section 10 concludes the paper with some pointers to future directions.

2 Related Work

Chrysanthis and Ramamritham proposed ACTA [10] which is 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 transactions in terms of different dependencies between transactions, and in terms of transaction's effects on data objects. During the past few years, ACTA has been used extensively for specifying and reasoning about advanced transaction models. Biliris et al. [8] propose "ASSET" – A System for Supporting Extended Transactions. They extend a programming language by providing a set of transaction primitives. Beyond the traditional transaction primitives (e.g., begin, commit, abort, get_parent), it also introduces new ones for creating dependencies between transactions, resource delegation, and giving permissions to access acquired resources. However, no implementation based on a real architecture is given and interfaces of the proposed primitives are not defined precisely. None of the given examples uses dependencies; instead, dependencies are implicitly modeled by flow control constructs. Mancini et al. [23] present a multiform transaction model that can implement a wide range of advanced transactions. A multiform transaction is defined as a pair <T,C>, where T defines the set of transactions with the partial orders and completion dependency on these transactions, while C defines the coordinate blocks to coordinate the relationship among the transactions in set T. None of these work focus on the analysis of dependencies.

Bertino, Chiola, and L. V. Mancini discuss deadlock detection issues in advanced transaction models [7]. The authors show that deadlocks may arise in advanced transactions because of the interactions of transactional and data dependencies. Such deadlocks cannot be detected by the conventional deadlock detection algorithms. The authors propose an algorithm based on using an AND-OR graph for detecting such deadlocks. The proposed algorithm has a computational complexity linear in the number of nodes and edges of the AND-OR graphs.

The use of formal methods for specification and analysis of dependencies have been investigated by a number of researchers [1, 5, 9, 13, 20, 25, 30, 32]. Davulcu et al. [13] provide a framework for specifying, analyzing, and executing workflows based on Concurrent Transaction Logic. Using this logic, the authors describe how to verify whether a workflow is consistent and can be scheduled. Other properties can be verified using this approach. Mukherjee et al. [25] discuss the relative merits and demerits of analyzing workflows using temporal logic, event algebra, and Concurrent Transactional Logic. The Concurrent Transaction Logic is the most suitable for expressing workflows because it can handle generalized constraints and can represent control flow graphs with transition conditions on the arcs. Bonner [9] investigated the expressive power of Concurrent Transaction Logic and proposed some theoretical results in this regard. Vortex [17] workflows are not based on logic. However, abstractions can be derived from Vortex workflows. This abstraction can be represented in temporal logic which can be automatically verified using model checkers. Attie et al. [5] discuss how workflows can be represented as a set of intertask dependencies. The tasks in a workflow are described in terms of significant events. When an event is received for execution, it is checked against every dependency to identify whether the task can be accepted, rejected, or delayed. The dependencies are specified in Computational Tree Logic. This work does not deal with the verification issues. However, model checking can be used in this case. Singh shows how event algebra can be used for specifying workflows [30]. Using this algebra, temporal intertask dependencies can be expressed. However, it cannot represent conditions on transitions between tasks. It is also not clear whether a given set of constraints has redundancy in it or whether it can be implied by a set of constraints. J. Tang and J. Veijalainen [31] have proposed a way to enforce intertask dependencies in transactional workflows.

Adam, Atluri and Huang [1] have discussed modeling and analysis of workflows using Petri Nets. The authors show how to use Petri Nets to model the workflow system at a conceptual level. They identify some structural properties of Petri Nets and demonstrate its use for the analysis and verification of workflow specifications. The authors discussed different control-flow dependency types – strong-causal type, weak-causal type and precedence type. They also mentioned value dependencies and temporal dependencies. The authors show how to use Petri Nets to represent control-flow, value and temporal dependencies; they also demonstrate how to use Petri Nets to conduct analysis on workflow structures – like inconsistent dependency specifications, terminable with an acceptable state, feasible to complete execution with the given temporal constraints. This work is very useful as it demonstrates Petri Nets as an effective tool for modeling and analysis of workflows.

Formal methods are indeed extremely useful for doing rigorous analysis. However, using formal methods has two potential problems. First, it requires a high degree of expertise that practitioners may not possess. For example, consider the tools used in formal methods, such as theorem provers and model checkers, that may be used to partially automate the analysis. Theorem provers are extremely hard to use, require human intervention, and have demonstrated limited practical benefits. Model checkers are relatively easy to use. However, the problem in model checking is that it can verify only finite state machines. Most applications have infinite states. Thus, some kind of abstraction must be done that will convert the infinite state application to a finite state one. The abstraction must be done such that the properties being verified are not altered in the process. This, in our opinion, requires a high degree of expertise as well. Second, formal methods allow us to express the dependencies in logic but fail to capture the relationships that are imposed due to practical constraints. An example will help illustrate this point. Consider the strong commit dependency: $T_i \rightarrow_{sc} T_j$. It says, if T_i commits then T_j must also commit. Formally $c_i \Rightarrow c_j$. This dependency does not specify the order of commit operations of T_i and T_j. So 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 does exist. To ensure this 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 ensure this dependency in a practical application, we need to enforce two additional dependencies: $T_j \rightarrow_c T_i$ and $T_j \rightarrow_a T_i$. Note that, none of the dependencies $T_j \rightarrow_c T_i$ or $T_j \rightarrow_a T_i$ can be logically derived from $T_i \rightarrow_{sc} T_j$. These dependencies exist because of practical constraints. A logic-based approach will therefore fail to capture these relationships. We feel that analysis of dependencies is extremely important for many practical applications. Our work aims to explain the interactions and implications of the dependencies such that it can be readily used by developers designing such applications who may not be experts in formal methods.

Xin and Ray [34] discuss the dependencies in advanced transactions, show how dependencies can be classified into event enforcement and event ordering dependencies, and also present a conflict detection algorithm. The current work extends and formalizes the ideas presented in the previous work. The following are the extensions that are addressed in the current work. First, it lists the conditions that are sufficient for the different kinds of conflicts to occur and gives formal proofs as to why these are sufficient conditions. Second, it formalizes the concept of minimality of dependencies and gives algorithms to get a minimal set of dependencies. Third, it introduces the concept of equivalence of dependencies. Fourth, it illustrates how dependencies impact the set of subtransactions that must be completed in a generalized advanced transaction, and the order in which they must be completed. Fifth, it defines the notion of correct form of a generalized advanced transaction and its correct execution.

3 Generalized Advanced Transaction Model

In this work, we propose the concept of a generalized advanced transaction. The generalized advanced transaction can be customized for different kinds of transaction models by restricting the type of dependencies that can exist between the component transactions. A generalized advanced transaction is specified by a set of subtransactions, the dependencies between these subtransactions, and the completion sets. All subtransactions specified in a generalized advanced transaction may not execute or commit. A completion set gives the set of subtransactions that need to be committed for completing the generalized advanced transaction. Some of these subtransactions must be committed in a specific order – the completion set needs to specify this ordering relation. Moreover, the set of subtransactions that commit in a generalized advanced transaction model may vary with different instances of the generalized advanced transaction. Thus, a generalized advanced transaction may have multiple completion sets.

Definition 1 [Generalized Advanced Transaction] A *generalized advanced transaction* $T = < S, D, C >$ where S is the set of subtransactions in T, D is the set of dependencies between the subtransactions in S, and C is the set of completion sets in T. The completion set C_i, where $C_i \in C$, is a partially ordered set specified by $(CT_i, \ll_i)$ – CT_i is the set of subtransactions that must be committed and $\ll_i$ is the order in which they must be committed.

Definition 2 [Subtransaction] A *subtransaction* T_i is the smallest logical unit of work in an advanced transaction. It consists of a set of data operations (read and write) and subtransaction primitives (begin, abort, and commit). The begin, abort and commit primitives of subtransaction T_i are denoted by b_i, a_i, and c_i respectively. The execution of these primitives is often referred to as an event. Thus, we can have begin, commit, or abort events. The commit or abort events are referred to as termination events. In other words, termination events are a generalization of commit or abort events.

Definition 3 [States of a subtransaction] A subtransaction T_i can be in any of the following states: *unscheduled (un_i)*, *initiation (in_i)*, *execution (ex_i)*, *prepare (pr_i)*, *committed (cm_i)* and *aborted (ab_i)*. The state of a subtransaction can be changed by the execution of a primitive or by the scheduler of the database management system.

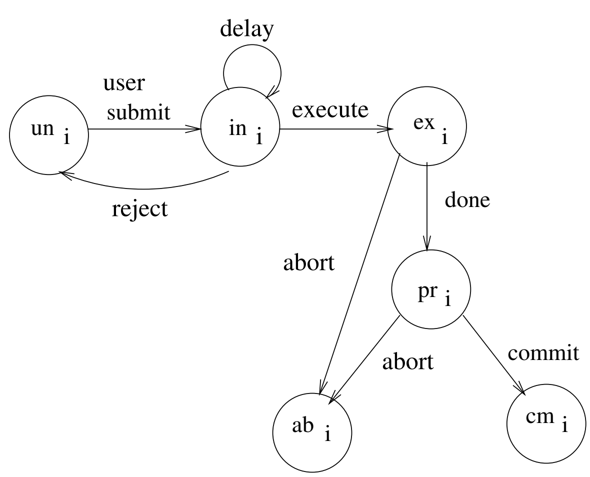

Figure 1: States of subtransaction T_i

The states of the subtransactions and state transitions are shown in Figure 1. These states are described below.

unscheduled (un_i)**:** T_i is the *unscheduled state* when it has not been submitted by the user. When the user submits T_i, it moves to the initiation state.

initiation (in_i)**:** T_i is in the *initiation state* when it has been submitted by the user and it is waiting to be executed. When T_i is selected for execution, it moves to the execution state. If T_i is rejected, then it moves to the unscheduled state.

execution (ex_i)**:** T_i is in the *execution state* when it has executed the begin primitive and it has not finished execution. From the execution state, T_i enters the aborted state if it completes unsuccessfully. If it completes successfully, it moves to the prepare state.

prepare (pr_i)**:** T_i is in the *prepare* state when it has completed execution and it is ready to commit. At this state, T_i can no longer unilaterally commit or abort. If T_i is aborted by the scheduler, it moves to the aborted state. Otherwise, it moves to the committed state.

committed (cm_i)**:** T_i is in the *committed* state after it has executed the commit primitive, that is, after it has committed.

aborted (ab_i)**:** T_i is in the *aborted* state after it has executed the abort primitive, that is, after it has aborted.

Note that, when a subtransaction T_i is in the committed or aborted state, its state cannot be changed any more and we say that T_i is in the final state.

Next, we describe the dependencies in our advanced transaction model. An advanced transaction model can have external, data flow, and control flow dependencies. External dependencies are caused by parameters external to the system, such as time. There can be various parameters external to the advanced transaction model. To limit the scope of our work, we do not consider external dependencies. Data flow dependencies are present when the output of a subtransaction, say, T_i is the input to another subtransaction T_j of the same advanced transaction. Since data flow dependencies imply the existence of control flow dependencies, we do not consider data flow dependencies separately in this paper. We focus on control flow dependencies only. Henceforth, we use the term 'dependency' to mean 'control flow dependency'.

Definition 4 [Dependency] A *dependency* specified between a pair of subtransactions T_i and T_j expresses how the execution of a primitive (begin, commit, and abort) of T_i causes (or relates to) the execution of the primitives (begin, commit, and abort) of another subtransaction T_j.

A set of dependencies has been defined in the work of ACTA [11]. A comprehensive list of transactional dependency definitions can be found in [4, 8, 11, 22, 27]. The different types of dependencies that typically occur in applications are the ones we analyze in this paper and are described below. In the following descriptions, T_i and T_j refer to the subtransactions; b_i, c_i, a_i refer to the events of T_i that are present in some history H; and the notation $e_i \prec e_j$ denotes that event e_i precedes event e_j in the history H.

[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 $e_i \in \{c_i, a_i\}$, $e_j \in \{c_j, a_j\}$.

[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, $b_j \Rightarrow (a_i \prec b_j)$.

[Weak Begin-on-commit dependency] ($T_i \rightarrow_{wbc} T_j$): If subtransaction T_i commits, subtransaction T_j can begin executing after T_i commits. Formally, $b_j \Rightarrow (c_i \Rightarrow (c_i \prec b_j))$.

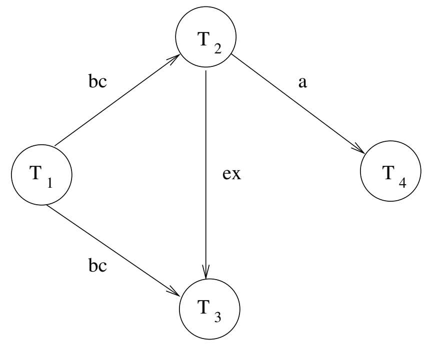

Figure 2: Dependencies in the advanced transaction of Example 1

A generalized advanced transaction T can be represented in the form of a directed graph $G_T =< V,E >$. 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 label on an edge (T_i,T_j), denoted by $L(T_i,T_j)$, is a set whose elements are the types of dependencies that exist from T_i to T_j.

We give an example of an advanced transaction below.

Example 1 Let $T =< S,D,C >$ be a generalized advanced transaction where $S = \{T_1,\ T_2,\ T_3,\ T_4\}$, $D = \{T_1 \rightarrow_{bc} T_2, T_1 \rightarrow_{bc} T_3, T_2 \rightarrow_{ex} T_3, T_2 \rightarrow_{a} T_4\}$, and $C = \{(\{T_1,T_2,T_4\},\{T_1 \ll T_2\}), (\{T_1,T_3\},\ \{T_1 \ll T_3\})\}$. The labels on each edge corresponds to the dependencies that exist between the tasks. $L(T_1,T_2) = \{bc\}$, $L(T_1,T_3) = \{bc\}$, $L(T_2,T_3) = \{ex\}$, $L(T_2,T_4) = \{a\}$. This transaction has two completion sets: $(\{T_1,T_2,T_4\},\{T_1 \ll T_2\})$ and $(\{T_1,T_3\},\{T_1 \ll T_3\})$. This generalized transaction can be represented graphically as shown in Figure 2.

A real world example of such a transaction may be a workflow associated with making travel arrangements. The subtransactions perform the following tasks. (i) Subtransaction T_1 – Reserves a ticket on Airlines A, (ii) Subtransaction T_2 – Purchases the Airlines A ticket, (iii) Subtransaction T_3 – Cancels the reservation, and (iv) Subtransaction T_4 – Reserves a room in Resort C. 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 nor T_3 can start before T_1 has committed. This ensures that the airlines ticket cannot be purchased or cancelled before a reservation has been made. The *exclusion* dependency between T_2 and T_3 ensures that if T_2 has committed and T_3 has begun executing, then T_3 must abort. In other words, if the airlines ticket has been purchased, then the airlines reservation cannot be cancelled. Finally, there is an *abort* dependency between T_4 and T_2. This means that if T_2 aborts then T_4 must abort. In other words, if the airlines ticket

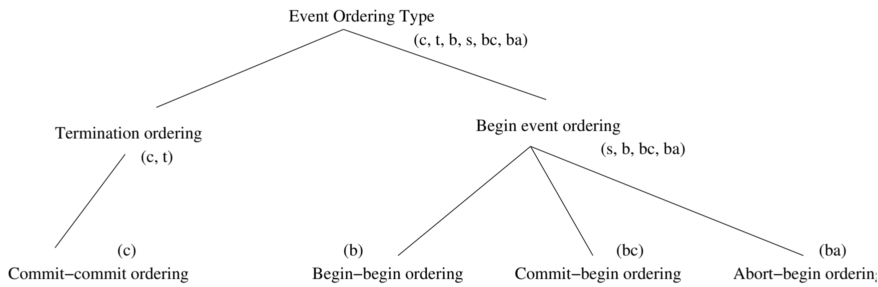

Figure 3: Event ordering type of dependencies

cannot be purchased, the resort room should not be reserved. The workflow has two completion sets which means that it can complete in two ways. The first one is satisfied when the travel reservation is successful and subtransactions T_1, T_2, and T_4 have committed. Moreover, in this case T_1 needs to commit before T_2. The second one occurs when the travel reservation is not successful – in this case, T_1 and T_3 must commit and the commitment of T_1 must precede that of T_3.

The generalized advanced transaction can be specialized by restricting the number and types of dependencies that can exist between its transactions. For instance a SAGA [18] consisting of three subtransactions T_1, T_2, T_3 and compensating subtransactions C_2 and C_1 can be expressed as a generalized advanced transaction T where $S = \{T_1, T_2, T_3, C_1, C_2\}$, $D = \{T_2 \rightarrow_{sc} T_1, T_3 \rightarrow_{sc} T_2, C_2 \rightarrow_{sc} T_2, C_1 \rightarrow_{sc} T_1, C_2 \rightarrow_{sc} C_1\}$, $C = \{(\{T_1, T_2, T_3\}, \{\}), (\{T_1, T_2, C_2, C_1\}, \{T_1 \ll C_1, T_2 \ll C_2\}), (\{T_1, C_1\}, \{T_1 \ll C_1\})\}$. Other advanced transactions can also be expressed in the form of a generalized transaction using different kinds of dependencies.

In this work, however, we do not restrict the dependencies that are possible between transactions of a generalized advanced transaction. Rather our goal is to illustrate which combinations of dependencies cause problems and which do not.

4 Dependency Classification

The above dependencies 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 5 [Event Ordering Dependency] An *event ordering dependency* $T_i \rightarrow_o T_j$ 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 events. Detailed classification of the event ordering dependencies is given in Figure 3.

Definition 6 [Event Enforcement Dependency] An *event enforcement dependency* $T_i \rightarrow_e T_j$ 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, exclusion, 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. Detailed classification of the event ordering dependencies is given in Figure 4.

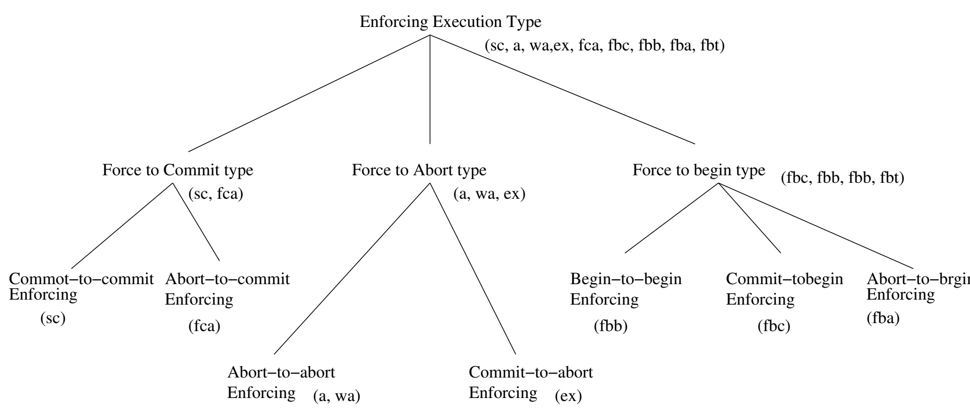

Figure 4: Event enforcement type of dependencies

In the preceding discussion, we have classified the dependencies into event enforcement and event ordering types. One point needs to be mentioned. Sometimes event enforcement dependencies impose an execution order on subtransactions. For instance, consider the strong commit dependency $T_i \rightarrow_{sc} T_j$. This dependency requires T_j to commit if T_i commits. It does not specify any order of execution. So 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 $T_j \rightarrow_a T_i$. Thus, to ensure the satisfaction of the strong commit dependency, an execution order is also imposed.

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.

5 Composite Dependencies

Our model allows us to specify multiple kinds of dependencies between any pair of subtransactions in a generalized advanced transaction. Such dependencies are called composite dependencies. Each individual dependency defined over a pair of subtransactions is referred to as a primitive dependency. Composite dependencies are often needed to express more powerful coordination controls over transactions.

Definition 7 [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 primitive dependencies $d_1, d_2, \ldots, d_n$. The effect of the composite dependency is the conjunction of the constraints imposed by the primitive dependencies $d_1, d_2, \ldots, d_n$.

We define two kinds of of dependency relationships that may exist between a pair of dependencies: *inclusion* and *conflict*.

Definition 8 [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 logically 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.

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. This is formalized by the following theorem.

Theorem 1 The abort dependency $T_i \rightarrow_a T_j$ includes the weak abort dependency $T_i \rightarrow_{wa} T_j$, that is, $(T_i \rightarrow_{wa} T_j) \subseteq (T_i \rightarrow_a T_j)$.

Proof 1 To prove this, we must show that the constraints imposed by the abort dependency $(a_i \Rightarrow a_j)$, implies the constraints imposed by the weak abort dependency $(a_i \Rightarrow (\neg(c_j \prec a_i) \Rightarrow a_j)$. In other words, we need to show that

$$(a_i \Rightarrow a_j) \Rightarrow (a_i \Rightarrow (\neg(c_j \prec a_i) \Rightarrow a_j)) = True$$

The left hand side on simplification evaluates to true which equals the right hand side.

Similarly, we can show that the force-begin-on-begin dependency includes force-begin-on-commit, force-begin-on-abort, and force-begin-on-terminate. Similarly, the serial dependency includes the begin dependency and the termination dependency. A complete list of inclusion relations is given in Table 1.

Definition 9 [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.

Dependency	Includes
$T_i \rightarrow_a T_j$	wa
$T_i \rightarrow_{fbb} T_j$	fbc, fba, fbt
$T_i \rightarrow_{fbt} T_j$	fba, fbc
$T_i \rightarrow_t T_j$	c
$T_i \rightarrow_s T_j$	b, t
$T_i \rightarrow_{bc} T_j$	t, b, s, wbc, c
$T_i \longrightarrow_{ba} T_j$	t, b, s

Table 1: Dependencies Inclusion Relationship

Dependency	Conflicting dependency
$T_i \longrightarrow_{sc} T_j$	ex, bc, s, c, a, t, wbc, 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, wbc
$T_i \longrightarrow_{fbc} T_j$	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
$T_i \longrightarrow_{wbc} T_j$	sc, fca

Table 2: Conflicting dependencies over same subtransaction pair

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. Thus, when T_i commits, it is impossible to satisfy both dependencies. Such conflicts can be found out by using a conjunction on the conditions imposed by the dependencies and evaluating the conjunct to find whether it will be satisfied for all possible executions. A second example of conflict is caused by the dependencies: $T_i \rightarrow_a T_j$ and $T_i \rightarrow_{fca} T_j$. Below, we give a formal proof of why these two dependencies conflict.

Theorem 2 The dependencies $T_i \rightarrow_a T_j$ and $T_i \rightarrow_{fca} T_j$ conflict.

Proof 2 To prove this we must show that the constraints imposed by the abort dependency $(a_i \Rightarrow a_j)$ and the force-commit-on-abort dependency $(a_i \Rightarrow c_j)$ cannot be satisfied for all possible executions. The conjunction of the constraints imposed by the two dependencies evaluate to $\neg a_i$. This constraint is only satisfied when T_i is not aborted. Thus if T_i aborts, the conjunction of the constraints imposed by the dependencies will not be satisfied.

Conflicts for the other dependency pairs listed in Table 2 can be proved in a similar manner.

6 Dependency Minimization

Dependencies specified in a generalized advanced transaction may not be independent. A given set of dependencies may logically imply the existence of other dependencies. For instance, a dependency d_x may include another dependency d_y. In such a case, even if d_y is not in the given set, it is logically implied. A second example is when dependencies have the *transitive property*. 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 this 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 different dependencies. Dependencies specified between different pairs of subtransactions may interact with each other and they may imply other dependencies. For instance, consider the dependencies $T_i \rightarrow_a T_j$ and $T_j \rightarrow_{bc} T_k$. Although there are no explicit dependencies between the subtransactions T_i and T_k, the interaction of the dependencies $T_i \rightarrow_a T_j$ and $T_j \rightarrow_{bc} T_k$ will put some constraints over the execution of T_i and T_k. $T_i \rightarrow_a T_j$ requires that T_j must abort if T_i aborts. $T_j \rightarrow_{bc} T_k$ requires that T_k cannot begin until T_j commits. Combining these two constraints, we obtain that T_k cannot begin until T_i commits, that is, $T_i \rightarrow_{bc} T_k$. Similarly, the dependencies $T_i \rightarrow_{fca} T_j$and $T_j \rightarrow_{fbc} T_k$ imply the existence of $T_i \rightarrow_{fba} T_k$. Also, $T_i \rightarrow_{bc} T_j$ and $T_i \rightarrow_{ex} T_k$ logically imply $T_j \rightarrow_{ex} T_k$. Many more such examples exist. This motivates us to propose the definition of closure of a dependency set. The definitions are in the same vein as defined for functional dependencies [16].

Definition 10 [Closure of a Dependency Set] Let **D** be a set of dependencies. The closure of **D**, denoted by $\mathbf{D}^+$, is the set of dependencies that can be logically derived from the dependencies in **D**.

Definition 11 [Cover] If every dependency that can be logically derived from the set $\mathbf{D_1}$ can also implied be derived from the set $\mathbf{D_2}$, then $\mathbf{D_2}$ is said to be a *cover* for $\mathbf{D_1}$. In other words, $\mathbf{D_2}$ is a cover for $\mathbf{D_1}$, if $\mathbf{D_1}^+ \subseteq \mathbf{D_2}^+$.

Thus, the specification of a generalized advanced transaction may contain extraneous dependencies. The presence of extraneous dependency slows down the processing of the transaction because dependencies need to be checked for ensuring correct behavior. In this section, we define how to minimize the given set of dependencies and obtain an equivalent minimal set.

Definition 12 [Redundant Dependency] A dependency $T_i \rightarrow_x T_j$ is said to be *redundant* in a set of dependencies **D**, if $(T_i \rightarrow_x T_j) \in (\mathbf{D} - \{T_i \rightarrow_x T_j\})^+$.

Informally speaking, a dependency $T_i \rightarrow_x T_j$ is redundant in the set **D** if the constraints imposed by $T_i \rightarrow_x T_j$ can be derived from the constraints of the dependencies in the set $\mathbf{D} - \{T_i \rightarrow_x T_j\}$. A dependency set not having any redundant dependencies is said to be minimal. A minimal dependency set is formally defined below.

Definition 13 [Minimal] A set of dependencies **M** is said to be *minimal* if it satisfies the following condition:

1. for each $(T_i \rightarrow_x T_j) \in M$, the dependency x is a primitive dependency.
2. for each $(T_i \rightarrow_x T_j) \in M$, $(M - \{T_i \rightarrow_x T_j\})^+ \neq M^+$.

Algorithm 1 Finding a Minimal Dependency Set

Input: **D** – Dependency sets that must be minimized

Output: **M** – Minimal dependency set

begin

  $\mathbf{M} = \mathbf{D}$

  for each $(T_i \rightarrow_x T_j) \in \mathbf{M}$

    if $x = x_1, x_2, \ldots, x_n$ is a composite dependency

      $\mathbf{M} = \mathbf{M} - \{T_i \rightarrow_x T_j\}$

      for $m = 1$ to n **do**

        $\mathbf{M} = \mathbf{M} \cup \{T_i \rightarrow_{x_m} T_j\}$

  for each $(T_i \rightarrow_y T_j) \in \mathbf{M}$

    if $(T_i \rightarrow_y T_j) \in (\mathbf{M} - \{T_i \rightarrow_y T_j\})^+$

      $\mathbf{M} = \mathbf{M} - \{T_i \rightarrow_y T_j\}$

end

The first step involves changing each composite dependency $T_i \rightarrow_x T_j$ into a set of primitive dependencies, $T_i \rightarrow_{x_1} T_j$, $T_i \rightarrow_{x_2} T_j$, …, $T_i \rightarrow_{x_n} T_j$. The second step involves checking whether each dependency $T_i \rightarrow_y T_j$ is logically implied by the remaining dependencies in the set $\mathbf{M} - \{T_i \rightarrow_y T_j\}$. If so, $T_i \rightarrow_y T_j$ is redundant and it is taken out of the minimal set **M**. The process is repeated for all the dependencies.

Definition 14 [Equivalence of Dependency Sets] Two dependency sets $\mathbf{D}_1$ and $\mathbf{D}_2$ are equivalent if $\mathbf{D}_1 \subseteq \mathbf{D}_2{}^+$ and $\mathbf{D}_2 \subseteq \mathbf{D}_1{}^+$. In other words, the closure of the two sets are equal, that is, $\mathbf{D}_1^+ = \mathbf{D}_2^+$.

Theorem 3 The minimal set **M** of dependencies obtained by Algorithm 1 is equivalent to the original set **D**.

Proof 3 We need to prove (i) $\mathbf{M} \subseteq \mathbf{D}^+$ and (ii) $\mathbf{D} \subseteq \mathbf{M}^+$. Since $\mathbf{M} \subseteq \mathbf{D}$, (i) is true. Since each dependency in **D** is removed from **M** only if it can be logically implied by the other remaining dependencies in **M**, $\mathbf{D} \subseteq \mathbf{M}^+$.

Algorithm 2 Checking the Equivalence of Dependency Sets

Input: $\mathbf{D_1}$ and $\mathbf{D_2}$ – Dependency sets that are to be checked for equivalence

Output: Boolean – True if they are equivalent, false otherwise

begin

  for each $(T_i \rightarrow_x T_j) \in \mathbf{D_1}$

if $(T_i \rightarrow_x T_j) \notin \mathbf{D_2}^+$

return false

for each $(T_i \rightarrow_x T_j) \in \mathbf{D_2}$

if $(T_i \rightarrow_x T_j) \notin \mathbf{D_1}^+$

return false

return true

end

The algorithm checks whether each dependency in $\mathbf{D}_1$ can be logically implied by the dependencies in $\mathbf{D}_2$. If not, the algorithm returns false. Otherwise, it checks whether each dependency in $\mathbf{D}_2$ can be derived from the dependencies in $\mathbf{D}_1$. If not, it returns false. Otherwise the result is true.

7 Detecting Conflicts

In the previous section, we outlined how to ensure that the composite dependency is conflict-free. Even though each composite dependency is conflict-free, we can still have conflicts involving multiple transactions. Figure 5 gives examples of conflicts involving multiple sub-transactions. Figure 5(a) shows a conflict that occurs when there are the following dependencies: $T_i \rightarrow_{sc} T_j$, $T_j \rightarrow_{sc} T_k$ and $T_i \rightarrow_{bc} T_k$. Since the strong commit dependency has the transitive property, $T_i \rightarrow_{sc} T_j$ and $T_j \rightarrow_{sc} T_k$ implies $T_i \rightarrow_{sc} T_k$. Further, as outlined in Section 4, $T_i \rightarrow_{sc} T_k$ requires the existence of $T_k \rightarrow_c T_i$. This basically means that T_i must commit after T_k. However, the dependency $T_i \rightarrow_{bc} T_k$ requires that T_k cannot begin until T_i commits. In other words, this is an impossible situation – T_k cannot begin before T_i commits and at the same time T_k must commit before T_i. Figure 5(b) shows another example. Before describing the different kinds of conflicts, we define what we mean by conflicts in a set of dependencies.

Definition 15 [Conflict-free Property] A set of dependencies $\mathbf{D} = \{d_1, d_2, \ldots, d_n\}$ is said to have the *conflict-free* property if there are no i dependencies, where $i \leq n$ in the set $\{d_1, d_2, \ldots, d_n\}$ such that these dependencies conflict with each other.

Conflicts can occur between event ordering dependencies – infeasible ordering required by different event ordering dependencies. Conflicts can also occur between event enforcement dependencies – conflicting enforcement requirements placed by event enforcement dependencies.

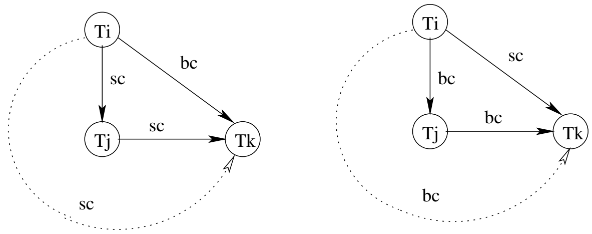

Figure 5: Example of conflict between dependencies of multiple subtransactions

Finally, conflicts can also occur between event ordering dependencies and event enforcement dependencies – this is because some event enforcement dependencies imply an event ordering, and some event ordering dependencies prohibit events.

7.1 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 generalized advanced transaction model and check for the presence of some cycles.

Theorem 4 A generalized advanced transaction has conflicting dependencies if the graph specifying the generalized transaction has a cycle $T_i \rightarrow_{d_a} T_j \rightarrow_{d_b} T_k \ldots T_n \rightarrow_{d_l} T_i$ where each edge is labeled with some event ordering dependency d_x and for any pair d_m, d_n of event ordering dependencies involved either $d_m \subseteq d_n$ or $d_n \subseteq d_m$.

Proof 4 Let the generalized advanced transaction model have a cycle $T_i \rightarrow_{d_a} T_j \rightarrow_{d_b} T_k \ldots T_n \rightarrow_{d_l} T_i$. There can be three cases. The first is when all the dependencies are of begin-event ordering type. Recall that a begin-event ordering dependency $T_i \rightarrow_{d_k} T_j$ requires that T_j can begin only after T_i has completed some event (which can be begin, commit or abort). For the given cycle, because of $T_i \rightarrow_{d_a} T_j$, we have T_j can begin only after T_i has completed some event. Similarly, for the edge $T_j \rightarrow_{d_b} T_k$, we have the restriction that T_k can begin only after T_j has completed some event. Since any event of T_j can occur at the same time or after its begin event, we can conclude that T_k can begin only after T_i has completed some event. Proceeding in this way, T_i can begin only after T_i has completed some event. But T_i cannot complete any event before it has begun. We, therefore, arrive at a contradiction, and such a cycle imposes impossible execution ordering. The second case is the one in which all the event ordering dependencies are of the termination ordering type. The termination ordering dependency $T_i \rightarrow_{d_a} T_j$ require that T_j cannot terminate (abort or commit) before T_i does so. Similarly, the dependency $T_j \rightarrow_{d_b} T_k$ require that T_k cannot terminate before T_j terminates. The dependency $T_n \rightarrow_{d_l} T_i$ require T_n to terminate before T_i. By transitivity, we have that T_i cannot terminate before T_i terminates. This is impossible. The third case is that there are some begin event ordering dependencies and some termination ordering dependencies. Since any pair of dependency must either be the same or related by the inclusion relationship, the only kinds of begin event ordering dependencies that can be present in the cycle are *s*, *wbc*, *bc*, and *ba*. Suppose one or more edges in the cycle $T_i \rightarrow_{d_a} T_j$ have such begin event ordering dependencies. These dependencies imply that T_j cannot begin until T_i terminates. Since begin of T_j precedes termination of T_j, these dependencies imply that T_j cannot terminate before T_i. In other words, we can assume the existence of a termination dependency between T_i and T_j. Thus, for each such begin event ordering dependency, we add a termination dependency. Now all the edges in the cycle have a termination dependency. Using reasoning similar to the second case, we can prove that existence of a cycle implies impossible satisfaction of the dependencies.

7.2 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. The following theorem formalizes the conditions under which event enforcement dependencies cause conflict.

Theorem 5 A generalized advanced transaction has conflicting dependencies if all the following conditions are satisfied:

1. any node T_k in the graph specifying the generalized transaction has at least two incident edges labeled with event enforcement dependencies, say, $T_i \rightarrow_{d_x} T_k$ and $T_j \rightarrow_{d_y} T_k$;
2. the dependency $T_i \rightarrow_{d_x} T_k$ imposes the constraint that the occurrence of event e_i necessitates the occurrence of event e_k;
3. the dependency $T_j \rightarrow_{d_y} T_k$ puts the constraint that the occurrence of event e_j requires the occurrence of event e'_k;
4. e_k and e'_k can never occur in the same instance of the advanced transaction; and

5. e_i and e_j can occur in the same instance of the advanced transaction.

Proof 5 Consider the following scenario. Suppose the events e_i and e_j have both occurred. The dependency $T_i \rightarrow_{d_x} T_k$ requires event e_k to occur. The dependency $T_j \rightarrow_{d_y} T_k$ requires event e'_k to occur. Since e_k and e'_k cannot both occur in the same instance of the advanced transaction, it is impossible to satisfy both the dependencies and there is a conflict.

Theorem 6 The complexity of checking whether two event enforcement dependencies $T_i \rightarrow_{d_x} T_k$ and $T_j \rightarrow_{d_y} T_k$ cause a conflict or not is in the worst case $O(2^n)$ where n is the number of subtransactions that lie in the path from T_i to T_j.

Proof 6 To prove that the event enforcement dependencies do not cause a conflict, we need to show that e_i and e_j cannot both occur. This is only possible if there is a path from e_i to e_j or vice-versa and the dependencies existing on this path impose constraints that ensure $e_i \Rightarrow \neg e_j$. To prove that the two dependencies do not cause conflict, we need to derive that $e_i \Rightarrow \neg e_j$ from the constraints implied by the dependencies of subtransactions connecting T_i and T_j. This derivation incurs an overhead of $O(2^n)$ where n is the number of subtransactions connecting T_i and T_j.

Instead of proving $e_i \Rightarrow \neg e_j$, we can detect the presence of certain dependencies between T_i and T_j. But this involves understanding what kinds of conflicts can occur between event enforcement dependencies. In the rest of this section, we discuss all the possible conflicts that can occur because of the presence of multiple event enforcement dependencies. Consider the node T_k that has multiple incoming edges – let two of these edges correspond to the dependencies $T_i \rightarrow_{sc} T_k$ and $T_j \rightarrow_{ex} T_k$. 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 edges. If not, then T_i and T_j can commit independently. Otherwise, we need to check whether the constraints imposed by the dependencies that exist between the T_i and T_j imply $c_i \Rightarrow \neg c_j$. 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 none of these dependencies exist 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 both abort. 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 together. 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 both 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.

7.3 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 Section 4). 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 7.1, we check for the presence of cycles to detect such conflicts.

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.

Theorem 7 A generalized advanced transaction has conflicting dependencies if all the following conditions are satisfied:

1. any node T_k in the graph specifying the generalized transaction has an incident edge labeled with an event ordering dependency, say, $T_i \rightarrow_{d_x} T_k$ and another incident edge

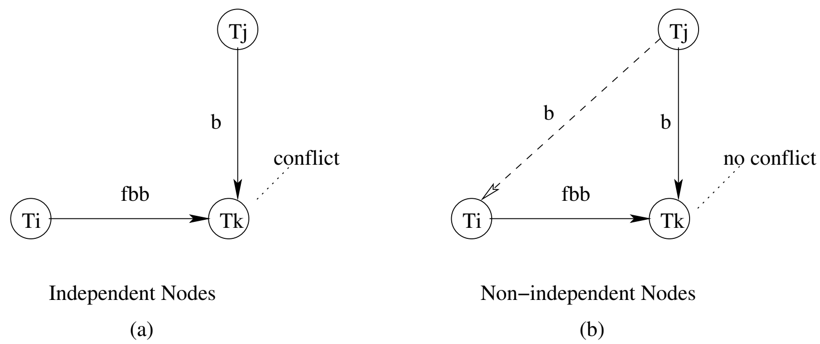

Figure 6: Conflicts with a force-begin-on-begin dependency and a begin dependency

labeled with an event enforcement dependency $T_j \rightarrow_{d_y} T_k$;

2. the event ordering dependency $T_i \rightarrow_{d_x} T_k$ imposes the constraint that the absence of event e_i prohibits the occurrence of event e_k;

3. the event enforcement dependency $T_j \rightarrow_{d_y} T_k$ imposes the constraint that the occurrence of event e_j requires the occurrence of event e_k;

4. the absence of event e_i and the presence of event e_j is possible in the same execution.

Proof 7 Since occurrence of event e_j does not guarantee the occurrence of e_i, it is possible that only e_j will occur and not e_i. Since e_j has occurred, by virtue of the dependency $T_j \rightarrow_{d_y} T_k$, e_k is required to occur. Moreover, since e_i has not occurred, e_k cannot occur because of the dependency $T_i \rightarrow_{d_x} T_k$. Thus, the two dependencies impose conflicting requirements on event e_k.

For instance, the event ordering dependency $T_j \rightarrow_b T_k$ does not allow T_k to begin unless T_j has begun. Suppose there is an event enforcement dependency $T_i \rightarrow_{fbb} T_k$. This is shown in Figure 6. Now if T_i begins and T_j does not begin, we have a problem: T_k is required to begin and not begin. 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 $\neg b_j$ 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.

Next, 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 occurrence of both b_i and $\neg a_j$. 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.

Similarly, the dependencies $T_i \rightarrow_{fba} T_k$ and $T_j \rightarrow_{b} T_k$ result in a conflict unless either $T_i \rightarrow_{fba} T_j$ or $T_j \rightarrow_{b} T_i$ exist between T_i and T_j. The dependencies $T_i \rightarrow_{fba} T_k$ and $T_j \rightarrow_{ba} T_k$ are problematic unless either $T_i \rightarrow_{a} T_j$ or $T_j \rightarrow_{ba} T_i$ exist between T_i and T_j. The dependencies $T_i \rightarrow_{fba} T_k$ and $T_j \rightarrow_{bc} T_k$ are problematic unless either $T_i \rightarrow_{fca} T_j$ or $T_j \rightarrow_{bc} T_i$ exist between T_i and T_j.

Similar problems are created by other dependencies as well. $T_i \rightarrow_{fbc} T_k$ and $T_j \rightarrow_{b} T_k$ result in a conflict unless either $T_i \rightarrow_{fbc} T_j$ or $T_j \rightarrow_{b} T_i$ exist between T_i and T_j. $T_i \rightarrow_{fbc} T_k$ and $T_j \rightarrow_{ba} T_k$ result in a conflict unless $T_j \rightarrow_{ba} T_i$ exists. Similarly, $T_i \rightarrow_{fbc} T_k$ and $T_j \rightarrow_{bc} T_k$ result in a conflict unless either $T_i \rightarrow_{sc} T_j$ or $T_j \rightarrow_{bc} T_i$ exist between T_i and T_j. $T_i \rightarrow_{fbt} T_k$ and $T_j \rightarrow_{ba} T_k$ result in a conflict unless $T_j \rightarrow_{ba} T_i$ exists. $T_i \rightarrow_{fbt} T_k$ and $T_j \rightarrow_{bc} T_k$ result in a conflict unless $T_j \rightarrow_{bc} T_i$ exists. $T_i \rightarrow_{fbt} T_k$ and $T_j \rightarrow_{b} T_k$ result in a conflict unless either $T_j \rightarrow_{b} T_i$ or $T_i \rightarrow_{fbt} T_j$ exists.

Sometimes event ordering dependencies, such as, *ba* and *bc* prohibit the occurrence of events. $T_i \rightarrow_{ba} T_k$ and $T_j \rightarrow_{ba} T_k$ result in a problem unless there exists a dependency of the form $T_i \rightarrow_{a} T_j$. Similar situations are created by the dependencies $T_i \rightarrow_{bc} T_k$ and $T_j \rightarrow_{bc} T_k$. There is a problem unless there exists a dependency of the form $T_i \rightarrow_{sc} T_j$. Also, $T_i \rightarrow_{ba} T_k$ and $T_j \rightarrow_{bc} T_k$ conflict unless there exists a dependency of the form $T_i \rightarrow_{ex} T_j$ or $T_i \rightarrow_{ba} T_j$.

Once such relationships between dependencies are identified and tabulated, conflict detection is simplified. Conflict detection involves representing the advanced transaction in the form of a graph, checking for certain cycles in the graph, looking for nodes having multiple incident edges corresponding to dependencies and checking whether the source nodes of these edges are related by certain dependencies.

8 Impact of Dependencies on Completion Sets

Recall that a generalized advanced transaction $T = < S, D, C >$ specifies the set of subtransactions S, the dependencies between these subtransactions D, and the set C that contains the completion sets of T. A completion set of a generalized advanced transaction specifies the subtransactions that need to be committed and the order in which they must be committed for successful execution of some instance of the generalized advanced transaction. Since different subtransactions may be committed in different instances of an advanced transaction, an advanced transaction may have multiple completion sets. The completion sets specified by a user in a generalized advanced transaction may not conform to the given dependencies. We provide an algorithm that checks whether the completion set complies with the dependencies in an advanced transaction.

Algorithm 3 Check whether Completion Set Conforms to Dependency

Input: Specification of generalized advanced transaction $T = < S, D, C >$

Output: Returns true if completion set conforms to the dependencies, false otherwise

begin

  for each $C_m \in \mathbf{C}$ where $C_m = (CT_m, \ll_m)$

    if $(T_i \rightarrow_{sc} T_j \in \mathbf{D}) \wedge (T_i \in CT_m) \wedge (T_j \notin CT_m)$

      return false

    if $(T_i \rightarrow_{bc} T_j \in \mathbf{D}) \wedge (T_i \notin CT_m) \wedge (T_j \in CT_m)$

      return false

    if $(T_i \rightarrow_{ex|ba} T_j \in \mathbf{D}) \wedge (T_i \in CT_m) \wedge (T_j \in CT_m)$

      return false

    if $(T_i \rightarrow_{c|t|s|bc} T_j \in \mathbf{D}) \wedge (T_i \in CT_m) \wedge (T_j \in CT_m) \wedge (T_i \not\ll_m T_j)$

      return false

  return true

end

The algorithm looks at each completion set to check whether it complies with the dependencies. Not all dependencies impact a completion set. The algorithm begins by checking whether the completion set complies with the strong commit dependency. The strong commit dependency $T_i \rightarrow_{sc} T_j$ requires that if T_i commits, then T_j must commit. Hence, if the completion set contains T_i and not T_j, then the algorithm reports false signifying that the completion set does not conform to the dependency. Next, we consider the begin-on-commit dependency $T_i \rightarrow_{bc} T_j$. In this case, the algorithm reports false if T_i is not in the completion set but T_j is present. The algorithm then checks for exclusion or begin-on-abort dependency between T_i and T_j. In such cases, if the completion set contains both T_i and T_j, then we have a problem because the completion set violates the dependency. Finally, if there is a commit, termination, serial, and begin-on-commit dependency between T_i and T_j, and the commit order required by these dependencies does not comply with that specified in the completion set, the algorithm returns false. Otherwise the algorithm returns true indicating that the completion set conforms with the dependencies.

9 Well-Structured Generalized Advanced Transaction

Definition 16 [Well-Structured Generalized Advanced Transaction] A generalized advanced transaction $T =< S, D, C >$ is well-structured if the following conditions hold:

1. dependencies in D satisfy the conflict-free property;
2. dependencies in D satisfy the minimality property; and
3. completion sets C conforms to the dependencies specified in D.

Different instances can be generated from a given generalized advanced transaction. These instances differ with respect to the subtransactions executed and completed.

Definition 17 [Correct Execution] An execution of a generalized advanced transaction $T =< S, D, C >$ is *correct* if the subtransactions executed satisfy the dependencies listed in **D**.

Definition 18 [Complete Execution] An execution of a generalized advanced transaction $T =< S, D, C >$ is *complete* if all subtransactions in S are either in unscheduled, aborted or committed states.

A generalized advanced transaction can complete successfully or unsuccessfully. Below we define what it means for a generalized advanced transaction to complete unsuccessfully. Stated informally, an execution of a generalized advanced transaction $T =< S, D, C >$ is unsuccessful if all the subtransactions in S are either in unscheduled or aborted states.

Definition 19 [Unsuccessful Complete Execution] The execution of a generalized transaction $T = < S, D, C >$ is said to be *unsuccessful* if the following condition holds: $\forall T_i \in S \bullet (un_i = true \vee ab_i = true)$.

Next we define what it means for a generalized advanced transaction to be successfully completed. A generalized advanced transaction is said to be successfully completed if all the subtransactions specified in one of the completion sets have committed and all the other subtransactions specified for the generalized advanced transaction are either aborted or unscheduled.

Definition 20 [Successful Complete Execution] A generalized advanced transaction of type $T = < S, D, C >$ is said to be successfully completed if the following condition holds: $\exists C_m = (CT_m, \ll_m) \in C \bullet ((\forall T_i \in CT_m \bullet cm_i = true) \wedge (\forall T_j \in S - CT_m \bullet ab_j = true \vee un_j = true))$.

The definition of well-structured generalized advanced transactions and a notion of their correct execution will help us develop a transaction processing system that is capable of processing various kinds of advanced transactions.

10 Conclusion

In this work, we have shown how dependencies present in advanced transaction models can be analyzed to ensure correct execution. In this paper, we focused on two important properties of dependencies: conflict-free and minimality. Conflict-free property ensures that the dependencies are physically realizable. It also ensures that the dependencies are free from deadlocks. Minimality ensures that no redundant dependencies are specified. This helps to eliminate the processing of unnecessary dependencies. Once the interactions of dependencies are well-understood, new kinds of transaction processing models can be synthesized.

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 transaction scheduler must take into account the nature of dependencies before scheduling the operations of a transaction. The recovery manager of the traditional transaction 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.

Database attacks will occur in spite of sophisticated preventive techniques. A generalized advanced transaction or one of its component transactions may be malicious. One future work is on focusing how to survive such attacks and how to repair the damages caused by malicious transactions. The presence of different dependencies in an advanced transaction helps to spread the attacks caused by malicious transactions. One future work is how to identify and repair the damage caused by such malicious transactions.

References

[1] N. R. Adam, V. Atluri, and W. K. Huang. Modeling and Analysis of Workflows using Petri Nets. *Journal of Intelligent Information Systems*, 10(2):131–158, March 1998.

[2] G. Alonso, H. Kuno, F. Casati, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer, 2003.

[3] 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.

[4] 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.

[5] Paul C. Attie, Munindar P. Singh, Amit P. Sheth, and Marek Rusinkiewicz. Specifying and Enforcing Intertask Dependencies. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 134–145, Dublin, Ireland, August 1993. Morgan Kaufmann.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[7] E. Bertino, G. Chiola, and L. V. Mancini. Deadlock Detection in the Face of Transaction and Data Dependencies. In *Proceedings of the 19th International Conference on Application and Theory of Petri Nets*, volume 1420 of *Lecture Notes in Computer Science*, pages 266–285, Lisbon, Portugal, June 1998. Springer-Verlag.

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

[9] A.J. Bonner. Workflows, Transactions and Datalog. In *Proceedings of the 18th SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Philadelphia, Pennsylvania, May 1999.

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

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

[12] Panos Chrysanthis and Krithi 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.

[13] H. Davulcu, M. Kifer, C.R. Ramakrishnan, and I.V. Ramakrishnan. Logic based Modeling and Analysis of Workflows. In *Proceedings of the 17th SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Seattle, Washington, June 1998.

[14] U. Dayal, M. Hsu, and R.Ladin. Organizing Long-Running Activities with Triggers and Transactions. In *Proceeding of the 17th International Conference on Very Large DataBases*, September 1991.

[15] A. K. Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

[16] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems Systems*. Addison-Wesley, Reading, MA, 4th edition, 2003.

[17] X. Fu, T. Bultan, R. Hull, and J. Su. Verification of Vortex Workflows. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Genova, Italy, April 2001.

[18] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 249–259, San Francisco, CA, 1987.

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

[20] R. Gunthor. Extended Transaction Processsing based on Dependency Rules. In *Proceedings of the 3rd International Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems*, pages 207–214, Vienna, Austria, April 1993.

[21] George Theodore Heineman. A Transaction Manager Component Supporting Extended Transaction Models. PhD Thesis, Columbia University, 1996.

[22] J. Klein. Advanced Rule Driven Transaction Management. In *CompCon*, pages 568–573. IEEE, February 1991.

[23] L. V. Mancini, I. Ray, S. Jajodia, and E. Bertino. Flexible Transaction Dependencies in Database Systems. *Distributed and Parallel Databases*, 8:399–446, 2000.

[24] J. E. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. PhD Thesis 260, MIT, Cambridge, MA, April 1981.

[25] S. Mukherjee, H. Davulcu, M. Kifer, P. Senkul, and G. Yang. Logic-Based Approaches to Workflow Modeling and Verification. In J. Chomicki and R. v. d. Meyden and G. Saake, editor, *Logics for Emerging Applications of Databases*. Springer, 2004.

[26] M. P. Papazoglou. Web Services and Business Transactions. *World Wide Web Journal*, 6(1):49–91, March 2003.

[27] M. Prochazka. Extending Transactions in Enterprise JavaBeans. Tech. Report No. 2000/3, Dep. of SW Engineering, Charles University, Prague, January 2000.

[28] Muller Robert. Event-Oriented Dynamic Adaption of Workflows: Model, Architecture and Implementation. PhD Disseration, University of Leipzig, January 2003.

[29] Marek Rusinkiewicz and Amit P. Sheth. Specification and Execution of Transactional Workflows. In *Modern Database Systems 1995*, pages 592–620, 1995.

[30] Munindar 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.

[31] Jian Tang and J. Veijalainen. Enforcing Intertask Dependencies in Transactional Workflows. Research Report No. J-2/95. VTT Information Technology, Finland, Jan. 1995.

[32] W. M. P. van der Aalst. The Application of Petri-Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 1(8), 1998.

[33] Helmut Wächter and Andreas Reuter. The ConTract Model. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 219–263. Morgan Kaufmann, 1992.

[34] T. Xin and I. Ray. Conflict Detection of Control Flow Dependencies in Advanced Transaction Models. In *9th International Database Engineering and Applications Symposium*, Montreal, Canada, July 2005.