Execution Dependencies in Transaction Closures

Kerstin Schwarz Can Türker Gunter Saake

Otto-von-Guericke-Universität Magdeburg
Institut für Technische und Betriebliche Informationssysteme
Postfach 4120, D–39016 Magdeburg, Germany
E-mail: {schwarz|tuerker|saake}@iti.cs.uni-magdeburg.de
Phone: ++49/391/67-{18066|12994|18800} Fax: ++49/391/67-12020

Abstract

Activities of advanced applications can be modeled by interrelated transactions. These relations can be described by different kinds of transaction dependencies. The notion of transaction closure is a generalization of nested transactions providing means to describe complex activities such as transactional workflows. In this paper, our main focus lies on execution dependencies for describing certain control flows among related transactions of transaction closures. In particular, we consider the transitivity property for all kinds of transaction execution dependencies and discuss their relationship to other kinds of dependencies such as transaction termination dependencies. We point out that some of these dependency combinations are incompatible. As a result we present rules for reasoning about the transitivity of execution dependencies. Thus, we are able to conclude how arbitrary transactions of a transaction closure are transitively interrelated.

Keywords: transaction closure, execution dependencies, transitive dependencies, dependency combinations.

1 Introduction

Modern information systems require advanced transaction models providing means to describe complex activities such as transactional workflows. Complex activities consist of sets of transactions which are interrelated, i.e., there are dependencies among several transactions. As a suitable framework for the design of complex applications, we introduced the concept of *transaction closure* [STS98a] as a generalization of the well-known concept of nested transactions [Mos85] together with a set of transaction dependencies. A transaction closure comprises a set of transactions which are (transitively) initiated by the same (root) transaction. In contrast to classical nested transactions, a child transaction in a transaction closure may survive the termination of its parent transaction — a case which is needed for example in models for long-during activities [DHL91, RKT+95], workflows [GHS95, KR96], or transactions in active databases [HLM88, BÖH+92].

Execution dependencies play a central role in transaction closures. An execution dependency is a constraint on the temporal occurrence of the start and termination events of related transactions. The set of execution dependencies determine the valid control flows among related transactions of a transaction closure. Whereas the execution dependencies for direct child transactions have to be explicitly specified, the dependencies of transitively related transactions can be computed based on this direct dependencies. Execution dependencies are especially required for defining the precise relationship between the triggering and triggered transaction in active database systems [DHW95]. Considerations where transactions has to meet certain constraints with regard to their invocation and completion times (known from the area of real-time transactions [SKS96]) are not subject of this paper.

As discussed in detail in [STS98b], *termination dependencies* are another issue of transactions closures. A termination dependency is a constraint on the possible combinations (and orders) of the termination events (commit/abort) of two related transactions. Execution and termination dependencies constrain the set of valid execution orders for transaction pairs. Execution dependencies may influence termination dependencies such that only the abortion of one or more transactions is valid. These dependency combinations are denoted as incompatible.

As an extension of the work we presented in [STS98a, STS98b], we present in this paper a framework for handling execution dependencies for transitive ancestor relations in transaction closures. The result is a practical algorithm for computing derived execution dependencies for transitively related transactions. Furthermore, we combine execution and termination dependencies and investigate incompatible dependency combinations. The rules obtained by the algorithm and the rules which identify invalid dependency combinations enable us to reason about the transitive relationship between transactions and to detect invalid parts of the specification during the design time. The concept of transaction closure together with the dependencies and rules provide the basis for a transaction design and analyzing tool. Such a tool can help to understand the entire semantics of a complex application and thus it may support the design of better and more efficient applications.

The paper is organized as follows: In Section 2, we introduce the basic notions including the concept of transaction closure. Thereafter, in Section 3, we discuss transaction execution dependencies which deal with the valid ordering of the start and end event of related transactions of a transaction closure. In Section 4, we consider the transitivity property of the execution dependencies introduced and develop an algorithm for deriving all valid transitive dependencies. Afterwards, in Section 5, execution dependencies in combination with termination dependencies are considered. The application of transaction closures, especially the derivation of transitive dependencies in transaction closures, is shown by an example in Section 6. Finally, the paper is concluded by an outlook on future work.

2 Foundations

In this section, we declare the basic concepts and notions which are used throughout this paper. Here, we use the basic notions of the ACTA formalism [CR91, CR94].

Traditionally, a *transaction* is an execution unit consisting of a set of database operations. A transaction t_i is started by invoking the transaction management primitive *begin* (b_{t_i}) and is terminated by the primitive *end* (e_{t_i}). In detail, the termination primitive of a transaction t_i is either *commit* (c_{t_i}) or *abort* (a_{t_i}). These primitives are termed as *significant events*. Furthermore, a transaction invokes operations, termed as *object events*, to access and manipulate the state of database objects. A *history* [BHG87] of a concurrent execution of a set of transactions T comprises all events associated with the transactions in T and indicates the (partial) order in which these events occur. The complete history which contains only terminated transactions is denoted as H, the current (incomplete) history is termed as H_{ct}.

A single arrow ($\rightarrow$) between significant events of transactions which appears in H denotes temporal sequence. For instance, the begin of transaction t_i precede the begin of transaction t_j is expressed by ($b_{t_i} \rightarrow b_{t_j}$). We assume that two events cannot occur at the same time. Constraints on the significant events involving the begin and end of two related transactions are called *execution dependencies*. The following fundamental axiom has to be fulfilled by each transaction. The begin event of a transaction always precedes the end event of the same transaction:

$$(b_{t_i} \rightarrow e_{t_i}) \tag{0}$$

A set of transactions with dependencies among them can be considered as transaction closure [STS98a]. Transaction closures are a generalization of the well-known concept of nested transactions [Mos85]. For the definition of the notion of a transaction closure we first define some basic notions.

Definition 2.1 *The following self-explanatory functions and predicates describe general relationships* *between a transaction and its initiator:*

$$\begin{aligned} parent(t_i, t_j) &:= (\, t_i \text{ is parent of } t_j \,) \\ root(t_j) &:= (\, t_j \text{ has no parent} \,) \\ ancestor(t_i, t_j) &:= (\, parent(t_i, t_j) \vee (\exists t_k : ancestor(t_i, t_k) \wedge parent(t_k, t_j)) \,) \end{aligned}$$

Definition 2.2 (Transaction Closure) *Suppose tc denotes the set of transactions of a transaction closure and let t_i and t_j be two transactions of this closure:*

- *Each transaction closure has* exactly one[1] *root transaction:*

$$\exists!\, t_i \in tc : root(t_i)$$

- *Each non-root transaction has* exactly one *parent transaction:*

$$\forall t_j \in tc : \neg root(t_j) \Rightarrow (\exists!\, t_i \in tc : parent(t_i, t_j))$$

- *Each transaction closure is acyclic:*

$$\nexists t_i \in tc : ancestor(t_i, t_i)$$

- *The initiation of a transaction must follow the initiation of the parent:*

$$\forall t_j \in tc : \neg root(t_j) \Rightarrow (\exists t_i \in tc : parent(t_i, t_j) \wedge (b_{t_i} \rightarrow b_{t_j}))$$

The effects of transactions on other transactions are described by dependencies which are constraints on possible histories.

3 Execution Dependencies

In this section, we investigate *execution dependencies* between transactions which can be expressed in terms of significant events associated with the corresponding transactions. The begin and end events are the significant events which are relevant for execution dependencies. Execution dependencies restricts the temporal occurrence of the significant events of the related transactions in H. Considering two transactions t_i and t_j there are four cases in which the significant events of transaction t_i occur before the significant events of transaction t_j. We identify the following basic ordering terms:

$$(b_{t_i} \rightarrow b_{t_j}) \tag{1}$$
$$(b_{t_i} \rightarrow e_{t_j}) \tag{2}$$
$$(e_{t_i} \rightarrow e_{t_j}) \tag{3}$$
$$(e_{t_i} \rightarrow b_{t_j}) \tag{4}$$

The negation of such a term means that the arrow is changed into the opposite direction. For example, term (2) express that the begin of transaction t_i has to precede the termination of transaction t_j. The negation results in a term $(\overline{2})$ where the termination of transaction t_j precedes the begin of transaction t_i: $(e_{t_j} \rightarrow b_{t_i})$.

Our investigations in combining these basic terms lead to the three execution dependencies: *parallel strict overlapping* $(lap(t_i, t_j))$, *parallel including* $(inc(t_i, t_j))$, and *sequential* $(seq(t_i, t_j))$. The execution dependency *parallel* is a generalisation of the parallel strict overlapping and parallel including dependencies. These execution dependencies are defined over the basic ordering terms.

[1] The symbol ∃! stands for "it exists exactly one".

Definition 3.1 (Parallel Strict Overlapping) *Two different transactions t_i and t_j are executed* parallel strict overlapping *if and only if the begin of t_i precedes the begin of t_j, the begin of t_j precedes the termination of t_i, and the termination of t_i precedes the termination of t_j:*

$$lap(t_i, t_j) \quad :\Leftrightarrow \quad (b_{t_i} \rightarrow b_{t_j}) \wedge (b_{t_j} \rightarrow e_{t_i}) \wedge (e_{t_i} \rightarrow e_{t_j})$$

The execution dependency $lap(t_i, t_j)$ is illustrated in Figure 1. Here, we explicitly depict the relationships between the significant events of the transactions t_i and t_j. Each relationship represents an ordering term. The numbers of the related ordering terms is illustrated next to the corresponding term. In case of parallel strict overlapping transactions t_i and t_j, transaction t_i begins before transaction t_j and terminates before t_j's termination. Additionally, the begin of transaction t_j precedes the termination of transaction t_i.

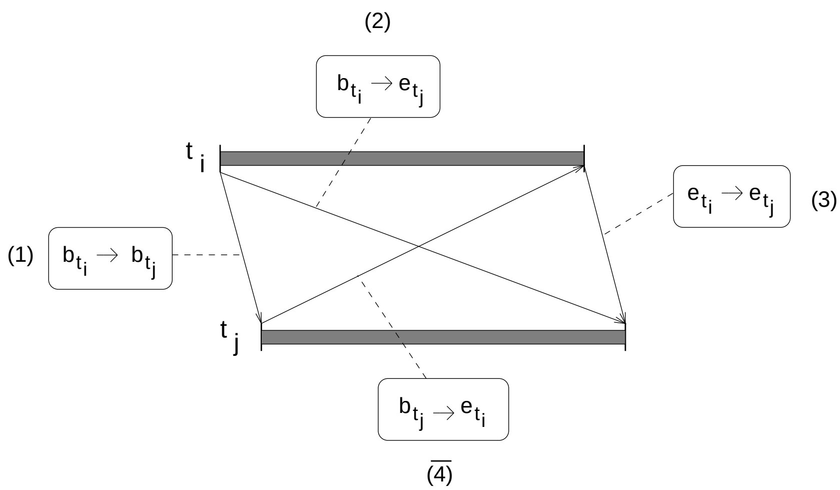

Figure 1: Ordering of Significant Events in Case of Parallel Strict Overlapping: $lap(t_i, t_j)$

Definition 3.2 (Parallel Including) *Two different transactions t_i and t_j are executed* parallel including *if and only if the begin of t_i precedes the begin of t_j but the termination of t_j precedes the termination of t_i:*

$$inc(t_i, t_j) \quad :\Leftrightarrow \quad (b_{t_i} \rightarrow b_{t_j}) \wedge (e_{t_j} \rightarrow e_{t_i})$$

The relation between the significant events of the transactions t_i and t_j which are parallel including are presented in Figure 2. A parallel including dependency between the transactions t_i and t_j means that the begin of transaction t_i precedes the begin of t_j whereas the termination of transaction t_i follows the termination of transaction t_j.

Moreover, we define a general form of the parallel strict overlapping and parallel including dependencies. Such a dependency is useful in case the termination order of the related transactions is not important.

Definition 3.3 (Parallel) *Two different transactions t_i and t_j are executed* parallel *if and only if the begin of t_i precedes the begin of t_j and the begin of t_j precedes the termination of t_i. In other words, two transactions t_i and t_j are executed parallel if and only if they are executed parallel strict overlapping or parallel including:*

$$par(t_i, t_j) \quad :\Leftrightarrow \quad lap(t_i, t_j) \vee inc(t_i, t_j)$$

Finally, transactions may be executed sequentially. This leads to the sequential execution dependency defined in the following:

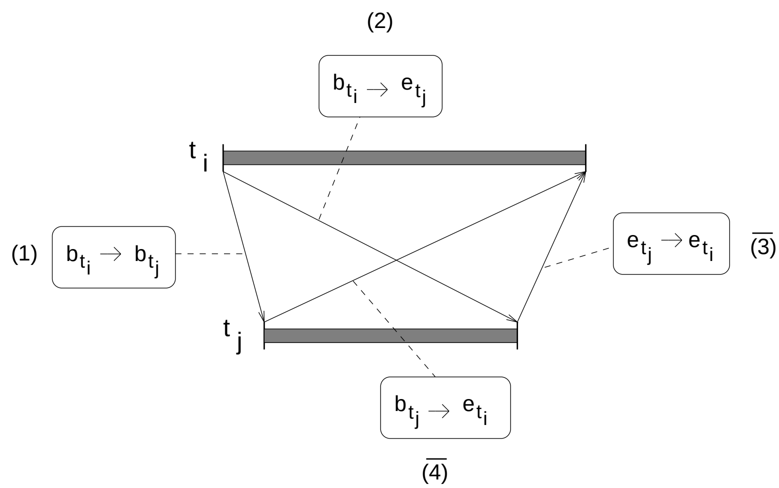

Figure 2: Ordering of Significant Events in Case of Parallel Including: $inc(t_i, t_j)$

Definition 3.4 (Sequential) *A transaction t_j is executed* sequentially *after another transaction t_i if and only if the termination of t_i precedes the begin of t_j:*

$$seq(t_i, t_j) \quad :\Leftrightarrow \quad (e_{t_i} \to b_{t_j})$$

The sequential execution dependency is illustrated in Figure 3. Here, transaction t_i is completely executed before transaction t_j starts executing. Therefore, this execution can be defined only by the ordering relation between the end event of transaction t_i and the start event of transaction t_j. The other relation between the significant events of the corresponding transactions can be derived from this ordering term. This observation is discussed in detail in the following section.

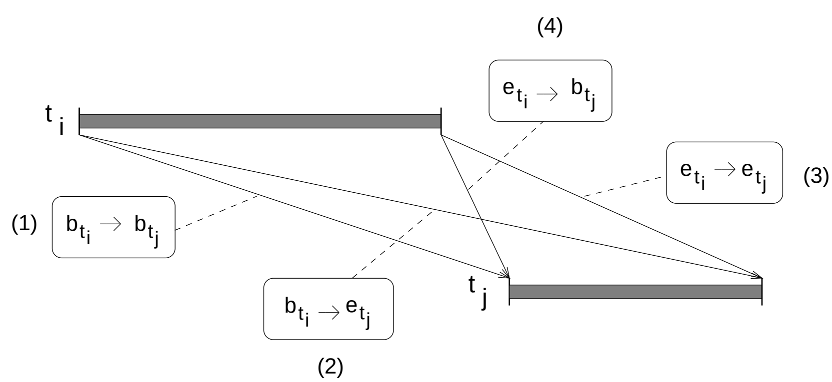

Figure 3: Ordering of Significant Events in Case of Sequential: $seq(t_i, t_j)$

4 Deriving Transitive Execution Dependencies

In the previous section, we defined the execution dependencies parallel strict overlapping, parallel including, parallel, and sequential between two transactions. In Figure 4 we illustrate the scenario in which more than two transactions are involved. Suppose, the transactions t_i and t_k are related by an execution dependency X and the transactions t_k and t_j by an execution dependency Y. We are interested in the transitive dependency Z between the transaction t_i and t_j. Especially, we want to know in which way the relation between the transactions t_i and t_j is constrained by the dependencies X and Y.

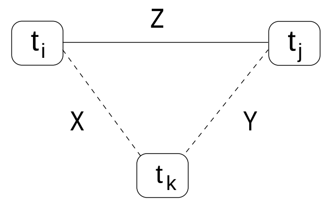

Figure 4: Example of a Transitive Dependency

Deriving the dependency Z between the transactions t_i and t_j from the given dependencies X between the transactions t_i and t_k and Y between the transactions t_k and t_j leads to the following considerations. As presented in Section 3, the dependencies X and Y consist of a set of ordering terms. Furthermore, each begin event of a transaction precedes the termination event of the same transaction (see formula (0)). Combining the basic ordering terms (1)–(4) with formula (0) we are able to derive further ordering terms which are valid in addition to the basic terms:

$$(1): \; (b_{t_i} \rightarrow b_{t_j}) \;\Rightarrow\; (b_{t_i} \rightarrow e_{t_j}) \tag{5}$$
$$(2): \; (b_{t_i} \rightarrow e_{t_j}) \;\Rightarrow\; \mathit{true} \tag{6}$$
$$(3): \; (e_{t_i} \rightarrow e_{t_j}) \;\Rightarrow\; (b_{t_i} \rightarrow e_{t_j}) \tag{7}$$
$$(4): \; (e_{t_i} \rightarrow b_{t_j}) \;\Rightarrow\; (b_{t_i} \rightarrow e_{t_j}) \wedge (b_{t_i} \rightarrow b_{t_j}) \wedge (e_{t_i} \rightarrow e_{t_j}) \tag{8}$$

From this follows that all basic terms including the term (2) itself imply that the begin of transaction t_i has to precede the termination of transaction t_j. Term (4) which build the basis for the sequential execution dependency implies all other basic terms (see (8)). Analogously, the inverse terms $(\overline{1})$–$(\overline{4})$ can be extended:

$$(\overline{1}): \; (b_{t_j} \rightarrow b_{t_i}) \;\Rightarrow\; (b_{t_j} \rightarrow e_{t_i}) \tag{9}$$
$$(\overline{2}): \; (e_{t_j} \rightarrow b_{t_i}) \;\Rightarrow\; (b_{t_j} \rightarrow e_{t_i}) \wedge (b_{t_j} \rightarrow b_{t_i}) \wedge (e_{t_j} \rightarrow e_{t_i}) \tag{10}$$
$$(\overline{3}): \; (e_{t_j} \rightarrow e_{t_i}) \;\Rightarrow\; (b_{t_j} \rightarrow e_{t_i}) \tag{11}$$
$$(\overline{4}): \; (b_{t_j} \rightarrow e_{t_i}) \;\Rightarrow\; \mathit{true} \tag{12}$$

A simple method to derive the transitive execution dependency Z between the transactions t_i and t_j over another transaction t_k and the dependencies X and Y is a *graph based approach.* The *vertices* of the graph are the significant events of the related transactions t_i, t_k, and t_j, e.g. b_{t_j} and e_{t_j}. The *directed edges* are the ordering terms defined by the dependencies X and Y, the derived relations summarized in the formulas (5)–(12), and the relation between the start and end event of the related transactions.

For example, assuming the dependency X is $lap(t_i, t_k)$ and dependency Y is $inc(t_j, t_k)$. The dependency $lap(t_i, t_k)$ is specified by the ordering terms: $(b_{t_i} \rightarrow b_{t_k})$, $(b_{t_k} \rightarrow e_{t_i})$, and $(e_{t_i} \rightarrow e_{t_k})$, whereas the dependency $inc(t_k, t_j)$ is defined by: $(b_{t_j} \rightarrow b_{t_k})$ and $(e_{t_k} \rightarrow e_{t_j})$. These relations and the relation between the begin to the end event of the transactions are illustrated by an arrow in Figure 5. Due to formulas (5)–(12), the graph is extended by the derived terms: $(b_{t_i} \rightarrow e_{t_k})$, $(b_{t_k} \rightarrow e_{t_j})$, and $(b_{t_j} \rightarrow e_{t_k})$. These derived terms are represented by dashed arrows.

The dependency Z between the transactions t_i and t_j can be derived by determining a *path* from the significant events of t_i to the significant events of t_j and vice versa. In the first direction we try to find a connection of arrows from the vertex b_{t_i} to b_{t_j}, from b_{t_i} to e_{t_j}, from e_{t_i} to b_{t_j}, and from e_{t_i} to e_{t_j}. The same is done for the other direction. This is a classical problem which can be computed by Dijkstra's algorithm [Dij59] or by Floyd's algorithm [Flo62]. Dijkstra's algorithm evaluates a path (shortest path) from one vertex to all other vertices and Floyd's algorithm solves the all-pairs shortest path problem. The complexity of both algorithms is polynominal.

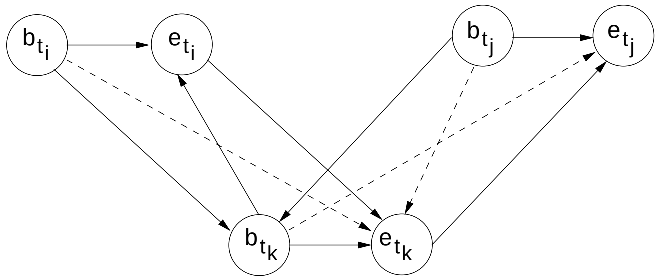

Figure 5: Graph of $lap(t_i, t_k)$ and $inc(t_j, t_k)$

In our example we can derive the following relations between the transactions t_i and t_j (compare Figure 5):

$$
\begin{aligned}
(b_{t_i} \to b_{t_k}) \wedge (b_{t_k} \to e_{t_j}) &\Rightarrow (b_{t_i} \to e_{t_j}) \\
(e_{t_i} \to e_{t_k}) \wedge (e_{t_k} \to e_{t_j}) &\Rightarrow (e_{t_i} \to e_{t_j}) \\
(b_{t_j} \to b_{t_k}) \wedge (b_{t_k} \to e_{t_i}) &\Rightarrow (b_{t_j} \to e_{t_i})
\end{aligned}
$$

The transitive dependency Z between the transactions t_i and t_j may be $lap(t_i, t_j)$, $inc(t_i, t_j)$, $seq(t_i, t_j)$, $lap(t_j, t_i)$, $inc(t_j, t_i)$, and $seq(t_j, t_i)$. Considering the ordering term of these execution dependencies only the dependencies $lap(t_i, t_j)$ and $inc(t_j, t_i)$ consist of the three terms $(b_{t_j} \to e_{t_i})$, $(e_{t_i} \to e_{t_j})$, and $(b_{t_i} \to e_{t_j})$ evaluated by the algorithm above:

$$
\begin{aligned}
lap(t_i, t_j): &\quad (b_{t_i} \to b_{t_j}) \wedge (b_{t_j} \to e_{t_i}) \wedge (e_{t_i} \to e_{t_j}) \wedge (b_{t_i} \to e_{t_j}) \\
inc(t_i, t_j): &\quad (b_{t_i} \to b_{t_j}) \wedge (b_{t_j} \to e_{t_i}) \wedge (e_{t_j} \to e_{t_i}) \wedge (b_{t_i} \to e_{t_j}) \\
seq(t_i, t_j): &\quad (b_{t_i} \to b_{t_j}) \wedge (e_{t_i} \to b_{t_j}) \wedge (e_{t_i} \to e_{t_j}) \wedge (b_{t_i} \to e_{t_j}) \\
lap(t_j, t_i): &\quad (b_{t_j} \to b_{t_i}) \wedge (b_{t_j} \to e_{t_i}) \wedge (e_{t_j} \to e_{t_i}) \wedge (b_{t_i} \to e_{t_j}) \\
inc(t_j, t_i): &\quad (b_{t_j} \to b_{t_i}) \wedge (b_{t_j} \to e_{t_i}) \wedge (e_{t_i} \to e_{t_j}) \wedge (b_{t_i} \to e_{t_j}) \\
seq(t_j, t_i): &\quad (b_{t_j} \to b_{t_i}) \wedge (b_{t_j} \to e_{t_i}) \wedge (e_{t_j} \to e_{t_i}) \wedge (e_{t_j} \to b_{t_i})
\end{aligned}
$$

This solution is also illustrated in Figure 6. In the set of ordering terms derived by the algorithm a relationship between the begin events of the transactions t_i and t_j is absent. Thus, we have to distinguish two cases. In case the start of transaction t_i precedes the start of t_j we conclude the transitive dependency is $lap(t_i, t_j)$. In contrast, the start of transaction t_j may precede the start of transaction t_i which leads to a dependency $inc(t_j, t_i)$.

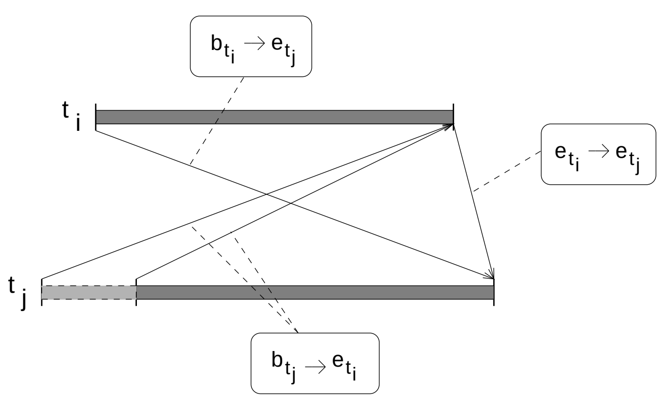

Figure 6: Transitive Dependencies between the Transactions t_i and t_j

Applying the method above leads to a set of paths from the vertices of transaction t_i to the vertices of transaction t_j and vice versa. These paths represent ordering terms. The algorithm evaluates one to four ordering terms. More ordering terms cannot be evaluated because we have only eight possible terms (1)–(4) and ($\overline{1}$)–($\overline{4}$) and the combination of one term and its inverse term is always contradictory, e.g. $(1 \wedge \overline{1})$. Thus, more than four terms, e.g. $(1 \wedge 2 \wedge 3 \wedge 4 \wedge \overline{1})$, are always contradictory and cannot be results of the algorithm.

In the following we investigate the cases in which the algorithm evaluates one, two, three, and four terms, respectively. In doing so, we combine the basic ordering terms of the formulas (1)–(4) including the inverse formulas ($\overline{1}$)–($\overline{4}$). From these term combinations as results of the algorithm we can derive the dependencies between the transactions t_i and t_j. These dependencies are candidates for the transitive dependency Z.

1. In case we only evaluate one term as a path from the significant events of transaction t_i to t_j by the algorithm then dependency Z can be of one of the following dependencies. The dependency $any(t_i, t_j)$ stands for the disjunction of $lap(t_i, t_j)$, $inc(t_i, t_j)$, and $seq(t_i, t_j)$.

$$(1): \quad (b_{t_i} \rightarrow b_{t_j}) \quad \Rightarrow \quad any(t_i, t_j) \tag{13}$$
$$(2): \quad (b_{t_i} \rightarrow e_{t_j}) \quad \Rightarrow \quad any(t_i, t_j) \vee lap(t_j, t_i) \vee inc(t_j, t_i) \tag{14}$$
$$(3): \quad (e_{t_i} \rightarrow e_{t_j}) \quad \Rightarrow \quad lap(t_i, t_j) \vee seq(t_i, t_j) \vee inc(t_j, t_i) \tag{15}$$
$$(4): \quad (e_{t_i} \rightarrow b_{t_j}) \quad \Rightarrow \quad seq(t_i, t_j) \tag{16}$$

   Due to symmetry, the inverse terms ($\overline{1}$)–($\overline{4}$) are not considered.

2. On the other hand, the algorithm may evaluate two terms as paths, e.g. the terms $(1 \wedge 3)$. We investigate combinations of two terms and derive dependencies which are valid as transitive dependency Z. In this discussion, we follow four rules which are directly derived from the formulas (5)–(12):

   (a) A combination of a term and the inverse form of this term is always contradictory, e.g. $(1 \wedge \overline{1})$.

   (b) Combining term (4) with another term is equivalent to term (4). In other words, term (4) implies the terms (1)–(4) (see formula (8)). Thus, an inverse term always contradicts term (4). For example, in case of a combination $(\overline{3} \wedge 4)$, term (4) implies term (3) which contradicts term ($\overline{3}$).

   (c) All terms (1)–(4) implies term (2) (see the formulas (5)–(8)). Thus, a combination of these terms with term (2) can be reduced to a combination without term (2), e.g. $(1 \wedge 2) \equiv (1)$.

   (d) The term ($\overline{2}$) contradics the terms (1)–(4). This directly follows from the first and the third rule.

   In the following, we consider the combinations of two terms which may be results of the application of the algorithm. We start with term (1). A combination with term (2) is equivalent with term (1) because the last term implies term (2) (see the third rule). Due to the last rule, a combination with ($\overline{2}$) is contradictory. On the other hand, the term (3) and the inverse term are valid combinations to term (1). The second rule indicates that a combination of term (4) with term (1) is equivalent to term (4) which is already expressed in formula (16). Finally, the combination with term ($\overline{4}$) is valid. The intermediate results of the contradictory and implicit terms are listed below:

$$(1 \wedge 2) \equiv (1)$$
$$(1 \wedge \overline{2}) \equiv (1 \wedge 2 \wedge \overline{2})$$
$$(1 \wedge 4) \equiv (4)$$

The term (2) has additionally to be combined with term (3), ($\overline{3}$), (4), and ($\overline{4}$). Due to the third rule, the terms (3) and (4) implies term (2). Thus, these combinations are equivalent to the formulas (15) and (16). The negative terms are valid combinations to term (2). Moreover, term (3) has to be combined with term (4) and ($\overline{4}$). Due to the last rule, only the combination with term ($\overline{4}$) provide new results. The other are summarized in the following:

$$\begin{aligned} (2 \wedge 3) &\equiv (3) \\ (2 \wedge 4) &\equiv (4) \\ (3 \wedge 4) &\equiv (4) \end{aligned}$$

The valid term combination and the corresponding dependencies between the transactions t_i and t_j are listed below:

$$\begin{aligned}
(1 \wedge 3): &\quad (b_{t_i} \rightarrow b_{t_j}) \wedge (e_{t_i} \rightarrow e_{t_j}) &\Rightarrow&\quad lap(t_i, t_j) \vee seq(t_i, t_j) & (17)\\
(1 \wedge \overline{3}): &\quad (b_{t_i} \rightarrow b_{t_j}) \wedge (e_{t_j} \rightarrow e_{t_i}) &\Rightarrow&\quad inc(t_i, t_j) & (18)\\
(1 \wedge \overline{4}): &\quad (b_{t_i} \rightarrow b_{t_j}) \wedge (b_{t_j} \rightarrow e_{t_i}) &\Rightarrow&\quad lap(t_i, t_j) \vee inc(t_i, t_j) & (19)\\
(2 \wedge \overline{3}): &\quad (b_{t_i} \rightarrow e_{t_j}) \wedge (e_{t_j} \rightarrow e_{t_i}) &\Rightarrow&\quad lap(t_j, t_i) \vee inc(t_i, t_j) & (20)\\
(2 \wedge \overline{4}): &\quad (b_{t_i} \rightarrow e_{t_j}) \wedge (b_{t_j} \rightarrow e_{t_i}) &\Rightarrow&\quad lap(t_i, t_j) \vee inc(t_i, t_j) \vee lap(t_j, t_i) \vee inc(t_j, t_i) & (21)\\
(3 \wedge \overline{4}): &\quad (e_{t_i} \rightarrow e_{t_j}) \wedge (b_{t_j} \rightarrow e_{t_i}) &\Rightarrow&\quad lap(t_i, t_j) \vee inc(t_j, t_i) & (22)
\end{aligned}$$

3. The algorithm may also evaluate three terms (paths). Thus, we have to consider the term combinations ($1 \wedge 2 \wedge 3$), ($1 \wedge 2 \wedge 4$), ($1 \wedge 3 \wedge 4$), and ($2 \wedge 3 \wedge 4$) including the negative form of the terms. First, we reduce the term combinations to the combinations which consist of term (2). We omit the combinations which consist of the term ($\overline{2}$) because the term (1) implies (2) which contradicts ($\overline{2}$) (see the last rule):

$$\begin{aligned} (1 \wedge \overline{2} \wedge 3) &\equiv (1 \wedge 2 \wedge \overline{2} \wedge 3) \\ (1 \wedge \overline{2} \wedge \overline{3}) &\equiv (1 \wedge 2 \wedge \overline{2} \wedge \overline{3}) \\ (1 \wedge \overline{2} \wedge 4) &\equiv (1 \wedge 2 \wedge \overline{2} \wedge 4) \\ (1 \wedge \overline{2} \wedge \overline{4}) &\equiv (1 \wedge 2 \wedge \overline{2} \wedge \overline{4}) \end{aligned}$$

The combination ($1 \wedge 2 \wedge 3$) is equivalent to ($1 \wedge 3$) which is already considered in formula (17). The same is valid for ($1 \wedge 2 \wedge \overline{3}$) which is equivalent to ($1 \wedge \overline{3}$). These terms are listed below:

$$\begin{aligned} (1 \wedge 2 \wedge 3) &\equiv (1 \wedge 3) \\ (1 \wedge 2 \wedge \overline{3}) &\equiv (1 \wedge \overline{3}) \end{aligned}$$

Due to the second rule, all combinations with the basic terms (1)–(3) involving term (4) are represented by formula (16). In contrast, a combination of term (4) with a inverse term is always contradictory. In the following we state these intermediate results:

$$\begin{aligned} (1 \wedge 2 \wedge 4) &\equiv (4) \\ (1 \wedge 3 \wedge 4) &\equiv (4) \\ (1 \wedge \overline{3} \wedge 4) &\equiv (1 \wedge \overline{3} \wedge 2 \wedge 3 \wedge 4) \\ (2 \wedge 3 \wedge 4) &\equiv (4) \\ (2 \wedge \overline{3} \wedge 4) &\equiv (2 \wedge \overline{3} \wedge 1 \wedge 3 \wedge 4) \end{aligned}$$

Thus, we have only to consider term combinations without ($\overline{2}$) and (4). ($1 \wedge 2 \wedge \overline{4}$) is equivalent to ($1 \wedge \overline{4}$) (see formula (19)), ($1 \wedge 3 \wedge \overline{4}$) is valid, ($1 \wedge \overline{3} \wedge \overline{4}$) is equivalent to ($1 \wedge \overline{3}$) (see formula

(18)), $(2 \wedge 3 \wedge \overline{4})$ is equivalent to $(3 \wedge \overline{4})$ as considered in formula (22), and $(2 \wedge \overline{3} \wedge \overline{4})$ is equivalent to $(2 \wedge \overline{3})$ presented in formula (20). The intermediate results of the contradictory and implicit terms are listed in the following:

$$\begin{aligned}
(1 \wedge 2 \wedge \overline{4}) &\equiv (1 \wedge \overline{4}) \\
(1 \wedge \overline{3} \wedge \overline{4}) &\equiv (1 \wedge \overline{3}) \\
(2 \wedge 3 \wedge \overline{4}) &\equiv (3 \wedge \overline{4}) \\
(2 \wedge \overline{3} \wedge \overline{4}) &\equiv (2 \wedge \overline{3})
\end{aligned}$$

In conclusion, only one valid combination of three terms is left.

$$(1 \wedge 3 \wedge \overline{4}): \quad (b_{t_i} \rightarrow b_{t_j}) \wedge (e_{t_i} \rightarrow e_{t_j}) \wedge (b_{t_j} \rightarrow e_{t_i}) \quad \Rightarrow \quad lap(t_i, t_j) \tag{23}$$

4. After investigating combinations of three or less terms, we consider the case in which four terms are evaluated by the algorithm. Due to symmetry, the second and third rule we only investigate the combinations with the terms (1), $(\overline{4})$ and (2): $(1 \wedge 2 \wedge 3 \wedge \overline{4})$ and $(1 \wedge 2 \wedge \overline{3} \wedge \overline{4})$. The first combination $(1 \wedge 2 \wedge 3 \wedge \overline{4})$ is equivalent to the combination $(1 \wedge 3 \wedge \overline{4})$ which is discussed in formula (23) and $(1 \wedge 2 \wedge \overline{3} \wedge \overline{4})$ is equivalent to $(1 \wedge \overline{3})$. Thus, in the formulas (13)–(23) all possible combinations of terms are presented. These contradictory and implicit term combinations are stated below:

$$\begin{aligned}
(1 \wedge 2 \wedge 3 \wedge 4) &\equiv (4) \\
(1 \wedge \overline{2} \wedge 3 \wedge 4) &\equiv (1 \wedge 2 \wedge \overline{2} \wedge 3 \wedge 4) \\
(1 \wedge 2 \wedge \overline{3} \wedge 4) &\equiv (1 \wedge 2 \wedge \overline{3} \wedge 3 \wedge 4) \\
(1 \wedge 2 \wedge 3 \wedge \overline{4}) &\equiv (1 \wedge 3 \wedge \overline{4}) \\
(1 \wedge \overline{2} \wedge \overline{3} \wedge 4) &\equiv (1 \wedge 2 \wedge \overline{2} \wedge \overline{3} \wedge 3 \wedge 4) \\
(1 \wedge \overline{2} \wedge 3 \wedge \overline{4}) &\equiv (1 \wedge 2 \wedge \overline{2} \wedge 3 \wedge \overline{4}) \\
(1 \wedge 2 \wedge \overline{3} \wedge \overline{4}) &\equiv (1 \wedge \overline{3}) \\
(1 \wedge \overline{2} \wedge \overline{3} \wedge \overline{4}) &\equiv (1 \wedge 2 \wedge \overline{2} \wedge \overline{3} \wedge \overline{4})
\end{aligned}$$

Applying the method described in this section leads to one of the transitive dependencies stated by the formulas (13)–(23). Please note, a transaction closure consisting of n transactions has $\frac{n(n-1)}{2}$ dependencies (edges between n vertices). The full set of rules evaluated by the algorithm is summarized in the Appendix A.

5 Combining Execution and Termination Dependencies

Execution dependencies are defined over the begin and end events of related transactions. In contrast, termination dependencies explicitly distinguish between the commit and abort of a transaction as termination event. Investigating constraints on the occurrence of the significant termination events commit and abort leads to different termination dependencies. In case of two transactions t_i and t_j there are four possible combinations of termination events:

(1) both transactions abort (a_{t_i}, a_{t_j}),

(2/3) one transaction commits whereas the other one aborts $(a_{t_i}, c_{t_j})/(c_{t_i}, a_{t_j})$, and

(4) both transactions commit (c_{t_i}, c_{t_j}).

These termination event combinations may be valid in any order (denoted by $\surd$) or are not valid (denoted by —). As depicted in Table 1, we identify five dependencies as applicable according to real-world application semantics. The termination dependency between t_i and t_j is called *vital-dependent*, denoted as $vital\_dep(t_i, t_j)$, if the transactions are *abort-dependent* on each other. In detail, the abortion of transaction t_i leads to the abortion of t_j and vice versa. Thus, either the commit of t_i and t_j or the abort of these transactions are valid. The vital-dependent dependency is (as the name suggests) a combination of the dependencies *vital* and *dependent*. The *vital* dependency between two transactions t_i and t_j, denoted as $vital(t_i, t_j)$, concerns the case where the abortion of transaction t_i leads to the abortion of transaction t_j. In comparison to a vital transaction, a *dependent* transaction t_j has to abort if transaction t_i aborts. This fact is defined as $dep(t_i, t_j)$. Two transactions are called *exclusive* dependent on each other, denoted as $exc(t_i, t_j)$, if only one of the transactions is allowed to finish successfully. Our fifth dependency concerns the case where each combination of transaction termination events is valid. Therefore, the involved transactions t_i and t_j are called *independent*, denoted as $indep(t_i, t_j)$. For a formal definition of the termination dependencies and more detail information see [STS98b].

t_i	t_j	$vital\_dep(t_i, t_j)$	$vital(t_i, t_j)$	$dep(t_i, t_j)$	$exc(t_i, t_j)$	$indep(t_i, t_j)$
a_{t_i}	a_{t_j}	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
a_{t_i}	c_{t_j}	—	—	$\surd$	$\surd$	$\surd$
c_{t_i}	a_{t_j}	—	$\surd$	—	$\surd$	$\surd$
c_{t_i}	c_{t_j}	$\surd$	$\surd$	$\surd$	—	$\surd$

Table 1: Termination Dependencies between two Transactions t_i and t_j

Combining execution and termination dependencies leads to an ordering of the termination events of the related transactions. The general parallel execution dependency has no influence on the termination dependencies, because we cannot state whether the termination of t_i precedes the termination of t_j or vice versa. Therefore, we omit the discussion of combining $par(t_i, t_j)$ with the termination dependencies.

The execution dependency parallel strict overlapping $lap(t_i, t_j)$ requires that the termination of transaction t_i precedes the termination of t_j. Combining this dependency with the termination dependencies leads to the results represented in Table 2.

t_i	t_j	$vital\_dep(t_i, t_j)$ $lap(t_i, t_j)$	$vital(t_i, t_j)$ $lap(t_i, t_j)$	$dep(t_i, t_j)$ $lap(t_i, t_j)$	$exc(t_i, t_j)$ $lap(t_i, t_j)$	$indep(t_i, t_j)$ $lap(t_i, t_j)$
a_{t_i}	a_{t_j}	$a_{t_i} \rightarrow a_{t_j}$	$a_{t_i} \rightarrow a_{t_j}$	$a_{t_i} \rightarrow a_{t_j}$	$a_{t_i} \rightarrow a_{t_j}$	$a_{t_i} \rightarrow a_{t_j}$
a_{t_i}	c_{t_j}	—	—	$a_{t_i} \rightarrow c_{t_j}$	$a_{t_i} \rightarrow c_{t_j}$	$a_{t_i} \rightarrow c_{t_j}$
c_{t_i}	a_{t_j}	—	$c_{t_i} \rightarrow a_{t_j}$	—	$c_{t_i} \rightarrow a_{t_j}$	$c_{t_i} \rightarrow a_{t_j}$
c_{t_i}	c_{t_j}	$c_{t_i} \rightarrow c_{t_j}$	$c_{t_i} \rightarrow c_{t_j}$	$c_{t_i} \rightarrow c_{t_j}$	—	$c_{t_i} \rightarrow c_{t_j}$

Table 2: Termination Dependencies in Combination with Parallel Strict Overlapping

In comparison to the parallel strict overlapping dependency, the parallel including dependency influences the termination dependencies in the opposite direction. Here, the termination of transaction t_j precedes the termination of transaction t_i. The combinations of parallel including and the termination dependencies are illustrated in Table 3. A combination of the dependency $lap(t_j, t_i)$ with the

t_i	t_j	$vital\_dep(t_i, t_j)$ $inc(t_i, t_j)$	$vital(t_i, t_j)$ $inc(t_i, t_j)$	$dep(t_i, t_j)$ $inc(t_i, t_j)$	$exc(t_i, t_j)$ $inc(t_i, t_j)$	$indep(t_i, t_j)$ $inc(t_i, t_j)$
a_{t_i}	a_{t_j}	$a_{t_j} \rightarrow a_{t_i}$	$a_{t_j} \rightarrow a_{t_i}$	$a_{t_j} \rightarrow a_{t_i}$	$a_{t_j} \rightarrow a_{t_i}$	$a_{t_j} \rightarrow a_{t_i}$
a_{t_i}	c_{t_j}	—	—	$c_{t_j} \rightarrow a_{t_i}$	$c_{t_j} \rightarrow a_{t_i}$	$c_{t_j} \rightarrow a_{t_i}$
c_{t_i}	a_{t_j}	—	$a_{t_j} \rightarrow c_{t_i}$	—	$a_{t_j} \rightarrow c_{t_i}$	$a_{t_j} \rightarrow c_{t_i}$
c_{t_i}	c_{t_j}	$c_{t_j} \rightarrow c_{t_i}$	$c_{t_j} \rightarrow c_{t_i}$	$c_{t_j} \rightarrow c_{t_i}$	—	$c_{t_j} \rightarrow c_{t_i}$

Table 3: Termination Dependencies in Combination with Parallel Including

termination dependencies where transaction t_i is the first argument and t_j the second argument, e.g. $vital\_dep(t_i, t_j)$, leads to the same results concerning the termination events.

In contrast to the parallel execution dependencies, the sequential execution dependency may contradict termination event combinations. Here, we explicitly distinguish between the termination events commit and abort. We define the following two additional dependencies:

Definition 5.1 (Sequential-Commit) *Two different transactions t_i and t_j are executed* sequentially after commit *if and only if the commitment of t_i precedes the begin of t_j:*

$$seq\_commit(t_i, t_j) \quad :\Leftrightarrow \quad (c_{t_i} \rightarrow b_{t_j})$$

Definition 5.2 (Sequential-Abort) *Two different transactions t_i and t_j are executed* sequentially after abort *if and only if the abortion of t_i precedes the begin of t_j:*

$$seq\_abort(t_i, t_j) \quad :\Leftrightarrow \quad (a_{t_i} \rightarrow b_{t_j})$$

In case transaction t_j is executed sequential after the commit of t_i (seq_commit(t_i, t_j)), the event combination of the abortion of t_i (a_{t_i}) and the commit of t_j (c_{t_j}) is invalid. In contrast, the abortion of both transactions is always valid. The commitment of transaction t_i precedes the termination of t_j because of the sequential-commit dependency. The resulting dependency combinations are illustrated in Table 4.

t_i	t_j	$vital\_dep(t_i, t_j)$ $seq\_commit(t_i, t_j)$	$vital(t_i, t_j)$ $seq\_commit(t_i, t_j)$	$dep(t_i, t_j)$ $seq\_commit(t_i, t_j)$	$exc(t_i, t_j)$ $seq\_commit(t_i, t_j)$	$indep(t_i, t_j)$ $seq\_commit(t_i, t_j)$
a_{t_i}	a_{t_j}	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
a_{t_i}	c_{t_j}	—	—	—	—	—
c_{t_i}	a_{t_j}	—	$c_{t_i} \rightarrow a_{t_j}$	—	$c_{t_i} \rightarrow a_{t_j}$	$c_{t_i} \rightarrow a_{t_j}$
c_{t_i}	c_{t_j}	$c_{t_i} \rightarrow c_{t_j}$	$c_{t_i} \rightarrow c_{t_j}$	$c_{t_i} \rightarrow c_{t_j}$	—	$c_{t_i} \rightarrow c_{t_j}$

Table 4: Termination Dependencies in Combination with Sequential-Commit

If we have a closer look at Table 4, we see that the combination of the exclusive termination dependency with the sequential-commit dependency only allows the abortion of transaction t_j (a_{t_j}). Thus, t_j does not need to be executed. On the other hand, a combination with the termination dependencies vital-dependent and dependent cannot be fulfilled. In this cases, transaction t_i is only allowed to commit if transaction t_j commits, too. Due to the sequential-commit dependency, the commitment of t_i has further to precede the commitment and the begin of t_j: ($c_{t_i} \rightarrow b_{t_j} \rightarrow c_{t_j}$). However, in case transaction t_j aborts after the commit of t_i, transaction t_i cannot be aborted afterwards. On the other hand, transaction t_i cannot be forced to commit if t_j is executed after the commit of t_i. Thus, these dependency combinations are invalid.

The sequential-abort dependency disallows the termination event combination where transaction t_i commits (see Table 5). On the other hand, the abortion of t_i always precedes the termination of t_j. A combination of this execution dependency and the vital-dependent or vital termination dependency makes no sense, because only the abortion of both transactions is valid.

t_i	t_j	$vital\_dep(t_i, t_j)$ $seq\_abort(t_i, t_j)$	$vital(t_i, t_j)$ $seq\_abort(t_i, t_j)$	$dep(t_i, t_j)$ $seq\_abort(t_i, t_j)$	$exc(t_i, t_j)$ $seq\_abort(t_i, t_j)$	$indep(t_i, t_j)$ $seq\_abort(t_i, t_j)$
a_{t_i}	a_{t_j}	$a_{t_i} \rightarrow a_{t_j}$	$a_{t_i} \rightarrow a_{t_j}$	$a_{t_i} \rightarrow a_{t_j}$	$a_{t_i} \rightarrow a_{t_j}$	$a_{t_i} \rightarrow a_{t_j}$
a_{t_i}	c_{t_j}	—	—	$a_{t_i} \rightarrow c_{t_j}$	$a_{t_i} \rightarrow c_{t_j}$	$a_{t_i} \rightarrow c_{t_j}$
c_{t_i}	a_{t_j}	—	—	—	—	—
c_{t_i}	c_{t_j}	—	—	—	—	—

Table 5: Termination Dependencies in Combination with Sequential-Abort

In summary, the following dependency combinations are contradictory ($\leftrightsquigarrow$). The sequential dependency is a disjunction of the sequential-commit and the sequential-abort dependency. Transactions which are vital-dependent cannot combined neither with sequential-commit nor with sequential-abort. From this follows that this termination dependency also contradicts the sequential dependency.

$$seq\_commit(t_i, t_j) \leftrightsquigarrow vital\_dep(t_i, t_j) \tag{24}$$
$$seq\_commit(t_i, t_j) \leftrightsquigarrow dep(t_i, t_j) \tag{25}$$
$$seq\_commit(t_i, t_j) \leftrightsquigarrow exc(t_i, t_j) \tag{26}$$
$$seq\_abort(t_i, t_j) \leftrightsquigarrow vital\_dep(t_i, t_j) \tag{27}$$
$$seq\_abort(t_i, t_j) \leftrightsquigarrow vital(t_i, t_j) \tag{28}$$
$$seq(t_i, t_j) \leftrightsquigarrow vital\_dep(t_i, t_j) \tag{29}$$

In this section, the impact of the execution dependencies on the termination dependencies was considered. Execution dependencies partially restricts the occurrence of termination events by adding an ordering constraint on valid termination event combinations of the corresponding transactions. We identified combinations of execution and termination dependencies which are incompatible.

6 Execution and Termination Dependencies in an Example Scenario

The following example is intended to clarify the application of transaction closures. Especially, we show the derivation of transitive execution dependencies and their combination with related termination dependencies in such a transaction closure. The transaction closure in our example can be considered as a workflow with special dependencies among the related transactions.

Example 6.1 *A commonly used example is a travel planning activity. In our example this activity consists of the recording of the customer's data, reserving a flight, applying a visa, booking a room including a sport car or a family car rental. This activity is modeled as a transaction closure with the transactions t_2, t_3, t_4, t_5, t_6, t_7, and the coordinating root transaction t_1.*

Transaction t_2 represents the recording of the customer's data which is done independently whether the trip reservation is successful or not. The flight reservation (t_3) is essential for the trip. After reserving a flight, we apply a visa (t_5). Moreover, a room in a hotel (t_4) may be reserved and a sport car rent (t_6). If no sport car is available, we try to rent a family car (t_7).

From this scenario follows that the transactions t_2, t_3, and t_4 which are child transactions of the root transaction t_1 are connected to t_1 by the following termination dependencies:

$$indep(t_1, t_2) \wedge vital\_dep(t_1, t_3) \wedge vital(t_1, t_4)$$

Furthermore, these transactions are executed parallel to t_1 while the transactions t_3 and t4 have to terminate before the end of the travel planning.

$$par(t_1, t_2) \wedge inc(t_1, t_3) \wedge inc(t_1, t_4)$$

Transactions t_5 is a child transaction of transaction t_3 and the transactions t_6, and t_7 are child transactions of transaction t_4. Transactions t_3 is connected to its parent transactions by the vital-dependent termination dependency and the other by a vital termination dependency. Thus, the abortion of a parent leads to the abortion of the child transactions. Furthermore, the child transactions are executed sequentially after the parent:

$$\begin{aligned} vital\_dep(t_3, t_5) \wedge \quad & vital(t_4, t_6) \quad \wedge vital(t_4, t_7) \\ seq(t_3, t_5) \wedge \quad & seq(t_4, t_6) \quad \wedge seq\_commit(t_4, t_7) \end{aligned}$$

Additionally, we specify a parallel strict overlapping dependency between the sibling transactions t_3 *and* t_4 *and a sequential-abort dependency between* t_6 *and* t_7*:*

$$lap(t_3, t_4) \wedge seq\_abort(t_6, t_7)$$

Thus, the flight reservation and the hotel reservation is executed parallel whereas the flight reservation has to terminate before a room in a hotel is booked.

Our example transaction closure is graphically illustrated in Figure 7 where the arrows denote the direction of the abort dependencies. For example, $t_1 \longrightarrow t_4$ *means that the abortion of transaction* t_1 *leads to the abortion of* t_4*. The termination dependencies are graphically represented as follows:*

$$\begin{array}{rcl} vital(t_i, t_j) & corresponds\ to & t_i \longrightarrow t_j \\ dep(t_i, t_j) & corresponds\ to & t_i \longleftarrow t_j \\ vital\_dep(t_i, t_j) & corresponds\ to & t_i \longleftrightarrow t_j \\ exc(t_i, t_j) & corresponds\ to & t_i \not\longleftrightarrow t_j \\ indep(t_i, t_j) & corresponds\ to & t_i \longrightarrow t_j \end{array}$$

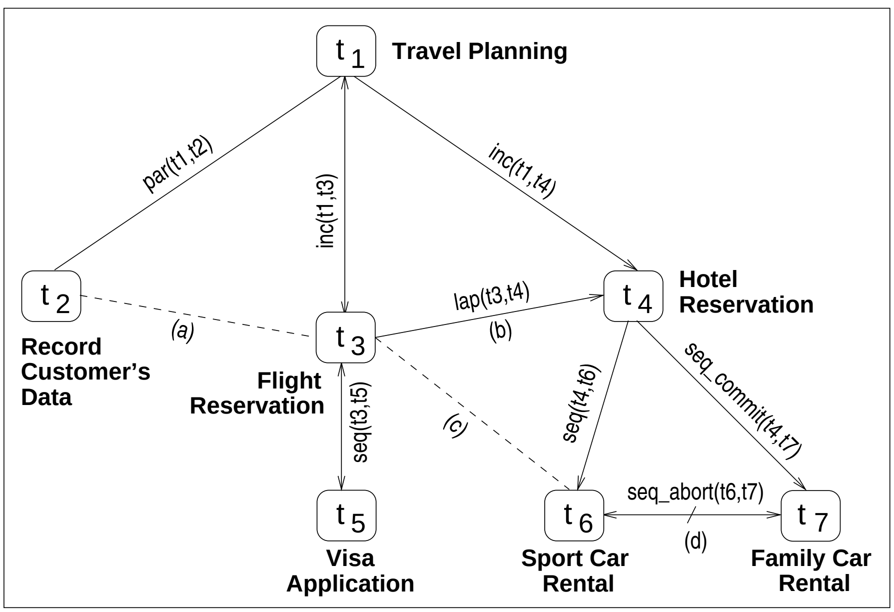

Figure 7: A Sample Transaction Closure for Travel Planning

From our dependency definitions, the contradictory dependency combinations summarized in the formulas (24)–(29), and the rules in Appendix A we can now derive the transitive and valid execution dependencies in the underlying transaction closure. Before considering transitive dependencies, we investigate the combinations of the specified termination and execution dependencies. All combinations including the parallel execution dependency are valid (see Section 5). Therefore, we consider the sequential dependencies and their combinations with the termination dependencies in detail. In our example the following dependency combinations occur:

$$\begin{array}{rcl} vital\_dep(t_3, t_5) & \wedge & seq(t_3, t_5) \\ vital(t_4, t_6) & \wedge & seq(t_4, t_6) \\ vital(t_4, t_7) & \wedge & seq\_commit(t_4, t_7) \\ exc(t_6, t_7) & \wedge & seq\_abort(t_6, t_7) \end{array}$$

The vital dependency can be combined with the sequential-commit dependency as well as with the sequential dependency. The exclusive dependency is compatible with the sequential-abort dependency. In contrast, a combination of vital-dependent and sequential is incompatible (see formula 29). In consequence, either the termination or the execution dependency has to be changed. Considering the semantics of the application, we have two possibilities. Either the termination dependency is set to vital or the execution dependency is adapted to parallel.

After considering the dependency combinations, we discuss some interesting cases of transitive execution dependencies which refer to the dashed lines and the corresponding letter (a), (b), (c), and (d) in Figure 7:

(a) *We start with the consideration of the transactions t_1, t_2, and t_3. As specified, t_1 is executed parallel to t_2 and parallel including to t_3. From these basic dependencies we can derive that the execution dependency between the transactions t_2 and t_3 may be transitively of any introduced execution dependency (see the Rules 37 and 46 in Appendix A):*

$$\begin{aligned} & par(t_1,t_2) \wedge inc(t_1,t_3) \\ \overset{Def\ 3.3}{\equiv} \quad & (lap(t_1,t_2) \vee inc(t_1,t_2)) \wedge inc(t_1,t_3) \\ \equiv \quad & (lap(t_1,t_2) \wedge inc(t_1,t_3)) \vee (inc(t_1,t_2) \wedge inc(t_1,t_3)) \\ \overset{Rule\ 37,46}{\Longrightarrow} \quad & (inc(t_2,t_3) \vee lap(t_3,t_2) \vee seq(t_3,t_2)) \vee (any(t_2,t_3) \vee any(t_3,t_2)) \\ \equiv \quad & any(t_2,t_3) \vee any(t_3,t_2) \end{aligned}$$

In relation to transaction t_1, both transactions the recording of the customer's data and the flight reservation are executed parallel. Thus, there is no constraint on the execution of these two transaction in relation to each other. In this case, the transaction designer can specify an arbitrary execution dependency. The only restriction is that the execution dependency has be compatible with the transitive termination dependency between t_1 and t_3.

(b) *An execution dependency between the transactions t_3 and t_4 is already specified. Thus, we have to check whether the parallel strict overlapping dependency is a valid specification or not. Again the Rule 46 is used to derive the transitive dependency between these transactions over transaction t_1:*

$$inc(t_1,t_3) \wedge inc(t_1,t_4) \overset{Rule\ 46}{\Longrightarrow} any(t_3,t_4) \vee any(t_4,t_3)$$

Thus, the dependency $lap(t_3,t_4)$ is valid.

(c) *Considering the transactions t_3, t_4, and t_6 we derive the transitive dependency between t_3 and t_6 with Rule 32:*

$$lap(t_3,t_4) \wedge seq(t_4,t_6) \overset{Rule\ 32}{\Longrightarrow} seq(t_3,t_6)$$

The hotel reservation finished after the flight reservation and the sport car rental follows the hotel reservation. Thus, renting a sport car is done after a room in a hotel is booked. Because of the vital termination dependency between the transactions t_4 and t_6, the car rental is only executed if t_4 commits. Therefore, the dependency $seq(t_4,t_6)$ is refined to a sequential-commit dependency.

(d) *Finally, we consider the sequential-abort dependency between the transactions t_6 and t_7. Due to the termination dependencies between the transactions t_4, t_6, and t_7, the scenario is as follows. If a hotel room cannot be booked, then the car rental is aborted, too. This case is reflected by the sequential-commit dependency between t_4 and t_7. Transaction t_7 starts executing only after a successful execution of transaction t_4. Furthermore, the exclusive dependency between the car* *rental transactions express that in case one of the transactions finished successfully the other activity is canceled. Therefore, at most one car is rent. Additionally, transaction t_7 is executed sequentially after the abortion of transaction t_6. In the case that no sport car is available, we try to rent a family car. Thus, there is a priority specified between the two car rentals. According to Rule 56 the specification of these dependencies are valid:*

$$seq(t_4, t_6) \wedge seq(t_4, t_7) \stackrel{Rule\ 56}{\Longrightarrow} any(t_6, t_7) \vee any(t_7, t_6)$$

Example 6.1 showed that we are able to detect contradictory and redundant parts of a transaction closure definition. Different execution dependencies in combination with the termination dependencies allows the transaction designer to specify complex applications. The transitivity property of the execution dependencies is essential to conclude how two arbitrary transactions are related in terms of execution and termination ordering.

7 Conclusions and Outlook

The original model of nested transactions is not suitable for generalized transaction models where child transactions may survive the termination of the parent transaction. The concept of transaction closures together with the dependencies introduced provide an appropriate model for discussing such extended models (as proposed in e.g. [Elm92]). Transaction closures are collections of transactions where the connection is given by termination conditions between transactions and their immediate child transactions. In this paper, we extended the framework by execution dependencies. Here, we explicitly distinguish between different kinds of parallel and sequential execution dependencies. The framework allows to effectively compute derived execution conditions, and, thus, it builds the basis for transaction design tools which can help in designing less failure-prone and more efficient applications.

The concept of transaction closure also captures dependencies considering the aspects of transaction compensation and object visibility constraints [STS98a, STS98b]. However, due to space restrictions we have omit a discussion on the impact of such kinds of dependencies on execution dependencies.

Our future work will concentrate on the enforcement of execution and termination conditions in generalized transaction management systems. Here, we will attempt to adopt the methods proposed in e.g. [GHK93, Gün96] to our framework and provide some extensions to capture the transitive properties of transaction dependencies. Besides this, we will use our framework for describing global transactions in federated database environments where the component systems support different transaction types. We believe that with our framework we are able to exactly formulate the different relationships of the global child transactions which are executed by the possibly heterogeneous and autonomous component database systems.

References

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[BÖH+92] A. Buchmann, M. T. Özsu, M. Hornick, D. Georgakopoulos, and F. Manola. A Transaction Model for Active Distributed Object Systems. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 123–151, Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[CR91] P. K. Chrysanthis and K. Ramamritham. A Formalism for Extended Transaction Models. In G. M. Lohmann, A. Sernadas, and R. Camps, editors, *Proc. of the 17th Int. Conf. on Very Large Data Bases (VLDB'91), Barcelona, Spain*, pages 103–112. Morgan Kaufmann Publishers, San Mateo, CA, September 1991.

[CR94] P. K. Chrysanthis and K. Ramamritham. Synthesis of Extended Transaction Models Using ACTA. *ACM Transaction on Database Systems*, 19(3):450–491, September 1994.

[DHL91] U. Dayal, M. Hsu, and R. Ladin. A Transaction Model for Long-Running Activities. In G. M. Lohmann, A. Sernadas, and R. Camps, editors, *Proc. of the 17th Int. Conf. on Very Large Data Bases (VLDB'91), Barcelona, Spain*, pages 113–122. Morgan Kaufmann Publishers, San Mateo, CA, September 1991.

[DHW95] U. Dayal, E. Hanson, and J. Widom. Active Database Systems. In W. Kim, editor, *Modern Database Systems*, pages 434–456. ACM Press, New York, NJ, 1995.

[Dij59] E. W. Dijkstra. A Note on two Problems in Connexion with Graphs. In Springer-Verlag, editor, *Numerische Mathematik*, pages 269–271, Vol. 1, A. Householder and R. Sauer and E. Stiefel and J. Todd and A. Walther, Berlin, 1959.

[Elm92] A. K. Elmagarmid, editor. *Database Transaction Models For Advanced Applications.* Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[Flo62] R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[GHK93] D. Georgakopoulos, M. Hornick, and P. Krychniak. An Environment for the Specification and Management of Extended Transactions in DOMS. In H.-J. Schek, A. P. Sheth, and B. D. Czejdo, editors, *Proc. of the 3rd Int. Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems (RIDE-IMS'93), Vienna, Austria*, pages 253–257, IEEE Computer Society Press, April 1993.

[GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, April 1995.

[Gün96] R. Günthör. The Dependency Manager: A Base Service for Transactional Workflow Management. In *Proc. of the 6th Int. Workshop on Research Issues in Data Engineering: Interoperability in Nontraditional Database Systems (RIDE-NDS'96), New Orleans, Louisiana*, pages 86–95, IEEE Computer Society Press, February 1996.

[HLM88] M. Hsu, R. Ladin, and D. R. McCarthy. An Execution Model For Active Data Base Management Systems. In C. Beeri, J. W. Schmidt, and U. Dayal, editors, *Proc. of the 3rd Int. Conf. on Data and Knowledge Bases: Improving Usability and Responsiveness, Jerusalem, Israel, June, 1988*, pages 171–179, Morgan Kaufmann Publishers, 1988.

[KR96] M. U. Kamath and K. Ramamritham. Correctness Issues in Workflow Management. *Distributed Systems Engineering*, 3(4), December 1996.

[Mos85] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, Cambridge, MA, 1985.

[RKT+95] M. Rusinkiewicz, W. Klas, T. Tesch, J. Wäsch, and P. Muth. Towards a Cooperative Transaction Model — The Cooperative Activity Model. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *Proc. of the 21st Int. Conf. on Very Large Data Bases (VLDB'95), Zürich, Switzerland*, pages 194–205, Morgan Kaufmann Publishers, September 1995.

[SKS96] N. R. Soparkar, H. F. Korth, and A. Silberschatz. *Time-Constrained Transaction Management: Real-Time Constraints in Database Transaction Systems.* Kluwer Academic Publishers, Boston, 1996.

[STS98a] K. Schwarz, C. Türker, and G. Saake. Analyzing and Formalizing Dependencies in Generalized Transaction Structures. In *Proc. of the Int. Workshop on Issues and Applications of Database Technology (IADT'98), July 6-9, 1998, Berlin, Germany*, 1998. *To appear.*

[STS98b] K. Schwarz, C. Türker, and G. Saake. Transitive Dependencies in Transaction Closures. In *Proc. of the 1998 Int. Database Engineering and Applications Symposium (IDEAS'98), July 8–10, 1998, Cardiff, Wales, UK*, 1998. *To appear.*

A Rules for Transitive Execution Dependencies

The algorithm presented in Section 4 brings out a set of rules for deriving transitive execution dependencies. We omit the discussion of the generale parallel dependency $par(t_i, t_j)$ because its the disjuncition of $lap(t_i, t_j)$ and $inc(t_i, t_j)$. Thus, we can easily derive the rules for the parallel dependency.

In the following, we list the resulting transitive dependency rules which have to hold for three different transactions t_i, t_k, and t_j:

$$lap(t_i,t_k) \wedge lap(t_k,t_j) \Rightarrow lap(t_i,t_j) \vee seq(t_i,t_j) \tag{30}$$
$$lap(t_i,t_k) \wedge inc(t_k,t_j) \Rightarrow any(t_i,t_j) \tag{31}$$
$$lap(t_i,t_k) \wedge seq(t_k,t_j) \Rightarrow seq(t_i,t_j) \tag{32}$$
$$lap(t_i,t_k) \wedge lap(t_j,t_k) \Rightarrow lap(t_i,t_j) \vee inc(t_i,t_j) \vee lap(t_j,t_i) \vee inc(t_j,t_i) \tag{33}$$
$$lap(t_i,t_k) \wedge inc(t_j,t_k) \Rightarrow lap(t_i,t_j) \vee inc(t_j,t_i) \tag{34}$$
$$lap(t_i,t_k) \wedge seq(t_j,t_k) \Rightarrow inc(t_i,t_j) \vee lap(t_j,t_i) \vee seq(t_j,t_i) \tag{35}$$
$$lap(t_k,t_i) \wedge lap(t_k,t_j) \Rightarrow lap(t_i,t_j) \vee inc(t_i,t_j) \vee lap(t_j,t_i) \vee inc(t_j,t_i) \tag{36}$$
$$lap(t_k,t_i) \wedge inc(t_k,t_j) \Rightarrow inc(t_i,t_j) \vee lap(t_j,t_i) \vee seq(t_j,t_i) \tag{37}$$
$$lap(t_k,t_i) \wedge seq(t_k,t_j) \Rightarrow any(t_i,t_j) \tag{38}$$
$$inc(t_i,t_k) \wedge lap(t_k,t_j) \Rightarrow lap(t_i,t_j) \vee inc(t_i,t_j) \tag{39}$$
$$inc(t_i,t_k) \wedge inc(t_k,t_j) \Rightarrow inc(t_i,t_j) \tag{40}$$
$$inc(t_i,t_k) \wedge seq(t_k,t_j) \Rightarrow any(t_i,t_j) \tag{41}$$
$$inc(t_i,t_k) \wedge lap(t_j,t_k) \Rightarrow inc(t_i,t_j) \vee lap(t_j,t_i) \tag{42}$$
$$inc(t_i,t_k) \wedge inc(t_j,t_k) \Rightarrow lap(t_i,t_j) \vee inc(t_i,t_j) \vee lap(t_j,t_i) \vee inc(t_j,t_i) \tag{43}$$
$$inc(t_i,t_k) \wedge seq(t_j,t_k) \Rightarrow inc(t_i,t_j) \vee lap(t_j,t_i) \vee seq(t_j,t_i) \tag{44}$$
$$inc(t_k,t_i) \wedge lap(t_k,t_j) \Rightarrow lap(t_i,t_j) \vee seq(t_i,t_j) \vee inc(t_j,t_i) \tag{45}$$
$$inc(t_k,t_i) \wedge inc(t_k,t_j) \Rightarrow any(t_i,t_j) \vee any(t_j,t_i) \tag{46}$$
$$inc(t_k,t_i) \wedge seq(t_k,t_j) \Rightarrow seq(t_i,t_j) \tag{47}$$
$$seq(t_i,t_k) \wedge lap(t_k,t_j) \Rightarrow seq(t_i,t_j) \tag{48}$$
$$seq(t_i,t_k) \wedge inc(t_k,t_j) \Rightarrow seq(t_i,t_j) \tag{49}$$
$$seq(t_i,t_k) \wedge seq(t_k,t_j) \Rightarrow seq(t_i,t_j) \tag{50}$$
$$seq(t_i,t_k) \wedge lap(t_j,t_k) \Rightarrow lap(t_i,t_j) \vee seq(t_i,t_j) \vee inc(t_j,t_i) \tag{51}$$
$$seq(t_i,t_k) \wedge inc(t_j,t_k) \Rightarrow lap(t_i,t_j) \vee seq(t_i,t_j) \vee inc(t_j,t_i) \tag{52}$$
$$seq(t_i,t_k) \wedge seq(t_j,t_k) \Rightarrow any(t_i,t_j) \vee any(t_j,t_i) \tag{53}$$
$$seq(t_k,t_i) \wedge lap(t_k,t_j) \Rightarrow any(t_j,t_i) \tag{54}$$
$$seq(t_k,t_i) \wedge inc(t_k,t_j) \Rightarrow seq(t_j,t_i) \tag{55}$$
$$seq(t_k,t_i) \wedge seq(t_k,t_j) \Rightarrow any(t_i,t_j) \vee any(t_j,t_i) \tag{56}$$