ANALYZING AND FORMALIZING DEPENDENCIES IN GENERALIZED TRANSACTION STRUCTURES*

Kerstin Schwarz  Can Türker  Gunter Saake

Otto-von-Guericke-Universität Magdeburg
Postfach 4120, 39016 Magdeburg, Germany
E-mail: {schwarz|tuerker|saake}@tt.cs.uni-magdeburg.de

ABSTRACT
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. We analyze different kinds of dependencies and present an extended formal framework (based on ACTA) for describing advanced transaction models. We introduce the notion of transaction closures as a generalization of nested transactions which captures different orthogonal dependencies such as abort and object visibility dependencies, and explicitly show how these dependencies work together. The formal framework is demonstrated by formalizing detached transactions known from the area of active database systems.

1 Introduction
Advanced transaction models are the basis for realizing complex activities in information systems. Traditionally, such models base on the idea of nesting transactions. Nested transactions (Mos85) consist of smaller transactions called child transactions. The initiating transaction is called parent. A child transaction may also consist of child transactions. Depending on the different variations of the nested transaction model, child transactions can be executed sequentially or parallel, and the nesting of the transactions can be closed or open (WS92). However, all transactions have to be terminated before the parent transaction can be committed. This follows immediately from the term ‘nesting’ which means to include something.

In recent years, especially with the introduction of active databases (DHW95), there is an increasing demand for transaction models which provide a more general framework in order to support more flexible transactions realizing complex activities. For instance, detached transactions with various modes are required (Bue94). A detached transaction is initiated by another transaction and is executed as a separate transaction. In comparison to classical child transactions, detached transactions may leave the execution boundaries of the initiating transaction. Depending on the detached mode, the abort of the initiating transaction may not have any influence on a detached transaction.

The DOM transaction model (BÖH+92) goes in this direction by providing different kinds of transactions such as vital, non-vital, contingency, and compensating transactions. A further step towards a more general transactional execution framework are the activity model (DHL91), the ConTract model (WR92), and (transactional) workflow models (RS95). All these models have in common that they support the specification of a certain control flow among transactions.

We present a general formal framework for describing advanced transaction models. For that, we extend the ACTA formalism (CIR94) concerning aspects of activity models. One extension is the notion of transaction closures generalizing nested transactions. A transaction closure contains all transactions initiated (transitively) by one (root) transaction. In contrast to nested transactions, a transaction closure may consist of transactions which terminate after the termination of their parents. In other words, a parent can terminate without waiting for the termination of the child.

There may be different kinds of dependencies among transactions of a transaction closure. We analyze an orthogonal set of dependencies. First, we consider abort dependencies in both directions, from parent to child transactions and vice versa. Furthermore, we investigate how results of transactions are made visible to other transactions. Another interesting issue concerns execution order dependencies being used to describe the control flow among transactions.

The paper is organized as follows: Section 2 defines basic properties of advanced transactions. In Section 3 we analyze abort dependencies among transactions. Section 4 discusses object visibility dependencies whereas Section 5 deals with execution order dependencies. In Section 6 the notion of transaction closures is introduced and different dependencies are combined. The application of our formal framework to detached transactions is demonstrated in Section 7.

2 Foundations
A transaction is an execution unit consisting of a set of operations. A transaction \( t_i \) is started by invoking the transaction management primitive begin \( (b_i) \) and

---

*This research was partially supported by the German State Sachsen-Anhalt under FKZ 1987A/0025 and 1987/2532TR.
1We explicitly do not use the term subtransaction because traditionally subtransactions are executed in the scope of the parent. From our point of view, a parent initiates child transactions without any requirements on the termination of these transactions.
is terminated either by commit \((c_{t_i})\) or abort \((a_{t_i})\). A transaction invokes operations, termed as object events, to access and manipulate the state of database objects. The object event \(p_{t_i}[o]\) refers to the invocation of an operation \(p\) on an object \(o\) by a transaction \(t_i\). The effects of an object event of \(t_i\) are made persistent by using the operation \(c[p_{t_i}[o]]\) and are obliterated by the operation \(a[p_{t_i}[o]]\). The access set \(AS_{t_i}\) contains all operations \(p[o]\) for which transaction \(t_i\) is responsible. The commit of \(t_i\) commits all its operations.

A history of a concurrent execution of a set of transactions \(T\) is a partial order of all events associated with the transactions in \(T\). The complete history \(H\) contains only terminated transactions, the current (incomplete) history is termed as \(H_{cd}\). The occurrence of an event in a history may be constrained as follows:

1. An event \(e'\) may occur only after another event \(e\) in a history \(H\): \(e \rightarrow e'\) is true if \(e\) precedes \(e'\) in \(H\).
2. An event \(e\) may occur only if a certain condition is true: \((e \in H) \Rightarrow Cond_H\) specifies \(e\) can belong to \(H\) only if \(Cond_H\) is satisfied. \(Cond_H\) is a predicate involving the events in \(H\), e.g., \((e' \rightarrow e)\). It can be formulated using well-known logical connectives, e.g., \(\sim, \land, \lor, \Rightarrow, \Leftrightarrow, \exists, \forall\).
3. A condition requires the occurrence of an event: \(Cond_H \Rightarrow (e \in H)\) specifies that \(e\) should be in history \(H\) if \(Cond_H\) holds.

A transaction must be in one of the following states:

\[
\text{active}(t_i) \iff b_{t_i} \in H_{cd} \land c_{t_i} \notin H_{cd} \land a_{t_i} \notin H_{cd}
\]

\[
\text{committed}(t_i) \iff c_{t_i} \in H_{cd}
\]

\[
\text{aborted}(t_i) \iff a_{t_i} \in H_{cd}
\]

The effects of transactions on other transactions are described by dependencies being constraints on possible histories. In the sequel, we recall some fundamental dependencies described in more detail in (CR94). Let \(t_i\) and \(t_j\) be transactions and \(H\) be a finite history:

**Begin Dependency** \((t_i B D t_j)\). Transaction \(t_i\) can only be initiated if \(t_j\) is already initiated:

\[
b_{t_i} \in H \Rightarrow (b_{t_j} \rightarrow b_{t_i})
\]

**Commit Dependency** \((t_i C D t_j)\). If \(t_i\) and \(t_j\) commit, then the commit of \(t_j\) must precede the commit of \(t_i\):

\[
c_{t_i} \in H \Rightarrow (c_{t_j} \rightarrow c_{t_i})
\]

**Abort Dependency** \((t_i A D t_j)\). If \(t_j\) aborts, \(t_i\) has to abort, too: \(a_{t_j} \in H \Rightarrow a_{t_i} \in H\)

**Weak-Abort Dependency** \((t_i W D t_j)\). If \(t_i\) commits and \(t_j\) aborts, the commit of \(t_i\) precedes the abort of \(t_j\):

\[
(a_{t_j} \rightarrow b_{t_i}) \lor (a_{t_i} \rightarrow a_{t_j})
\]

**Serial Dependency** \((t_i S D t_j)\). Transaction \(t_i\) cannot begin executing until \(t_j\) either commits or aborts:

\[
b_{t_i} \in H \Rightarrow ((c_{t_i} \rightarrow b_{t_i}) \lor (a_{t_i} \rightarrow b_{t_j}))
\]

**Begin-on-Abort Dependency** \((t_i B A D t_j)\). Transaction \(t_i\) cannot begin its execution until \(t_j\) aborts:

\[
b_{t_i} \in H \Rightarrow (a_{t_j} \rightarrow b_{t_i})
\]

**Begin-on-Commit Dependency** \((t_i B C D t_j)\). Transaction \(t_i\) cannot begin executing until \(t_j\) commits:

\[
(b_{t_i} \in H) \Rightarrow (c_{t_j} \rightarrow b_{t_i})
\]

Additionally to the dependencies from (CR94) stated above, we introduce the following dependencies:

**Force-Begin-on-Abort Dependency** \((t_i F B A D t_j)\). If \(t_j\) aborts, \(t_i\) has to begin: \(a_{t_j} \in H \Rightarrow (a_{t_j} \rightarrow b_{t_i})\)

**Commit-on-Termination Dependency** \((t_i C T D t_j)\). Transaction \(t_i\) cannot commit until \(t_j\) either commits or aborts: \(c_{t_i} \in H \Rightarrow ((c_{t_i} \rightarrow c_{t_i}) \lor (a_{t_j} \rightarrow c_{t_i}))\)

**Begin-before-Abort Dependency** \((t_i B B A D t_j)\). Transaction \(t_i\) cannot abort until \(t_j\) starts its execution: \(a_{t_i} \in H \Rightarrow (b_{t_j} \rightarrow a_{t_i})\)

**Begin-before-Commit Dependency** \((t_i B B C D t_j)\). Transaction \(t_i\) cannot commit until transaction \(t_j\) starts executing: \(c_{t_i} \in H \Rightarrow (b_{t_j} \rightarrow c_{t_i})\)

Advanced transactions are structured, i.e., they consist of sets of transactions which are interrelated.

**Definition 2.1** The following predicates describe general relationships between a transaction and its creator:

\[
\text{root}(t_j) := \{ t_j \text{ has no parent} \}
\]

\[
\text{parent}(t_i, t_j) := \{ t_i \text{ is parent of } t_j \}
\]

\[
\text{superior}(t_i, t_j) := \{ \text{parent}(t_i, t_j) \lor \text{parent}(t_i, t_j) \}
\]

**Definition 2.2** Suppose \(ST\) denotes the set of structured transactions:

- Each structured transaction has exactly one root:
  \(\forall t_{st} \in ST : \exists t_i \in st_t : \text{root}(t_i)\)

- Each non-root transaction has exactly one parent and the begin of the child must follow the begin of the parent:
  \(\forall t_{st} \in ST : \forall t_j \in st_t : \neg \text{root}(t_j) \Rightarrow \exists t_k \in st_t : \text{parent}(t_i, t_j) \land (t_j B D t_i)\)

- Each transaction structure is acyclic:
  \(\forall t_{st} \in ST : \forall t_i \in st_t : \neg \text{superior}(t_i, t_i)\)

### 3 Abort Dependencies

Fundamental dependencies in transaction structures are abort dependencies which state whether the abortion of a transaction has an influence on another transaction. In the literature, e.g., (CR94), the abort dependencies in nested transactions are well investigated. However, due to the basic assumption of nested transactions that a child transaction may not leave the scope of the parent, the abortion of a parent implies the abortion of all its child transactions. Consequently different abort dependencies are only considered in one direction (from a child to its parent). In contrast, we consider general transaction structures and thus distinguish between different **upwards** and **downwards** abort dependencies, i.e., different possible effects of a transaction abortion on superiors as well as on child transactions.

#### 3.1 Influences of Abortions on Superiors

First, we consider the effects of an abortion of a transaction on its parent and other superiors. We define four kinds of upwards abort dependencies: **vital, weak-vital, weak-non-vital, and non-vital**.

**Definition 3.1 (Vital)** A transaction \(t_j\) is **vital** for another transaction \(t_i\) if \(t_j\) is (transitively) abort as well as commit dependent on \(t_j\)
vital\( (t_i,t_j) \) \( \iff \) \( (\text{parent}(t_i,t_j) \land (t_i AD_t_j) \land (t_i CD_t_j)) \lor \) \( (\exists t_k : \text{vital}(t_i,t_k) \land \text{vital}(t_k,t_j)) \)

Thus, an abortion of a vital transaction \( t_j \) leads to the abortion of all superiors for which \( t_j \) is vital. In other words, a transaction is only allowed to commit, if all its vital child transactions are committed.

Upwards abort dependencies have an influence on the commit of the parent and on the abortion of a child. Table 1 illustrates that a parent \( t_i \) of a vital child \( t_j \) may only commit if \( t_j \) is committed. Due to the fact, that an abortion of a vital child leads to the abortion of the parent, the case, in which \( t_i \) wants to commit and \( t_j \) is already aborted, cannot occur. If \( t_i \) wants to abort, we have to consider the state of \( t_i \). In this case, \( t_i \) cannot be committed until the vital child \( t_j \) is committed.

<table>
<thead>
<tr>
<th>vital( (t_i,t_j) )</th>
<th>( t_i ) wants to commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>active( (t_j) )</td>
<td>( t_i ) must wait for ( t_j )'s termination and may only commit if ( t_j ) commits</td>
</tr>
<tr>
<td>committed( (t_j) )</td>
<td>this case cannot occur</td>
</tr>
<tr>
<td>aborted( (t_j) )</td>
<td>( t_i ) may abort but ( t_i ) must be aborted</td>
</tr>
</tbody>
</table>

Table 1 Vital Dependency

Our second upwards abort dependency bases on the notion of a contingency transaction.

**Definition 3.2 (Contingency)** Transaction \( t_c \) is a contingency transaction of transaction \( t_j \) iff \( t_c \) is semantically equivalent\(^2\) to \( t_j \), \( t_c \) is force-begin-on-abort dependent on \( t_j \), the common parent \( t_i \) of \( t_i \) and \( t_j \) is commit dependent on \( t_c \), and \( t_i \) is abort dependent on \( t_c \) or there exists a contingency transaction \( t_m \) for \( t_c \):

\[
\text{cont}(t_o,t_j) \iff \text{sem-equiv}(t_c,t_j) \land (t_c \land \text{FBAD}_{t_j}) \land \\
(\exists t_i : \text{parent}(t_i,t_j) \land \text{parent}(t_i,t_c) \land (t_i CD_{t_c}) \land \\
((t_i AD_{t_j}) \land (t_i CD_{t_c}) \land (t_i CD_{t_j}))
\]

A contingency transaction is executed as an alternative to transaction \( t_j \) in case \( t_j \) aborts and may have its own contingency transaction. Thus, a transaction \( t_c \) is termed as an alternative to \( t_j \) iff \( t_c \) is (transitively) a contingency transaction of \( t_j \):

\[
\text{alter}(t_c,t_j) \iff \text{cont}(t_c,t_j) \lor \\
(\exists t_k : \text{alter}(t_c,t_k) \land \text{cont}(t_k,t_j))
\]

**Definition 3.3 (Weak-Vital)** A transaction \( t_j \) is weak-vital for another transaction \( t_i \) iff \( t_i \) is (transitively) commit dependent on \( t_j \), begin-before-commit dependent on the contingency transaction \( t_c \) of \( t_j \) in case \( t_j \) aborts\(^3\), and there is an alternative \( t_m \) of \( t_j \) which is vital to terminate the execution of \( t_j \)'s alternatives:

\[
\text{weak-vital}(t_i,t_j) \iff (\text{parent}(t_i,t_j) \land (t_i CD_{t_j}) \land \\
(\exists t_c : (\text{cont}(t_c,t_j) \land \text{abort}_{t_j} \in H) \Rightarrow (t_j \land \text{BCD}_{t_c})) \land \\
(\exists t_m : \text{alter}(t_m,t_j) \land \text{vital}(t_i,t_m)) \lor \\
(\exists t_k : (\text{weak-vital}(t_i,t_k) \land \text{weak-vital}(t_k,t_j)) \lor \\
(\text{weak-vital}(t_i,t_k) \land \text{vital}(t_k,t_j)) \lor \\
(\text{vital}(t_i,t_k) \land \text{weak-vital}(t_k,t_j)))
\]

Table 2 shows the case in which a parent \( t_i \) of a weak-vital transaction \( t_j \) wants to commit. Transaction \( t_i \) may only commit if \( t_j \) is a non-alternative of \( t_j \) commits.

Hence, a weak-vital child can abort without forcing the parent to abort, if there exists an alternative transaction which commits. In case \( t_j \) wants to abort, \( t_j \) may abort but \( t_i \) must be initiated.

<table>
<thead>
<tr>
<th>weak-vital( (t_i,t_j) )</th>
<th>( t_i ) wants to commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>active( (t_i) )</td>
<td>( t_i ) may abort but ( t_i ) must be initiated</td>
</tr>
<tr>
<td>committed( (t_i) )</td>
<td>this case cannot occur</td>
</tr>
</tbody>
</table>
| aborted\( (t_i) \) | \( t_i \) may abort (no effects on \( t_i \))

Table 2 Weak-Vital Dependency

\( t_j \) must commit if \( t_j \) is already terminated. Thus, the case that \( t_i \) is committed and the child \( t_j \) wants to abort cannot occur.

\(^2\)The transaction designer defines which transactions are semantically equivalent (\text{sem-equiv}(t_i,t_j)).

\(^3\)This is necessary to guarantee that the contingency transaction \( t_c \) do not leave the scope of its parent \( t_i \).
The abortion of a non-vital child has no influence on the parent. The parent may continue even if a non-vital child aborts and can commit without waiting for the termination of its non-vital child transactions.

**Definition 3.5 (Non-Vital)** Transaction \( t_j \) is non-vital for another transaction \( t_i \) iff the abortion of \( t_j \) has no effects on the termination of \( t_i \).

### 3.2 Effects of Abortions on Child Transactions

Investigating the abort of a child and the influences on their superiors leads to the question in which way an abort of a superior may effect transactions. We define the dependent, weak-dependent and independent types.

A dependent child aborts if the parent aborts and is only allowed to commit if the parent commits.

**Definition 3.6 (Dependent)** A transaction \( t_j \) is dependent on another transaction \( t_i \) iff \( t_j \) is (transitively) abort dependent on \( t_i \):

\[
\text{dep}(t_i, t_j) : \equiv (\text{parent}(t_i, t_j) \land (t_j \rightarrow D_t_i)) \lor \left( \exists t_k : \text{dep}(t_i, t_k) \land \text{dep}(t_k, t_j) \right)
\]

Table 4 represents a parent \( t_i \) which may abort, if the dependent child \( t_j \) is active or aborted. The case that \( t_j \) is already committed is not valid, because the abort dependency \((t_j \rightarrow D_t_i)\) has to be fulfilled. The child \( t_j \) must wait for the termination of \( t_i \) to commit because the abortion of \( t_i \) leads to the abortion of \( t_j \).

### 4 Object Visibility Dependencies

A second category of dependencies among transactions concerns the visibility of effects of transactions to other transactions. We have again to investigate two directions of object visibility dependencies: (1) visibility of the effects of a transaction on its child transactions and (2) visibility of the effects in the other direction.

#### 4.1 Visibility of Effects on Child Transactions

Before considering different object visibility dependencies, we have to introduce the notion of the view of a transaction.

**Definition 4.1 (View)** The view of a transaction \( t \) (denoted by \( \text{view}_t \)) specifies the state of objects visible to transaction \( t \) at a point in time.

**Definition 4.2 (Inheriting)** A transaction \( t_j \) is an inheriting transaction from the viewpoint of transaction \( t_i \) iff \( t_j \) is (transitively) inheriting the view of \( t_i \):

\[
\text{inher}(t_i, t_j) : \equiv (\text{parent}(t_i, t_j) \land (\text{view}_{t_j} = \text{view}_{t_i} \cup \text{AS}_{t_j})) \lor \left( \exists t_k : \text{inher}(t_i, t_k) \land \text{inher}(t_k, t_j) \right)
\]

The formula above expresses that \( t_j \) sees, additionally to its own access set, everything what \( t_i \) sees. Here we used the term ‘inheriting’ to indicate that a transaction inherits the view of its parent and thus transitively the views of all inheriting superiors. In traditional nested transactions as introduced by Moss (Mos85) all child transactions are inheriting transactions.

**Definition 4.3 (Non-inheriting)** A transaction \( t_j \) is a non-inheriting transaction from the viewpoint of
transaction $t_i$ does not (transitively) inherits the view of $t_j$:
\[
\text{non-inher}(t_i, t_j) \iff (\text{parent}(t_i, t_j) \land (\text{view}_{t_j} = \{q \in \text{obs} \mid \text{committed}) \cup \text{AS}_{t_j}) \lor (\exists t_k : \text{non-inher}(t_i, t_k) \lor \text{non-inher}(t_k, t_j))
\]

A non-inheriting transaction sees after its creation only the effects of committed transactions. For example, detached transactions (Buc94) are non-inheriting.

### 4.2 Treatment of Results of Child Transactions

Now, we consider how the visibility of results of child transactions can be handled. We distinguish two cases:

- The effects are made visible (1) only to the parent or
- (2) to all other transactions. In the first case the effects of a transaction are delegated to the parent.

**Definition 4.4 (Delegate)** A transaction $t_j$ delegates the responsibility for committing or aborting its access set to transaction $t_i$ by using the operation $d_{j}[t_i]$. After $d_{j}[t_i]$ the access set of $t_i$ includes $t_j$'s access set.

**Definition 4.5 (Delegating)** A transaction $t_j$ is a delegating transaction from the viewpoint of transaction $t_i$ if the access set of $t_j$ is (transitively) delegated to $t_i$:
\[
\text{del}(t_i, t_j) \iff (\text{parent}(t_i, t_j) \land (c_{j} \in H \iff d_{j}[t_i] \in H) \land (d_{j}[t_i] \in H \Rightarrow (d_{j}[t_i] \rightarrow c_{i} \lor d_{j}[t_i] \rightarrow a_{i})) \lor (\exists t_k : \text{del}(t_i, t_k) \lor \text{del}(t_k, t_j))
\]

A delegating transaction $t_j$ commits through delegation implying that with the commit of $t_j$, the access set of $t_j$ is made visible only to its parent $t_i$. As a consequence, $t_i$ cannot terminate until $t_j$ delegates its access set.

For performance reasons, often it is necessary that a child makes its effects visible to concurrent transactions and, thus, release all accessed and manipulated objects with the commit. This may happen before the termination of the parent.

**Definition 4.6 (Releasing)** A transaction $t_j$ is a releasing transaction from the viewpoint of transaction $t_i$ if the access set of $t_j$ is visible with the commit of $t_j$:
\[
\text{rel}(t_i, t_j) \iff (\text{parent}(t_i, t_j) \land (\exists o \exists c_{j}[p[ob]] \in H \Rightarrow c_{j} \in H) \land (c_{j} \in H \Rightarrow (\forall p[ob] \in \text{AS}_{t_j} \Rightarrow c_{j}[p[ob]] \in H)) \lor (\exists t_k : \text{rel}(t_i, t_k) \lor \text{rel}(t_k, t_j))
\]

### 5 Execution Order Dependencies

In general, there are execution order dependencies among transactions, e.g., the sequential execution of a contingency transaction after the abortion of another transaction. We distinguish the following execution order dependencies: sequential and parallel.

**Definition 5.1 (Sequential)** A transaction $t_j$ is sequential dependent on another transaction $t_i$ if $t_j$ is (transitively) serial dependent on $t_i$. Analogously, the sequential-abort dependency and the sequential-commit dependency are defined:
\[
\text{seq}(t_i, t_j) \iff (t_j SD t_i) \lor (\exists t_k : \text{seq}(t_i, t_k) \land \text{seq}(t_k, t_j))
\]
\[
\text{seq-a}(t_i, t_j) \iff (t_j BCD t_i) \lor (\exists t_k : \text{seq-a}(t_i, t_k) \land \text{seq}(t_k, t_j))
\]
\[
\text{seq-c}(t_i, t_j) \iff (t_j BCD t_i) \lor (\exists t_k : \text{seq-c}(t_i, t_k) \land \text{seq}(t_k, t_j))
\]

**Definition 5.2 (Parallel)** A transaction $t_i$ is parallel to another transaction $t_j$ if $t_j$ is not sequential dependent on $t_i$. Transaction $t_i$ is parallel-abort dependent on $t_j$ if $t_i$ is begin-before-abort dependent on $t_j$ and parallel-commit dependent on $t_j$ if $t_i$ is begin-before-commit dependent on $t_j$:
\[
\text{par}(t_i, t_j) \iff \neg \text{seq}(t_i, t_j)
\]
\[
\text{par-a}(t_i, t_j) \iff (t_i BCD t_j) \lor (\exists t_k : \text{par-a}(t_i, t_k) \land \text{seq}(t_k, t_j))
\]
\[
\text{par-c}(t_i, t_j) \iff (t_i BCD t_j) \lor (\exists t_k : \text{par-c}(t_i, t_k) \land \text{seq}(t_k, t_j))
\]

### 6 Transaction Closures and Dependency Combinations

A transaction closure is a generalization of a nested transaction. A traditional nested transaction is a transaction closure where the child transactions cannot be independent, i.e., a child transaction cannot leave the execution boundaries of its parent.

**Definition 6.1 (Transaction Closures)** A transaction closure is a structured transaction where each (non-root) transaction is:

- vital, weak-vital, weak-non-vital, or non-vital, and
- dependent, weak-dependent, or independent, and
- inheriting or non-inheriting, and
- releasing or delegating, and
- sequential or parallel.

Theoretically, all introduced kinds of transaction dependencies could be combined to form transactions of a transaction closure. Due to space restrictions, we cannot discuss all possible combinations. However, we can state that some combinations are incompatible or result in dependencies with other properties. To simplify the discussion, we only consider combined dependencies between a child $t_j$ and its parent $t_i$. The presented results are also valid for superiors with the corresponding dependencies.

#### 6.1 Abort Dependencies

The combination of upwards and downwards abort dependencies results in restricted termination event combinations and special execution orders in some cases.

**Theorem 6.1** Combining the vital dependency with the downwards abort dependencies results in an abort and commit dependent parent $t_i$ and differ in the dependency of $t_j$ in case $t_i$ aborts:\footnote{Due to space restrictions, we do not present the proofs of the theorems discussed in this section. However, the theorems immediately follow from the combinations of the definitions given in the previous sections.}
Table 6 Combined Dependencies

<table>
<thead>
<tr>
<th>parent ( t_i )</th>
<th>( a_{t_i} )</th>
<th>( a_{t_j} )</th>
<th>( C_{t_j} )</th>
<th>( C_{t_i} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>child ( t_j )</td>
<td>( a_{t_i} )</td>
<td>( a_{t_j} )</td>
<td>( C_{t_i} )</td>
<td>( C_{t_j} )</td>
</tr>
</tbody>
</table>

- \( \text{vital}(t_i, t_j) \land \text{dep}(t_i, t_j) \Rightarrow (t_i \text{AD} t_j) \land (t_i \text{CD} t_j) \)
- \( \text{vital}(t_i, t_j) \land \text{weak-dep}(t_i, t_j) \Rightarrow (t_i \text{AD} t_j) \land (t_i \text{CD} t_j) \)
- \( \text{vital}(t_i, t_j) \land \text{indep}(t_i, t_j) \Rightarrow (t_i \text{AD} t_j) \land (t_i \text{CD} t_j) \)

Theorem 6.3 The combination of the weak-non-vital dependency with the downwards abort dependencies:
- \( \text{weak-non-vital}(t_i, t_j) \land \text{dep}(t_i, t_j) \Rightarrow (t_i \text{CD} t_j) \land (t_j \text{AD} t_i) \)
- \( \text{weak-non-vital}(t_i, t_j) \land \text{weak-dep}(t_i, t_j) \Rightarrow (t_i \text{CD} t_j) \land (t_j \text{WD} t_i) \)
- \( \text{weak-non-vital}(t_i, t_j) \land \text{indep}(t_i, t_j) \Rightarrow (t_i \text{CD} t_j) \land \text{true} \)

6.2 Object Visibility Dependencies

Abort dependencies are constraints on the occurrence of the termination events of related transactions. In contrast, object visibility dependencies have an influence on the view of the related transactions:

Combining delegating and inheriting: The view of a child \( t_j \) includes its access set and the view of the parent \( t_i \) which consists of the commit projection of the current history \( H_{\text{c}} = \{ \text{commit}(t_i) \} \) and the access set of \( t_i \) \( (\text{view}_{t_i} = H_{\text{c}} \cup \text{AS}_{t_i}) \). Thus, \( t_j \) can see the results of \( t_i \) and delegates its own access set with the commit to \( t_i \). In consequence, the access set and, thus, the view of the parent \( t_i \) is extended by the access set of the committed child \( t_j \). However, these results are not visible to (other) concurrent transactions.

---

\( \text{tern \_ termination events \_ combinations which \_ are \_ valid \_ in \_ both} \) 
\( \text{execution \_ orders \_ are \_ denoted \_ as \_ true} \) 
\( \text{and \_ invalid \_ termination \_ event} \) 
\( \text{combinations \_ are \_ denoted \_ as \_ false} \). 
\( \text{There \_ are \_ termination \_ event} \) 
\( \text{combinations \_ which \_ are \_ only \_ valid \_ if \_ they \_ are \_ executed \_ in \_ a \_ special} \) 
\( \text{order. \_ In \_ this \_ case \_ the \_ corresponding \_ execution \_ order \_ is \_ noted} \).
Combining releasing and inheriting: The child \( t_j \) inherits the view of its parent \( t_i \). Due to the releasing property, the results of the child \( t_j \) are immediately reflected in the commit projection of the current history with the commit of \( t_j \). Thus, the view of the parent \( t_i \) is the same as in the case of a delegating child. However, the access set of \( t_i \) is not extended by the results of \( t_j \).

Combining delegating and non-inheriting: A non-inheriting dependency reduces the view of a child \( t_j \) to the commit projection of the current history and its own access set. In other words, a child does not see the effects of its parent. A combination with the delegating property leads to an extension of the view of the parent \( t_i \) by delegating \( t_j \)'s access set with its commit.

Combining non-inheriting and releasing: This combined dependency is equivalent to a dependency between two concurrent transactions of different transaction closures. Both transactions see only their own access set in addition to the commit projection of the current history, i.e., they do not see the effects of each other since the effects are committed.

<table>
<thead>
<tr>
<th>( \text{inher}(t_i, t_j) \land \text{del}(t_i, t_j) )</th>
<th>( \text{view}<em>{t_i} ) (at ( b</em>{t_i} ))</th>
<th>( \text{view}<em>{t_j} ) (at ( c</em>{t_j} ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>( H_{t_i} \cup \text{AS}_{t_i} )</td>
<td>( H_{t_j} \cup \text{AS}<em>{t_j} \cup \text{AS}</em>{t_j} )</td>
<td></td>
</tr>
<tr>
<td>( \text{inher}(t_i, t_j) \land \text{rel}(t_i, t_j) )</td>
<td>( H_{t_i} \cup \text{AS}_{t_i} )</td>
<td>( H_{t_j} \cup \text{AS}_{t_j} )</td>
</tr>
<tr>
<td>( \text{non-inher}(t_i, t_j) \land \text{del}(t_i, t_j) )</td>
<td>( H_{t_i} )</td>
<td>( H_{t_j} \cup \text{AS}<em>{t_j} \cup \text{AS}</em>{t_j} )</td>
</tr>
<tr>
<td>( \text{non-inher}(t_i, t_j) \land \text{rel}(t_i, t_j) )</td>
<td>( H_{t_i} )</td>
<td>( H_{t_j} \cup \text{AS}_{t_j} )</td>
</tr>
</tbody>
</table>

Table 7 Combined Object Visibility Dependencies

In Table 7 we compare the view of the child \( t_j \) at the starting time of \( t_j \) and the view of the parent \( t_i \) at the commit time of \( t_j \) for all combinations discussed above. Due to the consideration of the views of the related transactions at different points in time, the commit projection of the current history \( H_{t_i} \) may be changed, e.g., with the commit of a releasing transaction.

6.3 Object Visibility and Abort Dependencies

The object visibility dependencies have also an influence on the abort dependencies. Concerning the delegating property, the delegation of the access set of \( t_j \) to its parent \( t_i \) has to precede the termination of \( t_i \). A combination with the independent property restricts the independent property to a weak-dependent property because the requirement that the commit of \( t_j \) must precede the abortion of \( t_i \) is added by the delegating dependency.

A combination of a non-vital dependency with the delegating property, converts the former one to a weaknon-vital dependency. This is due to the fact that the commit of transaction \( t_j \) has to precede the commit of its parent \( t_i \). The abortion of \( t_j \) also has to precede the commit of \( t_i \) because the parent has to wait for the termination decision of the child to commit.

Theorem 6.5 Combining independent and non-vital with the delegating dependency leads to a transformation (\( \Rightarrow \)) to a weak-dependent and weak-non-vital dependency, respectively:

\[
\begin{align*}
\text{indep}(t_i, t_j) \land \text{del}(t_i, t_j) & \Rightarrow \text{weak-dep}(t_i, t_j) \\
\text{non-vital}(t_i, t_j) \land \text{del}(t_i, t_j) & \Rightarrow \text{weak-non-vital}(t_i, t_j)
\end{align*}
\]

Because of space restrictions, we only consider the combination of the delegating dependency with non-vital and the downwards abort dependencies (Table 6). We can easily see that the delegating dependency constrain all valid termination event combinations by adding that the child has to terminate before the parent.

In contrast, a releasing dependency requires no restriction on the termination order of the related transactions. Therefore, in this case, there are no additional constraints on the abort dependencies. The inheriting and non-inheriting properties are related to the begin of a child. Thus, these dependencies have also no influence on the abort dependencies of the related transactions.

6.4 Execution Order Dependencies

Execution order dependencies can be combined with the abort and object visibility dependencies. In general, parent and child transactions are executed in parallel. However, in advanced applications, e.g., in active databases, a transaction may be triggered within another transaction and executed sequentially after the termination of the triggering transaction.

Assume that a child \( t_j \) has to be executed sequentially after the termination of the parent \( t_i \) (seq\((t_i, t_j)\)):

1. In case \( t_j \) has to be started after the commit of \( t_i \) (seq-c\((t_i, t_j)\)), the dependency between these two transactions should be non-vital. In case of the vital dependency, the abortion of the child may force the parent to abort, too. However, in combination with a sequential dependency, this is not reasonable because the parent terminates before the child begins executing. As shown in Table 6, the vital, weak-vital, and weak-non-vital dependency requires that the child \( t_j \) commits before the parent \( t_i \) can commit. However, this contradicts the requirement of the sequential dependency that \( t_j \) has to start after the termination of \( t_i \). Thus, this combinations are incompatible.

2. The same argument holds also for the case that the child \( t_j \) has to be started after the abortion of the parent \( t_i \) (seq-a\((t_i, t_j)\)). Here, the downwards abort dependency should be independent. According to Table 6, a weak-dependent transaction \( t_j \) requires that its commit precedes the abortion of its parent \( t_i \). However, this also contradicts the requirement of the sequential dependency that \( t_j \) has to start after the termination of \( t_i \). In case of dependent transactions, the case that \( t_i \) aborts and \( t_j \) commits is disallowed. Thus, the execution of \( t_j \) after the abortion of \( t_i \) cannot lead to a commit of \( t_j \). However, the abortion of \( t_j \) is valid. Thus, the execution of transaction \( t_j \) makes no sense because it is aborted in any case.

Table 6 represents the combination of the sequential dependency with vital and the downwards abort dependencies. Analogously to the weak-vital and weak-non-vital dependency, a combination of vital and sequential leads to contradictions in the execution ordering requirements. Thus, the last column is completely adapted to false where in the second column only the combination with weak-dependent is changed to false.

The delegating dependency requires the delegation of the results of \( t_j \) to the parent \( t_i \). From this follows,
a sequential execution of \( t_j \) after the termination of \( t_i \) is impossible (analogously to the abort dependencies).

Finally, we can state that a sequential dependency between two transactions requires (\( \sqcap \)) that these transactions are also non-vital, independent, and releasing:

\[
\text{seq}(t_i, t_j) \supset \text{non-vital}(t_i, t_j) \land \text{indep}(t_i, t_j) \land \text{rel}(t_i, t_j)
\]

The combination of a sequential dependency with the other abort and object visibility dependencies are either not reasonable or (logically) impossible.

7 Exemplary Application of the Framework

Using the notion of transaction closures we are now able to specify advanced transaction models as well as detached transactions. In the literature (for a survey see (Elm92)), several advanced transaction models are proposed which base on the nested transaction model (Mos85). For instance, closed nested transactions are transaction closures which consist of transactions which are vital, dependent, inheriting, and delegating from the viewpoint of their parents. In the following, we concentrate on detached transactions.

The notion of detached transactions comes from the area of active databases where triggered actions are executed as separate transactions out of the scope of the triggering transaction. We will focus on the four detached modes described in (Bu94).

In our framework, the triggering transaction is the parent \( t_i \) of a triggered child \( t_j \) with special dependencies. Here, we have to consider two dimensions of dependencies: the execution order dependency and the commit dependency.

Detached transactions are executed as separate transactions. From this follows that a detached transaction is a non-inheriting and releasing transaction. It has its own scope and does not see the effects of its parents and other superiors until they are committed. A further implication is that detached transactions are non-vital, i.e., the abortion of a detached transaction has no influence on the parent and the transaction can leave the execution boundaries of the parent. The other dependencies like the downwards abort dependency or the execution order dependency differ in the four cases of detached transactions:

(1) A detached independent transaction is executed in parallel to the parent and is independent of the termination of the parent:

\[
\text{non-inher}(t_i, t_j) \land \text{rel}(t_i, t_j) \land \text{non-vital}(t_i, t_j) \land \\
\text{par}(t_i, t_j) \land \text{indep}(t_i, t_j)
\]

(2) A detached parallel causally dependent transaction is executed parallel to the parent transaction, but it may only commit if the parent commits. Thus, a detached parallel causally dependent transaction is abort dependent on the parent:

\[
\text{non-inher}(t_i, t_j) \land \text{rel}(t_i, t_j) \land \text{non-vital}(t_i, t_j) \land \\
\text{par}(t_i, t_j) \land \text{dep}(t_i, t_j)
\]

(3) A detached sequential causally dependent transaction is only executed if the parent transaction commits. Hence, there exists no abort dependency between a detached transaction and its parent:

\[
\text{non-inher}(t_i, t_j) \land \text{rel}(t_i, t_j) \land \text{non-vital}(t_i, t_j) \land \\
\text{seq-c}(t_i, t_j) \land \text{indep}(t_i, t_j)
\]

(4) A detached exclusive but causally dependent transaction which is executed sequential after an abortion of the parent has to be independent of the parent’s abortion. This type is called exclusive because the detached transaction may only commit if the parent aborts:

\[
\text{non-inher}(t_i, t_j) \land \text{rel}(t_i, t_j) \land \text{non-vital}(t_i, t_j) \land \\
\text{seq-a}(t_i, t_j) \land \text{indep}(t_i, t_j)
\]

In conclusion, we have shown that the proposed extended framework allows an easy and elegant formalization of non-traditional transaction models which go far beyond the nested transaction model.

8 Conclusions and Outlook

In this paper, we presented a generalized framework for describing and classifying dependent transactions. The concept of transaction closures extends the concept of nested transaction for example allowing detached child transactions in such transaction closures. For formally describing transaction closures, we extended the ACTA framework. The resulting framework gives a richer repertoire for classifying dependency relations between transactions inside a transaction closure. These extensions may become relevant for example in the application areas of active databases and database federations, where classical nested transactions are too restrictive because a child of a nested transaction tree has to terminate in the scope of its parent transaction.

Our future work will concentrate on dependency enforcement in transaction closures. We will attempt to adopt the methods proposed in (GHK93) to our framework and provide some extensions to capture the transitive properties of the transaction dependencies.

REFERENCES


H. Wächter and A. Reuter. The ConTract Model. In (Elm92), pp. 219–263.