Serializability in Object-Oriented Database Systems

Thomas C. Rakow, Junzhong Gu*, Erich J. Neuhold

GMD
Integrated Publication and Information Systems Institute (IPSI)
Dolivostraße 15, D-6100 Darmstadt, West Germany
e-mail: rakow@darmstadt.gmd.dbp.de

Abstract
In an object-oriented database the objects are encapsulated, i.e., objects are only accessible by methods defined in the database system. Our definition of object-oriented serializability takes advantage of the semantics and of the nesting of the methods. Therefore, a lower rate of conflicting accesses than with the conventional definition of serializability is achieved.

Transactions of an object-oriented database are defined as open nested transactions. Depending on the semantics of operations actions can be serialized independently from the calling transactions. The techniques already used in multi-layer transaction systems are extended to object-oriented systems. Object-oriented serializability includes multi-layer serializability but allows a non-layered, more general structure of the database system.

Key words: object-oriented database system, open nested transaction, object-oriented serializability, semantic concurrency control.

1. Introduction

Conventional database systems are suitable for managing simple objects like account values or employee numbers. The management of more complex objects like electronic chips or office documents however is supported very insufficiently. This deficiency has led to intensive research and development in the field of database systems for such non-standard applications.

At the Integrated Publication and Information Systems Institute (IPSI) of the German national research institution GMD a non-standard database management system called VODAK is being developed. VODAK combines research from the areas of knowledge representation languages and of database systems [13]. The VODAK modeling language offers

- an object-oriented data model, which encapsulates objects together with their operations (methods), and supports inheritance of structure, operations and values,
- concepts for modeling complex structured objects,
- an extensible, typed, uniform data definition, data manipulation and programming language (under development).

A description of the model is given in [6], a theoretical framework is offering [5].

The transaction concept is an essential feature of a database management system (DBMS). A transaction is the execution of a program accessing a database. The DBMS allows the programmer to assume that each transaction executes atomically – as if no other programs execute concurrently – and reliably – as if there were no failures [2]. This paper describes how to guarantee atomicity in an object-oriented database system.

Concurrent execution of transactions may cause inconsistencies like lost updates, inconsistent reads, and occurrences of phantoms. The conventional method to guarantee the consistency of objects is conflict order preserving serializability. An often used protocol to guarantee serializability is two phase locking [2]. But complex objects may conform to large parts of an object-oriented database. Locking the whole object for the possibly long time a transaction may last is not acceptable as it would reduce concurrency significantly [8, 20].

A concurrency control protocol must balance more concurrency against additional costs for managing the concurrency. A relatively high degree – compared to the maximal possible degree – of concurrency is necessary for information and publication systems. For example, consider a publication system which allows the cooperative editing of documents by several authors (like this paper). Every author wants to write down his ideas immediately. But if another author edits the document simultaneously he must wait until the document is released (and perhaps the idea has flown away). If a system ensures that all authors see a consistent view, concurrent work is possible. On the other hand relatively high costs – compared to conventional transaction systems – of concurrency control will be acceptable. Compared to operations in high performance transaction systems like an airline database of flight bookings, editing is a slow operation.

Figure 1 shows the differences between conventional transactions and object-oriented operations with examples out of two typical application areas: financial markets and publication environments.

<table>
<thead>
<tr>
<th>conventional transactions</th>
<th>object-oriented operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>access to small objects</td>
<td>access to large and complex structured objects</td>
</tr>
<tr>
<td>(an account)</td>
<td>(a document)</td>
</tr>
<tr>
<td>short duration (milliseconds ... seconds)</td>
<td>long duration (seconds ... months)</td>
</tr>
<tr>
<td>simple actions (writing an account)</td>
<td>complex structured actions (processing the layout of a document consists of processing the contents, the chapters, ...)</td>
</tr>
</tbody>
</table>

* Now: Department of Computer Science, East China Normal University, Shanghai, People's Republic of China

Fig. 1: Differences between conventional transactions and object-oriented operations (with examples)
The crucial question is: Are the object-oriented operations also transactions? This paper proposes a treatment of the problem by defining object-oriented transactions and object-oriented serializability. Here we utilize

- the semantics of operations, i.e., the knowledge whether operations are in conflict or not, and
- the nesting of operations caused by the call of operations.

Transactions according to these principles are called open nested transactions [19]. Open nested transactions have already been applied to layered transaction systems [1, 3, 11, 23, 24]. In a layered system the transactions are implemented by actions at the underlying level of specialization. But an object-oriented system is a generalization of a layered system [3] when objects are considered as layers. The operations on an object represent the level of actions, the calling actions represent the level of transactions. The execution of transactions leads to a non-layered, more general structure of operations on objects.

Our approach applies a good old engineering rule: divide et impera! We classify the knowledge of maintaining serializability in a transaction system:

1. the information which is recorded at a single object,
2. the information which flows from one object to another and
3. the information which relates to several objects equally.

First of all, we consider single objects. Seen from the point of view of an object the nested structure of object-oriented operations is mapped to a flat two-level structure: operations accessing the object and calling operations. Usually an operation accessing an object is called action. Consider a notation which may be surprising at first look: A calling action plays its part as a transaction. Hence, it makes no difference whether a transaction calls an action directly or indirectly. A transaction depends on another transaction if they both access the same object and the accessing actions are in conflict. Two actions are in conflict if the effect of the first depends on whether the second was executed before or after the first. Using these semantics of operations an object can record a transaction dependency relation. Then, an object schedule - i.e., the interleaved execution of transactions seen from an object - is object-oriented serializable if the transaction dependency relation is the same as for a serial schedule.

Object-oriented serializability (oo-serializability) of an object schedule assumes that a schedule of actions is given, i.e., the knowledge of an execution order of actions. Actions on an object \( O \) may be transactions on another object \( P \), which is called a transaction on \( P \). We must not allow different views of the objects \( O \) and \( P \) for the same situation. The information about the transaction dependencies of the object \( P \) must flow to the object \( O \). Therefore, the transaction dependencies of the object \( P \) results in action dependencies of the object \( O \). Actions which are not transactions on any other object must fulfill a bootstrap condition: their dependencies must be given. Because actions on an object may be transactions on two or more other objects contradicting action dependencies are possible. Contradicting dependencies signify that actions have accessed an inconsistent state. Additionally to the condition for the transaction dependency relation, contradicting action dependencies are not allowed in an oo-serializable object schedule.\(^1\)

With the information flow between objects some information can be lost. There may exist transaction dependencies for two transactions on two objects but no common object for which both transactions are actions. This information may be recorded at a central instance, by one selected object or by both objects redundantly. We record the information at both objects. To the action dependency relation of an object we add an added action dependency relation. Then, the schedule of a transaction system is oo-serializable if all object schedules are oo-serializable and the added action dependency relation contains no contradictions.

The rest of the paper is organized as follows. Section 2 motivates our approach with an object-oriented application, considers related work and our approach. In section 3 the system model is defined, especially the object-oriented transactions. In section 4 the notion of schedule of an object and its properties are defined. An object schedule can be conform, serial and oo-serializable. Section 5 then contains the definition of oo-serializability for the schedule of a system. The definition is based on oo-serializability of an object schedule. Finally in section 6, we summarize and discuss our approach and consider future research.

2. Motivation

Before we motivate our approach by means of an object-oriented application we introduce the necessary terminology [2]. An interleaved execution of transactions is called a schedule. The DBMS prevents inconsistencies by allowing only schedules which are equivalent to a serial execution. Such schedules are called serializable. Two schedules are equivalent if there is no difference in their results. Two actions are in conflict if a different access order is determinable by these two actions or by another action. If two actions are not in conflict then they commute. An action, which is executed after another action and is in conflict, depends on the first action. Only if all dependencies of a transaction are executed in the same sequence is a schedule serializable. This property of a schedule is called conflict order preserving serializability. The transactions are said to be isolated from each other.

As an application we choose a – simply structured – encyclopedia with often changing items (for example, an encyclopedia which is used within the process of text analysis as a "world knowledge" base). The items of the encyclopedia are indexed by a B+ tree [15]. The encyclopedia named Enc consists of a linked list of

1. Because action dependencies of an object may be contradictory we denote the action dependencies as a relation and not as an order.
items named LinkedList and a B+ tree named BpTree (Figure 2). The keys of the items are indexed by BpTree. The data are stored on pages (only shown for the leaves and the items).

In conventional DBMSs operations on index structures like B+ trees are executed with special concurrency control protocols. For restructuring purposes separate independent transactions are started. The index manager of a DBMS knows the semantics of the whole index structures. Therefore, the DBMS can manage concurrency control efficiently. In an object-oriented DBMS applications may be complex similar to index structures. How can the knowledge about the semantics be transferred to the DBMS?

Objects are encapsulated. The implementor of an object type only knows his "world", i.e., the structure of objects, its operations and which messages are sent to other objects. He knows nothing about the implementation of these messages. But he can specify the semantics of the implemented object type. We will show that the DBMS can connect the specified semantics of different object types in one framework.

We use the principle of open nested transactions [3, 19, 20, 21]. A nested transaction is a calling tree of actions called subtransactions. The root is called top-level-transaction. By the use of conventional transactions and closed nested transactions [12] only top-level-transactions are isolated from each other. Subtransactions of open nested transactions are isolated against other subtransactions. A subtransaction may not consider dependencies of conflicting operations of underlying subtransactions relying on its semantics.

The semantics that is used for the determination of dependencies is the commutativity of operations [10]. Weihl [22] and Spector and Schwartz [18] explained how to get commuting operations on complex abstract data types (e.g., queues or directories). The escrow method [9, 14, 17] includes parameter values and the status of accessed objects in the commutativity definition. We will show examples of commuting operations but firstly we explain how they are used in multi-layer transaction systems.

In a multi-layer transaction systems [1, 3, 11, 23, 24] the transactions are implemented by actions at the underlying level of specialization. The process stops at a zero-layer, where the actions call no other actions. The concurrency control component of these systems considers two adjacent layers in one schedule. The operations of the higher layer are the transactions, the operations called at the lower layer are the actions. The order of conflicting actions is inherited upward to their corresponding transactions. Hence, the transactions are called quasi-ordered [23, 24]. If they are also in conflict, their order is further inherited upward, otherwise the inheritance stops. Now, let us have a look at the encyclopedia (Example 1).
Example 1

Figure 3 contains the legend for the following figures. Figure 4 shows three transactions, sent to the encyclopedia object Enc, and their operations on the B+ tree BpTree. T1 is sent to Enc. It calls insert(DBMS)11 on BpTree, noted as BpTree insert(DBMS)11. This call results in calls on nodes (not shown) and a call on a leaf Leaf1 insert(DBS)11. The insert calls a read and a write on Page4712. The other transactions are drawn in Figure 4 analogously to T1.

Assume, Page4712.write11 is executed before Page4712.read11. In the figure this dependency is noted by the dashed directed arc between the two actions, the conflict is labelled with 0. Using conventional serializability all the other conflicting actions of both transactions must preserve the order that is occurring in this dependency. But this demand is too restrictive. Semantically, Leaf11 insert(DBS)12 and Leaf11 insert(DBMS)12 are not in conflict. They insert different keys. Unfortunately these keys are stored on the same page. Every node and therefore the corresponding page contains many keys (rough up to 500). Operations on these keys will often conflict at the page level but commute at the node level.

The dependency must be remembered by the calling subtransactions to prevent lost updates on Page4712. The dependency is shown by the dashed directed arc at Leaf1 which points in the same direction as the arc at Page4712. The actions on Leaf1 act as transactions for the actions on page4712. The dependency is no longer relevant after lost updates are prevented, i.e., after the end of Leaf11 insert(DBS)11p. Because the calling actions commute (the @ at the dashed line), it makes no difference in which order the two inserts are written on Page4712. But the dependency must be noted because all other actions called by Leaf11 insert(DBS)11 and Leaf11 insert(DBMS)11 must serve the order that is occurring in this dependency. But the dependency must be inherited to the calling actions on BpTree. The dependency can be neglected at BpTree and at Enc. Other calls of the transactions BpTree insert(DBS)12 and of T2 need not preserve all the previous dependencies. Hence, more concurrency is possible.

Now let us have a look at T3 and T4. Assume, Page4712.write11 is executed before Page4712.read11. Hence, Page4712.read11 depends on Page4712.write11. The dependencies are noted as above. In contrast to the previous case the calling actions on Leaf1 are in conflict now. The actions Leaf11 insert(DBS)11 and Leaf11 search(DBMS)11 access the same key DBS. Therefore, the order of the two actions on Leaf11 must be preserved. The conflict is noted by dashed directed arc. The dependency is inherited to the calling actions on BpTree. On BpTree there is a conflict, too. Therefore, the dependency is inherited to T3 and T4.

End of Example 1

An object-oriented transaction system is a generalization of a layered system [3] when objects are considered as layers.

- The call depths of transactions are different. Levels do not exist. It is not predetermined at which level the call hierarchy of a transaction ends. Nevertheless, in database systems exists a common object type which methods call no other actions: the page.
- Transactions on an object can call actions on every object directly. For example, an item can be reached via the object LinkedIn and via the object BpTree (Figure 2). The numbers of levels on the two access paths are unequal.
- A transaction can call an action (directly or indirectly) and both can access the same object. Consider an insert of a key in a B+ tree leaf. If there is not enough space for the new key, the leaf must be split into two leaves. To enhance concurrency, lock coupling and B-linking is used [15]. A lock is only set on the leaf. After the split is executed but before this information is given to the father node, consistency is preserved by pointing to the new leaf from the old
leaf (so-called B-link). After the split is completed the lock is released. Therefore, the rearrangement of the father(s) may be implemented as a single subtransaction, called from the insert subtransaction. Consider the following schedule where the object \( Node6 \) (Figure 2) is accessed twice:

\[
\text{Node6.insert()} \rightarrow \text{Leaf11.insert()} \rightarrow \text{Leaf12.insert()} \rightarrow \text{Node6.rearrange()} ...
\]

\( \rightarrow \) means: calls with change of control between the objects,

\( \rightarrow \) means: calls a set of actions.

3. Object-Oriented Transaction System

In this section we define object-oriented transactions and a system consisting of such transactions. A transaction is the execution of a program accessing a database. It consists of messages which are sent to objects of the database. At the object the method (operation) which is specified by the message is executed. In turn the method may call other methods, which are sent as messages to other objects. We are only interested in executed methods, i.e., messages interacting with the database. In this paper we only speak about messages and say that a message calls another message.

Definition 1

A message \( m \) on an object \( O \) is defined as parameterized method of \( O \) sent to \( O \), denoted by \( O.m \) (parameters).

If the parameters are of no relevance we write \( O.m \). If it is clear, which object is accessed we simply write \( m \). We say that \( m \) accesses \( O \). If a message \( m \) calls another message \( m' \), we denote it with

\[ m \rightarrow m' \]

The transitive closure is denoted with \( m^* \). If the calling message \( m \) itself is included in the closure we denote it with \( m^+ \).

Messages relevant to the concurrency control are messages which may be interleaved with other messages. The interleaving may cause inconsistencies. For example, we must consider the access of an object \( O \). As usual the directly or indirectly called messages are hierarchically numbered and are called actions. They are grouped into sets. Each set contains all the actions called by the set defining action directly. According to the algorithm within an action set some called actions may be executed in parallel. The order is specified in a precedence relation between elements of one action set.

Definition 2

Let \( O.A_i \) be an action on the object \( O \), i.e., a numbered message on \( O \). The object-oriented transaction \( O.A_i \) consists of

- the action \( O.A_i \) itself,
- the set \( A_{O_i} \) of actions of the form

\[ A_w = A_{O_i} \cup A_{O_{w1}} \cup A_{O_{w2}} \cup \ldots \]

and

- for all \( A_w \), a partial order \( A_w \subseteq A_{O_i} \), called \( (O \text{-transaction}) \) precedence relation.

3. The decision which messages may be interleaved with others is crucial. It is part of the development of objects and its methods (e.g., see concurrent search structure algorithms [15]).

4. \( a \rightarrow b \Rightarrow \exists \; c \rightarrow a \).

An oo-transaction forms a tree. The nodes represent the actions, and the arcs represent the call relationship. The root is the originating action of the oo-transaction, the leaves are so called primitive actions. There is no possibility to interleave primitive actions with other actions.

Definition 3

An action \( O.A \) is called primitive, if it calls no other action. The set of primitive actions on an object \( O \) is denoted with \( P(O) \).

Example 2

Figure 5 shows the tree of an oo-transaction \( t \). The root is the action \( t_1 \), the leaves are \( a_{11}, a_{12}, a_{13}, a_{11}, \) and \( a_{12}. \) The action set \( A_{11} \), for example, consists of the two actions called by \( a_{11} \) and \( a_{12}. \) The precedence of actions is identified by the left to right order of arcs (e.g., \( a_{11} \rightarrow a_{12} \)).

We now define an object-oriented transaction system. It consists of a homogeneous set of objects. The objects are uniquely identified by an object identifier. There is a special object which is called the system object. All the oo-transactions are messages on the system object, i.e., they belong to the same system.

In addition the system consists of a set of oo-transactions. To distinguish oo-transactions on the system object from the other oo-transactions we call them top-level-transactions. Top-level-transactions are the working units as specified by the application programmer. Hence, top-level-transactions executed serially preserve the consistency of the database.

Definition 4

An object-oriented transaction system \( TS \) consists of

- a set \( OBJ \) of objects with a system object \( S \in OBJ \)
- a set \( TOP \) of oo-transactions on the system object \( S \), called top-level-transactions.

Top-level-transactions are noted with \( T, T_1, \ldots \).

If a transaction \( r \) calls an action \( a \) directly or indirectly and both access the same object \( O \), this situation results in a cycle of the call path. Because we want to distinguish between actions and transactions of an object we extend the system by constructing a virtual object to break the cycle. The transaction \( r \) accesses the original object \( O \), the action \( a \) accesses the virtual object \( O' \). But the conflicts of \( r \) and \( a \) may not disappear because of this plot. Therefore, all other actions on the object \( O \) are "virtually" duplicated, too. These virtual actions are connected to their originals with
the call relationship. The dependencies on the virtual object are inherited along these call relationships to the original object (as defined in Definition 10).

From now on we assume that an oo-transaction system is extended according to Definition 5.

Definition 5
Let \( TS \) = (OBJ, TOP) be an oo-transaction system. The set \( ACTO \) of actions on an object \( O \in OBJ \) is defined as:
\[
ACTO := \{ a_1 O_1 a, a \rightarrow T, T \in TOP \}
\]
Let \( a, b, t, u \in ACTO \). The extent of \( TS \) consists of the system \( TS \), but for every action \( a \) which fulfills the condition
\[
O_1 \rightarrow O_2 a
\]
and for which \( u \in u \) exists such that
\[
O_1 \rightarrow O_2 a
\]
- an object \( O \) with \( ACTO' := ACTO - \{ t \} \) is added,
- \( ACTO \) is defined as \( ACTO := ACTO - \{ a \} \), and
- for all \( b \in ACTO, b \in ACTO' \), \( b \in b' \) the call relationship \( b \rightarrow b' \) is added.

Example 3
In transaction \( t_1 \) (Figure 5) the action \( a_{11} \) calls the action \( a_{12} \), and both access the object \( O_1 \). Breaking this cycle by extending the system results in a call from \( a_{12} \) to \( O_1 t_1 \) to \( a_{12} \) on \( O_1 \). An action \( b_{22} \), for example, which accesses \( O_2 \) is virtually duplicated by \( b_{22}' \) on the object \( O_2' \). The action \( b_{22} \) is called by \( b_{22}' \) (Figure 6).

4. OO-Serializability of an Object Schedule

In this section we define an object schedule and three of its properties. An object schedule can be conform, serial, and oo-serializable. A schedule is conform if it preserves (algorithmically) given precedences. A schedule is serial if the top-level-transactions are executed serial. Object-oriented serializability (oo-serializability) combines conventional serializability with the object-oriented system approach. As usual a schedule is called serializable if there exists an equivalent serial schedule. But in contrast to conventional systems we consider the point of view of an object in the definition of an object schedule.

We define an object schedule of an Object \( O \) not only for the actions on the object \( O \). The object must know which actions in respect to which transactions must be serialized. In contrast to a conventional schedule this knowledge is not obvious. In a schedule we consider action and transaction dependencies. The transactions are only known in the system to which the object belongs. Therefore, an object schedule consists of a system, an object, an action dependency relation, and a transaction dependency relation.

Definition 6
Let \( TS = (OBJ, TOP) \) be an oo-transaction system. The set \( TRA_o \) of actions, which call an action on the object \( O \in OBJ \), is defined as:
\[
TRA_o := \{ t \rightarrow T \rightarrow a, T \in TOP, a \in ACTO \}
\]
Actions of \( TRA_o \) are called transactions on \( O \).
An object schedule \( Sch \) of an object \( O \) in \( TS \) consists of:
- the transaction system \( TS \) = (OBJ, TOP),
- the object \( O \in OBJ \),
- an action dependency relation \( \subseteq ACTO \times ACTO \) and
- a transaction dependency relation \( \subseteq TRA_o \times TRA_o \).

Transactions are noted with \( t, t', u, ..., \) actions with \( a, a', b, ... \)
An object schedule is conform if there exist no inconsistencies from intra-transaction parallelism. The definition must "look" a bit deeper than in the conventional case and observe inherited precedences. On the one hand the action dependency relation of a schedule must include the given precedences \( b_{ij} \) for all \( i \) and \( j \). But on the other hand the actions must follow the precedence given for their calling transactions as well. In the object schedule \( Sch = (TS, O_1, \ldots, \cdot) \) (Figure 5) the action \( a_{12} \) must precede \( a_{12} \) because \( a_{12} a_{12} \).

Definition 7
Let the object precedence relation \( n_3 \subseteq ACTO \times ACTO \) be defined as:
\[
a_3 a': \exists \exists t \rightarrow T \rightarrow a, T \rightarrow t \rightarrow a', T, T' \in TOP: T, T' \text{ for } an \ A_i
\]
An object schedule \( Sch = (TS, O, \cdot, \cdot) \) is conform, if \( n_3 \subseteq \cdot \).

Thus we assume that object schedules are conform.
An object schedule is serial if the top-level-transactions are not interleaved. This property assumes (1) totality of the action order. (2) For every transaction pair all the actions of the first transaction precede all the actions of the second transaction or vice versa. This definition is the same as for conventional systems.

Definition 8
An object schedule \( Sch = (TS, O, \cdot, \cdot) \) is serial if
(i) \( \cdot \) is total.
(ii) for all \( t, t' \in TRA_o, t \neq t' \):
\[
(t \rightarrow a, t' \rightarrow a', a \rightarrow a') \Rightarrow \text{ for all } O, O', b, b': b \rightarrow t, b' \rightarrow t' ; b \rightarrow b', b' \rightarrow b'.
\]
In our approach a conflict of two actions depends on the semantics of the accessed object. We assume a \textit{commutativity matrix} for every object for all their actions. It specifies for every action pair if they commute or if they are in conflict. If two actions commute then they preserve the consistent state of the accessed object even when executed in parallel. The matrix may be defined by the programmer or computed by the system, but this fact is of no relevance here.

A top-level-transaction may consist of several parallel processes because the precedence relations \( n_3 \) may not be total. Two actions of different processes may clearly be in conflict: But changes made by an action of a process may also be perceived by another action of the same process. Whether the access delivers a correct result is only a question of correct (serial) implementation and not of concurrency. Hence, actions of the same process can access objects independently. Such actions are never in conflict.

Definition 9
Two actions \( a \) and \( a' \) of different processes accessing an object \( O \) commute, denoted
\[
a \Theta a',
\]
if they are defined as commuting, else they are in conflict, denoted
\[
a \Theta a'.
\]
5. The action dependency is implied by the action dependency, denoted by \( \cdot \), and therefore we denote it by \( \cdot \).
Whether two schedules are equivalent is determined by the dependencies between conflicting actions and by the transactions which called them. But we want to take advantage of commutating actions. Therefore, we must answer two questions:

1. What are the dependencies of the actions?
2. What are the dependencies of the transactions?

The answers to these two questions are not independent of each other. From the point of view of an object an action may be a transaction. But this action accesses another object and is therefore an action for another object. We answer the second question and then the first.

Assume that an action dependency for an object schedule of the object O is given. For every pair a and a' of actions on O two cases exist: They are in conflict or they commute. If a and a' commute the calling actions r and r' cannot distinguish whether a depends on a' or a' depends on a. Hence, it makes no difference whether in an equivalent schedule r is executed before r' or vice versa. Dependencies of commutating actions need not be noted for transaction dependencies.

If a and a' are in conflict a difference whether a depends on a' or inverse can be determined. Therefore, the calling actions r and r' must have the same order in an equivalent serial schedule as a and a' have. The actions t and t' act as transactions for actions on O. In this case a transaction dependency for r and r' with the same direction as for a and a' is noted. We say that the dependency and its direction are inherited to the transactions t and t'.

We may not only consider action dependencies but also transaction dependencies. If a transaction depends on another transaction three cases exists.

1. They access different objects.
2. They access the same object and commute.
3. They access the same object and are in conflict. Assume, for the object schedule of the object O a transaction a' depends on another transaction a and they are called by r and r' respectively. In case (1) the dependency is not distinguishable for r and r' because a and a' access different objects. In case (2) the dependency can not be determined because a and a' commute. Only in case (3) according to conflicting actions the transactions r and r' must inherit the dependency from a and a'.

In the following we define the action dependency relation of an object schedule. A transaction depends on another transaction if the called actions are in conflict. Then the dependency of the actions determines the dependency of the transactions. However, two transactions also depend on each other, if the called actions are actions on some other object with a given dependency and if they are in conflict.

**Definition 10**

The action dependency relation \( \equiv \subseteq \text{TRA}_O \times \text{TRA}_O \) of an object schedule \( \text{Sch} = (\text{TS}, O, \cdot, \cdot) \) is defined as:

\[
(t, t') \equiv \begin{cases}
(t \rightarrow a, t' \rightarrow a') \land (a \cdot a' \land a \cdot a') & \text{or} \\
(t \rightarrow u, t' \rightarrow u') \land (u \cdot u' \land u \cdot u') & \text{or} \\
(t \rightarrow a, t' \rightarrow a') \land (a \cdot a' \land a \cdot a') & \text{or} \\
(t \rightarrow u, t' \rightarrow u') \land (u \cdot u' \land u \cdot u')
\end{cases}
\]

In the next section we give an example summarizing the definitions. (Example 4). Now we define the action dependency relation for an object schedule. Consider two object schedules named \( \text{Sch} = (\text{TS}, O, \cdot, \cdot) \) and \( \text{Sch}' = (\text{TS}, P, \cdot', \cdot') \). The actions of \( \text{Sch}' \) may be transactions of \( \text{Sch} \).

Therefore, an object schedule inherits the dependencies of conflicting actions from all the related object schedules. Two schedules are related if the first schedule specifies a transaction dependency and if one of the transactions is an action of the second schedule. But where does the inheritance process start?

In conventional systems an order of conflicting actions is (implicitly) assumed. It must, for example, be decidable if a "write" is executed before or after a "read". These actions are atomic but in an oo-system not all actions need to be atomic. However in an oo-system we have defined primitive actions. In case of a conflict primitive actions must be ordered. We postulate the following axiom that manifests the existence of this knowledge. Basing on the axiom we define the action dependency relation.

**Axiom 1**

\( a, a' \in \text{PR}_O \land a \cdot a' \Rightarrow \text{either either } a < a' \text{ or } a' < a. \)

**Definition 11**

Let \( a, a' \in \text{ACT}_O \). The action dependency relation \( \equiv \subseteq \text{ACT}_O \times \text{ACT}_O \) of an object schedule \( \text{Sch} = (\text{TS}, O, \cdot, \cdot) \) is defined as:

\( a \cdot a' \equiv \text{either Axiom 1 holds or} \)

A conventional serializable schedule must be equivalent to a serial schedule. "Equivalent" means that for every two transactions all conflicting actions are executed in the same order. This fact is also true for object schedules. But which transactions must be considered in an object schedule is determined by the transaction dependency relation. Therefore, we define equivalence with respect to the transaction dependency.

**Definition 12**

Two object schedules \( \text{Sch} = (\text{TS}, O, \cdot, \cdot) \) and \( \text{Sch}' = (\text{TS}, O, \cdot', \cdot') \) are equivalent if they have the same transaction dependency relation.

By defining serializability of an object schedule we must consider relationships between object schedules. Assume that three object schedules named \( \text{Sch}(\text{TS}, O, \cdot, \cdot), \text{Sch}'(\text{TS}, P, \cdot', \cdot') \) and \( \text{Sch}''(\text{TS}, Q, \cdot'', \cdot'') \) exist. Some transactions of \( \text{Sch}' \) and some transactions of \( \text{Sch}'' \) may be actions of \( \text{Sch} \). But their dependencies \( \cdot' \) and \( \cdot'' \) may contradict. This fact is not acceptable because it suggests a contradicting dependency of the transactions of an equivalent serial schedule for the object schedule of \( Q \). Such contradictions are not allowed, i.e. the action dependency relation must be acyclic.

**Definition 13**

An object schedule \( \text{Sch} = (\text{TS}, O, \cdot, \cdot) \) is object-oriented serializable if

(i) there exists an equivalent serial object schedule \( \text{Sch}' = (\text{TS}, O, \cdot', \cdot') \)

(ii) \( \cdot' \) is acyclic.

**5. OO-Serializability of a System Schedule**

Based on the definition of oo-serializability of object schedules we define serializability for a whole transaction system. Obviously, the schedule of a system is related to all object schedules. We define a system schedule as the set of the object schedules of all the objects of the system.

**Definition 14**

A system schedule \( SSch \) of a transaction system \( TS = (\text{OBJ}) \) is defined as:

\( SSch = \{ \text{Sch}(\text{TS}, O, \cdot, \cdot) \mid (O \in \text{OBJ}) \} \)

Surely, oo-serializability of a system schedule assumes oo-serializability of its object schedules. But we must consider relationships between object schedules. The action dependency relation of an object schedule of \( O \) considers only actions on \( O \). But contradicting transaction dependencies from other object schedules may also exist if only one (trans-)action accesses \( O \), the other (trans-)action accesses another object. Therefore, we define the added action dependency relation of an object schedule considering such actions as follows.

6. Conventionally (applied to our notation) two schedules \( \text{Sch} = (\text{TS}, O, \cdot, \cdot) \) and \( \text{Sch}' = (\text{TS}, O, \cdot', \cdot') \) are equivalent if the following holds:

\[
\text{for all } a, a' \in \text{ACT}_O : a \cdot a' \equiv a \cdot a'.
\]

This definition is the same as the first part in Definition 10, but the second part is not considered in the conventional definition.

7. Note that not all \( t \in \text{TRA}_O \) are \( \sim \)-related.
Definition 15
Let the added set of actions ADD₀ of the object O be defined as:

\[ ADD₀ = \{ b \mid \text{ ADD₀} \rightarrow Q.o \} \]

Let \( a \in \text{ACT}_0 \), \( b \in \text{ADD}_O \). The added action dependency relation \( \leftrightarrow \) of \( (\text{ACT}_0 \cup \text{ADD}_O) \times (\text{ACT}_0 \cup \text{ADD}_O) \) of an object schedule \( S_{ch} = (TS, O, \cdot, \cdot) \) is defined as follows:

\[ a \leftrightarrow b \iff \text{ a schedule } S_{ch} = (TS, R, \cdot, \cdot, \cdot) \text{ exists with: } a = b \]
\[ b \leftrightarrow a \iff \text{ a schedule } S_{ch'} = (TS, S, \cdot, \cdot, \cdot) \text{ exists with: } b = a \]

Now we can define serializability of a system schedule. The added action dependencies must not imply contradictions, i.e., they must be acyclic and all object schedules must be object-oriented serializable. We call such a system schedule object-oriented serializable.

Definition 16
A system schedule \( S_{sch} = (OBI, \text{TOP}) \) is object-oriented serializable, if for all \( S_{sch} = (TS, O, \cdot, \cdot) \) of \( S_{sch} \) the following holds:

(i) \( S_{sch} \) is object-oriented serializable and
(ii) \( \cdot \leftrightarrow \cdot \) is acyclic.

Example 4
Figure 7 contains the three top-level-transactions \( T_1, T_2 \) and \( T_4 \) of Example 1. In addition, \( T_2 \) changes the previous inserted item \( DBMS \). A fourth transaction \( T_4 \) reads the items sequentially. In Figure 7 we draw the dependencies for Page 4712 and Item 8. Figure 8 contains the dependencies for the other objects. Along the bold drawn arcs dependencies are inherited. Let us have a look at Page 4712. The actions called from \( T_1, \text{Leaf11, insert(DBMS)}_1 \), and \( T_3 \) respectively must preserve the same dependency order of conflicting actions. The dependencies are the same as in Example 1. Hence, object-oriented serializability includes multi-layer serializability.

<table>
<thead>
<tr>
<th>Objects</th>
<th>Schedule dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page 4712</td>
<td>see the long dashed arcs in Figure 7</td>
</tr>
<tr>
<td>Leaf11</td>
<td>insert(DBMS)_1, insert(DBMS)_2</td>
</tr>
<tr>
<td>BpTree</td>
<td>insert(DBMS)_1, search(DBMS)_1</td>
</tr>
<tr>
<td>Item8</td>
<td>see the short dashed arcs in Figure 7</td>
</tr>
<tr>
<td>LinkedList</td>
<td>( T_2 \leftrightarrow \text{readSeq} )</td>
</tr>
<tr>
<td>Enc</td>
<td>( T_1 \leftrightarrow \text{readSeq} )</td>
</tr>
</tbody>
</table>

Fig. 7: Object-oriented transactions
(with dependencies for Page 4712 and Item 8)
Discussions with Klaus Drosten, Manfred Kaul and Peter Muth are gratefully acknowledged. We are also indebted to the referees for their valuable comments.

Acknowledgement

References