Chapter 6

Concurrency Control on Objects: Notions of Correctness

"No matter how complicated a problem is, it usually can be reduced to a simple comprehensible form which is often the best solution." (An Wang)

"Every problem has a simple, easy-to-understand, wrong answer." (Anonymous)

6.1 Goal and Overview of the Chapter

After the previous chapters' detailed exploration of page-model concurrency control methods, we will now make a major step to enrich our underlying computational model by turning to the object model. This means, as you may recall from Chapter 2, that transactions are now modelled as trees of operation invocations rather than "flat" sequences (or partial orders) of simple read or write steps. The major incentive for studying the richer but also more complex object model lies in its potential for higher concurrency by exploiting specific semantic properties of the operations, in particular commutativity of higher-level operations such as "deposit" or "withdraw" on bank-account objects.

The goal of this chapter is to develop correctness criteria for concurrent executions of object-model transactions. Analogously to the flow of arguments for the page model, these criteria will then serve as the basis for practical concurrency control algorithms, to be developed in the subsequent chapters. In our quest for appropriate correctness criteria we have to address two major differences from the page model: (i) the operations of a transaction are more general than merely read and write, and (ii) operations can themselves invoke other operations thus leading to a transaction tree. After introducing the syntactical notions of object-model his-

tories and schedules in Section 6.2, we will address these two issues in two steps: in Section 6.3 we will consider "flat" schedules with semantically rich operations; then we will look into the general case of transaction trees in Section 6.4.

It will turn out that allowing general operations but sticking to the "flat" structure of transactions is a relatively straightforward extension of what we have discussed within the framework of the page model. In particular, we will be able to generalize the notion of conflict serializability in an appropriate manner. Coping with transaction trees poses more fundamental difficulties. Here we will see that neither view serializability nor conflict serializability can be generalized directly, and we will thus need to develop a more elaborate correctness criterion, in the style of (commutativity-based) reducibility as briefly discussed in Chapter 3. This fundamental criterion, to be coined *tree reducibility*, will be the subject of Section 6.4. The need for algorithmic tractability will then motivate our exploration of sufficient, more "constructive" conditions for correctness in Section 6.5. These considerations will emphasize a particularly interesting special case of the object model, namely, the case of *layered transactions* where the nodes of the transaction trees correspond to operations of a strictly layered system architecture (such as the five-layer database server sketched in Chapter 1). In the final Section 6.6, we will give an outlook on exploiting further semantic properties of operations, beyond the property of general (state-independent) commutativity.

6.2 Histories and Schedules

Let us first briefly recall from Chapter 2 how transactions are characterized in the object model. The dynamic invocations of operations by a transaction spawn a tree of labeled nodes where each node corresponds to an invoked operation together with its input and possibly also output parameters. The parent of a node is the operation from which the node's operation was called, and the children of a node are the operations called by the node. Ultimately, each invoked operation results in calls of read and write operations, which thus form the leaves of the tree. As we again want to allow concurrent or parallel threads even within a transaction, the execution order is modeled as a partial order of the leaves; the ordering (or non-ordering, i.e., concurrent execution) of higher-level operations is derived from the leaf order: we say that p precedes q if p terminates before q begins, or more technically, the last leaf-level descendant of p precedes the first leaf-level descendant of q.

The next step is to consider interleavings of such transaction trees. Not surprisingly, this leads to a notion of partially ordered forests:

object-model history

Definition 6.1 (Object-model history)

Let $T = \{t_1, \ldots, t_n\}$ be a (finite) set of transaction trees, where each $t_i \in T$ is a pair $(op_i, <_i)$ of labeled tree nodes along with a partial order of the leaves. A *history* s (or complete schedule) for T is a partially ordered forest $(op(s), <_s)$ with node set $op(s)$ and partial order $<_s$ s.t.

1. $op(s) \subseteq \bigcup_{i=1}^{n} op_i \;\cup\; \bigcup_{i=1}^{n} \{a_i, c_i\}$, and $\bigcup_{i=1}^{n} op_i \subseteq op(s)$, i.e., s contains the operations of the given transaction trees and a termination operation c_i (commit) or a_i (abort) for each transaction $t_i \in T$,
2. $(\forall i, 1 \leq i \leq n)\;\; c_i \in op(s) \Leftrightarrow a_i \notin op(s)$,
3. a_i or c_i is a leaf node with the transaction t_i as its parent,
4. $\bigcup_{i=1}^{n} <_i \;\;\subseteq\;\; <_s$, i.e., all transaction orders are contained in the partial order given by s,
5. $(\forall i, 1 \leq i \leq n)\; (\forall p \in op_i)\;\; p <_s a_i$ or $p <_s c_i$,
6. any pair of leaf operations $p, q \in op(s)$ from distinct transactions accessing the same data item s.t. at least one is a write operation must be ordered in s, i.e., either $p <_s q$ or $q <_s p$.

Like the individual transaction trees, the partial order of a history refers to the leaves of the forest, and we can derive a partial order among all nodes by the following convention:

tree-consistent node ordering

Definition 6.2 (Tree-consistent node ordering)

In an object-model history $s = (op(s), <_s)$ the ordering $<_s$ of leaf nodes is extended to arbitrary nodes as follows: two nodes p and q are considered as ordered, i.e., $p <_s q$ if for all leaf-level descendants p' and q' of p and q, resp., the order $p' <_s q'$ holds according to the original (i.e., non-extended) $<_s$. Such an ordering is called *tree-consistent.*

Now the notion of a *prefix* of a history, which we have introduced in Chapter 3 for the "flat" histories of the page model, can be applied to transaction forests: a prefix of a history $s = (op(s), <_s)$ is a forest $s' = (op(s'), <'_s)$ with $op(s') \subseteq op(s)$ and $<'_s \;\;\subseteq\;\; <_s$ such that for each $p \in op(s')$ all ancestors of p and all nodes $q \in op(s)$

with $q <_s p$ must be in $op(s')$, too, and $<'_s$ equals $<_s$ when restricted to $op(s')$. Now the definition of an object-model schedule is obvious:

object-model schedule

Definition 6.3 (Object-model schedule)

An object-model schedule is a prefix of an object-model history.

As an example consider a simplified version of the funds-transfer transaction from Chapter 2, consisting of a *withdraw* and a *deposit* operation on two bank accounts. In contrast to the more elaborate example in Chapter 2, we assume that both of these operations are directly "translated" into page operations, i.e., reads and writes. Figure 6.1 shows a history, i.e., concurrent execution, of two instances of such an object-model transaction: t_1 transfers money from account a to c, and t_2 transfers money from b to c. We assume that the relevant records for accounts a, b, and c reside on pages p, q, and r, respectively, and that all withdraw operations also need to access page t to obtain and track an (automatically generated) approval code; for simplicity, we disregard steps for locating these pages, e.g., index traversals. We also simplify the notation in that we only list those parameters of the invoked operations that are relevant for the discussion; for example, the withdraw operation actually has two input parameters and one result parameter, namely, the id of the bank account on which it operates, the amount of money to be withdrawn, and a returncode which states if the withdrawal was successful or not. Figure 6.1 only shows the first argument, but in other examples we may need to make more parameters explicit as well.

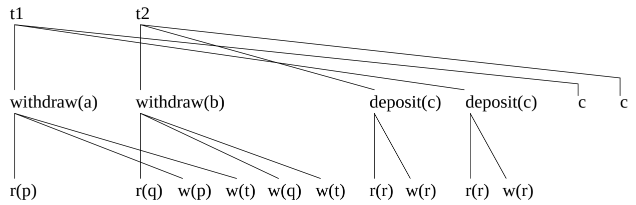


Figure 6.1: Example of an object-model history.

In examples such as Figure 6.1 we will indicate the ordering of operations by drawing the leaf nodes in their execution order from left to right. As the caller-callee relationship in transaction trees is captured by vertical or diagonal arcs, the crossing of such arcs indicates that two (non-leaf) operations are concurrent. In Figure 6.1, for example, the two withdraw operations that originate from t_1 and t_2, resp., are concurrent. Of course, for non-total leaf orders, we need a more explicit specification such as drawing directed arcs between ordered pairs of operations, as mentioned already in Chapter 3 for "flat" schedules.

notation for caller-callee relationship

Sometimes it may also be necessary to indicate the caller-callee relationship more explicitly rather than drawing the arcs of the trees; we will then use subscripts for tree nodes as follows: (i) the subscript of a root t_i is i, and (ii) the subscript of the j-th child of a node with subscript ω is ωj, i.e., the parent's subscript with j appended. So, in Figure 6.1, for example, the $w(t)$ operation belonging to t_1 would be denoted as $w_{113}(t)$, thus bearing the identification of its entire ancestor chain.

serial schedule

Now that we have both a formal definition and an intuitive understanding of object-model schedules, the notion of a serial schedule can be defined in a straightforward way:

Definition 6.4 (Serial object-model schedule)

An object-model schedule is *serial* if its roots are totally ordered and for each root and each $i > 0$ the descendants with distance i from the root are totally ordered.

The first condition in this definition states that transactions (i.e., the roots) are executed sequentially, and the second condition additionally requires that all steps within a transaction are also sequential. Note that it is not enough to require that all leaves of a root are ordered, as this would still allow interleaved executions of some inner-node operations.

isolated subtree

A concept that is related to the notion of serial executions is the following:

Definition 6.5 (Isolated subtree)

A node p and the corresponding subtree in an object-model schedule s are called *isolated* if

1. for all nodes q other than ancestors or descendants of p the property holds that for all leaves w of q either $w <_s p$ or $p <_s w$, and
2. for each $i > 0$ the descendants of p with distance i from p are totally ordered.

The second condition is identical to one of the two requirements in Definition 6.4, simply stating that the subtree corresponds to a strictly sequential execution. The first condition requires that no other subtree can have any leaf operation in between one of the leaves of subtree p (with the trivial exception of p's ancestors or descendants including p itself). So the subtree rooted at p forms an indivisible unit. Note, however, that it is allowed for another subtree to have some leaves both preceding and following p. In the sense of the theory of relative serializability developed in Section 3.10, our notion of an isolated subtree p makes a statement only about the indivisible units of p relative to other subtrees, namely, forcing all of p to be a single indivisible unit, but we do not say anything about the interleavings of other subtrees (i.e., their indivisible units relative to p). A very strong special case is when *all roots* of a schedule are isolated: such a schedule must be serial.

layered histories and schedules

The example in Figure 6.1 is special in that the two interleaved transaction trees are both perfectly balanced trees, meaning that all leaves, with the exception of the commit operations, have the same distance from their roots. This is not by accident in the example; rather it captures a common case that the invoked operations correspond to fixed levels of a strictly layered system architecture. We accounted for this specific structure already by drawing the tree nodes of Figure 6.1 in three levels: page accesses at the bottom level, method invocations on account objects at the middle level, and transactions at the top level. We refer to such highly structured object-model schedules as "layered schedules".

Definition 6.6 (Layered history and schedule)

An object-model history is a *layered history* if all leaves other than commit or abort operations have identical distance from their roots. A layered history with a leaf-to-root distance of n is called an n-level history. In an n-level history, operations with distance i to the leaf level are called *level-i operations* (L_i operations) (i.e., leaves are at level L_0 and roots at level L_n).

A *layered schedule* is a prefix of a layered history.

So, according to this definition the example of Figure 6.1 is a 2-level history, with levels numbered from 0 to 2 in a bottom-up manner. Other examples, to be studied later, would include 2-level histories with transactions at level L_2 that consist of record accesses at level L_1 which are translated into page accesses at level L_0. It is worth mentioning that page-model schedules are a special case of n-level schedules with $n = 1$.

Two examples for non-layered histories are shown in Figure 6.2.

Both are similar to the previous funds-transfer scenario, with the following differences: In the upper example, withdraw operations do not include the step for obtaining and tracking an approval code, so that the transactions must invoke the data-server operations that correspond to this specific step directly (resulting in the $w(t)$ operations); in the lower example, the deposit steps are no longer explicitly represented and the corresponding page operations are invoked directly by the transactions. The latter scenario may appear a bit odd: why would the withdrawals be explicit whereas the deposit operations no longer show up? This may have (at least) two different explanations: First, it could happen that the transactions do, for efficiency or other reasons, "bypass" the object interface avoiding the call of the deposit method, although they do observe the encapsulation of account objects as far as withdrawals are concerned. The second possible explanation simply is that the histories on which we reason about correctness are merely *models* of the actual executions in that they track particularly relevant actions but do not necessarily need to be complete pictures of everything that happened in the underlying server. So it could well be that the read and write operations on page r were invoked on behalf of explicitly invoked deposit operations, but these deposit operations are not observed and we only know the transactions to which these page operations belong. Of course, it is usually up to ourselves how completely and accurately we devise our modeling of some specific system, but the above should be taken as a word of caution that such "non-standard" scenarios may arise and should be tractable within our framework.

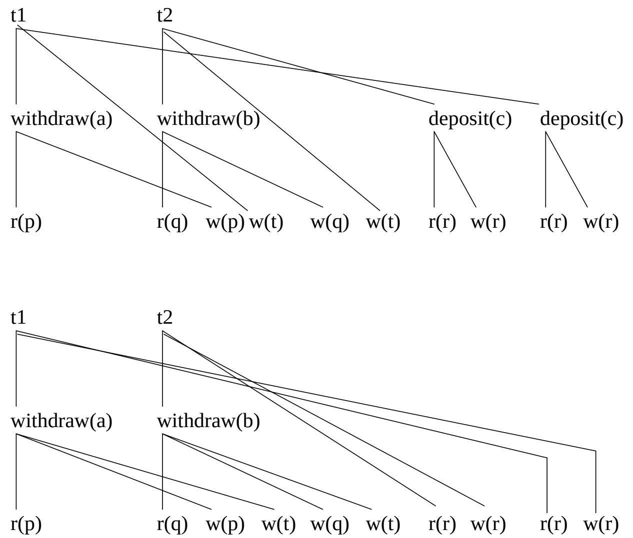


Figure 6.2: Two examples of non-layered object-model schedules.

6.3 Conflict Serializability for Flat Object Transactions

Within the class of layered schedules, a special subclass of particular interest are what we call *flat object schedules*. These are 2-level schedules with object-method invocations at the higher level L_1 and ordinary read/write operations at the lower level L_0. The decisive particularity, however, is that we essentially disallow concurrent L_1 operations in this subclass. In other words, transactions may be interleaved, but for each pair of object-method executions we require an ordering.

flat object schedule

Definition 6.7 (Flat object schedule)

A 2-level schedule s is called a *flat object schedule* if for each pair p, q of L_1 operations the following two conditions hold:

1. $(\forall p' \in child(p), q' \in child(q))\ \ p' <_s q'$ or
   $(\forall p' \in child(p), q' \in child(q))\ \ q' <_s p'$
2. $(\forall p', p'' \in child(p))\ \ p' <_s p''$ or $p'' <_s p'$

where $child(p)$ denotes the set of children of operation p.

The first condition requires a total order among L_1 operations, the second condition demands that each L_1 operation is itself a serial execution of page accesses.

Figure 6.3 shows an example of a flat object schedule. Note that the total ordering of L_1 operations and their sequentiality are easily recognized by the observation that none of the caller-callee arcs from the L_1 operations to their L_0 children cross each other.

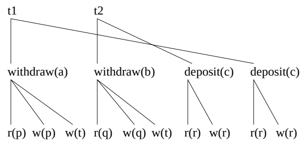


Figure 6.3: Example of a flat object schedule.

Our specific interest in flat object schedules lies in the fact that their structure is as close as possible to conventional, one-level schedules. As object-level operations

are executed sequentially, the resulting page-level accesses are actually not of specific interest as far as concurrency control is concerned. So an intriguing abstraction step now is to disregard the page-level operations completely, effectively constructing a one-level schedule that consists of abstract, yet *indivisible* operations. It is in this sense that we now have a situation comparable to that for conventional page-model schedules, with the key property of operations being their indivisibility.

The natural next step is to reconsider the technical approaches that we explored towards correctness criteria for the page model as to whether they are potentially applicable to our current setting as well. It is fairly obvious that the notions of view serializability and the underlying reads-from relation cannot be generalized to flat object schedules without "degrading" all operations into read and write actions and losing their richer semantics. The simpler notion of conflict serializability and its related notion of commutativity-based reducibility, on the other hand, are indeed promising candidates for this purpose. In fact, a conflict between operations is nothing but the observation that the operations do not commute, and this is obviously something that can also be applied to semantically richer operations.

commutative operations

Definition 6.8 (Commutative operations)

Two operations p and q (on some objects) are *commutative* (or they *commute*) if for all possible sequences of operations α and ω, the return parameters in the concatenated sequence $\alpha p q \omega$ are identical to those in the sequence $\alpha q p \omega$.

So, for example, two operations $deposit(a, 10)$ and $deposit(a, 20)$ that deposit \$10 and \$20 into account a can be tested by various numbers of successive *withdraw* operations to see that both possible orderings of the two *deposit* operations achieve the same effects.

The above definition emphasizes the encapsulated nature of objects; the state of an object is observable only through explicitly invoked methods rather than being exposed to arbitrary reads. In this sense, commutativity demands that the ordering of two operations is irrelevant: different orderings cannot be distinguished as far as other explicit method invocations are concerned. So if two operations p and q are not commutative, the difference between the two possible orderings should be observable in the result parameters of p or q or some subsequently invoked operation. In the latter case, the operation to observe the non-commutativity of p and q is not necessarily the next one after p and q; the difference may become visible only after a finite number of additional operations (i.e., the sequence ω in the definition). Finally note that we want commutativity to hold in every possible state of the object(s) on which p and q operate. As states are not directly visible, this means that we must take into account that p and q are executed after some (finite) history of operations that result in a certain state (i.e., the sequence α in the definition). We will refer to this most general notion of commutativity also as *state-independent* or *general commutativity*, and will discuss possible relaxations later in Section 6.6.

state-independent commutativity

Commutativity can be a delicate property, which is all but trivial to specify. As soon as the programming language in which object methods are written is sufficiently expressive, the commutativity of two methods is, like most other interesting metaproperties of programs, undecidable. Our escape from this dilemma is that rather than automatically inferring such properties, we expect an application expert to *assert* the commutativity of operations. In doing this, the expert may be conservative in that operations are assumed as non-commutative if she is in doubt about a positive statement. After all, operations that may superficially appear commutative may have side-effects on other objects that ultimately render them non-commutative. For example, two customer orders for different items (or even for the same item if the item is guaranteed to be on stock) are commutative as long as we consider only operations like PlaceOrder, Shipment, and Payment, but they turn out to be non-commutative if we add the twist that the customer who places the hundred thousandth order wins a car. It is for exactly such reasons that we have avoided establishing a one-to-one correspondence between methods and objects in our above definition. Rather, we do allow methods to operate on more than one object, possibly in a transitive manner. Likewise, we do not infer from the disjointness of two operations' parameter lists that they are necessarily commutative.

Despite these potential pitfalls, we will, in many practical situations, positively assume that non-commutative operations arise only on the same object type, thus declaring operations on different object types as pairwise conflict-free. For the object type "bank account", for example, we may consider three methods "withdraw", "deposit", and "getbalance," all of which have an object id as a primary input parameter; withdraw and deposit operations have an amount of money as a second parameter and merely return a binary status about the success or failure of the operation, and getbalance operations return the current balance of a bank account.

For this setting, two deposit operations are commutative because of the fact that both merely increment the account's balance, without caring about the prior or resulting value of the balance. In contrast, a withdraw operation does not commute with a deposit operation, as withdrawals usually enforce an overdraft protection, so that money can no longer be withdrawn once the balance drops below a certain threshold. Thus, if an account's balance had a value equal to this threshold and a deposit operation first incremented the balance, a withdraw operation would succeed if the requested amount is at most as high as the deposited amount, whereas the withdrawal would fail to withdraw money if the deposit operation were second in

the ordering.

A complete commutativity specification for these three operation types is given in the table below, with + standing for commutativity and − for non-commutativity. x denotes the account parameter of the operations and Δ the amount of money that is withdrawn or deposited. Strictly speaking, the table should also consider the case of two operations on different account objects x and y, but this is generally assumed as a + and therefore omitted. Also, we would typically assume that each of the three operations commutes with all operations on other object types such as customer objects.

	$withdraw(x, \Delta_2)$	$deposit(x, \Delta_2)$	$getbalance(x)$
$withdraw(x, \Delta_1)$	−	−	−
$deposit(x, \Delta_1)$	−	+	−
$getbalance(x)$	−	−	+

commutativity-based reducibility

Now that we have established a (symmetric) commutativity relation among the invoked operations, we can directly apply the notion of commutativity-based reducibility to flat object schedules, essentially repeating the arguments from Chapter 3.

Definition 6.9 (Commutativity-based reducibility)

A flat object schedule s is *commutativity-based reducible* if it can be transformed into a serial schedule by applying the following rules finitely many times:

1. *Commutativity rule:*
   The order of two ordered operations p and q with, say, the order $p <_s q$ can be reversed if
   (a) both are isolated, adjacent in that there is no other operation r with $p <_s r <_s q$, and commutative, and
   (b) the operations belong to different transactions or, if they belong to the same transaction t_i, the reversal does not contradict the specified order $<_i$ within t_i.
2. *Ordering rule:*
   Two unordered leaf operations p and q can (arbitrarily) be ordered, i.e., assuming either $p < q$ or $q < p$ if they are commutative (i.e., both are reads or they operate on different data items).

Note that the ordering rule is merely an auxiliary rule to cope with partial orders at the leaf level. Further note that the isolation condition is actually redundant for flat object schedules with a total ordering of the L_1 (i.e., object-level) operations. We nevertheless state this explicitly as it will play a crucial role when we turn back to the general case of object-model schedules.

Consider the example of Figure 6.3. All we need to do to transform this schedule into a serial one is to exchange the order of t_1's deposit operation with the deposit of t_2 and then, in a second transformation step, with the withdrawal of t_2. These steps pull t_1's deposit operation back, out of the interleaving with t_2. Note once again that the transformations are feasible only because the affected L_1 operations are already isolated in that their page-level executions are strictly sequential.

A dual way of reasoning would be to consider the conflicts, i.e., non-commutative pairs of operations, and test for conflict equivalence with a serial schedule. This leads to the following criterion:

conflict equivalence, conflict serializability

Definition 6.10 (Conflict equivalence and conflict serializability)

Two flat object schedules s and s' are *conflict equivalent* if they consist of the same operations and both have the same ordering for all non-commutative pairs of L_1 operations.

A flat object schedule is *conflict serializable* if it is conflict equivalent to a serial schedule.

Obviously, the *conflict graph* construction of Chapter 3 can be carried over to flat object schedules as well. Indeed, for a flat object schedule s, the corresponding conflict graph is a graph with the transactions of s as nodes and an edge from t_i to t_j if there are non-commutative L_1 operations p_i and q_j belonging to t_i and t_j, resp. Now the following theorem is obvious:

Theorem 6.11

Let s be a flat object schedule. Then s is conflict serializable iff its conflict graph is acyclic. Furthermore, s is conflict serializable iff it is commutativity-based reducible.

The conflict graph for the example of Figure 6.3 does not have any edges because all L_1 operations commute, and thus is trivially acyclic. Note, however, that a conventional conflict graph derived from the L_0 operations, i.e., page accesses of the two transactions, would have a cycle caused by the conflicts on pages t (with t_1

preceding t_2) and r (with t_2 preceding t_1). This shows exactly the benefit of exploiting the "semantic knowledge" about the object-level operations for withdrawals and deposits. We can now allow schedules that would be disallowed under the purely read-write-based viewpoint of the page model.

6.4 Tree Reducibility

To generalize the notions of reducibility or serializability to non-flat and, ultimately, even non-layered schedules, recall the example of Figure 6.1 from Section 6.2. The decisive difference between this schedule and a flat object schedule is that its object-level operations are again interleaved at the page level. So if we could reason that the page-level execution was equivalent to a schedule with isolated L_1 operations, then we would be in a position to abstract from the details of the page accesses and simply ignore the page level. This is essentially what we have done with flat object schedules anyway, with the only difference that we knew in advance about the isolation of the L_1 operations and thus found it easy to justify the abstraction step.

At this point it should be no surprise that the arguments for showing equivalence to operations being isolated are again commutativity arguments. In Figure 6.1, for example, the second and third page access of operation $withdraw_{11}(a)$, namely, $w_{112}(p)$ and $w_{113}(t)$, both commute with the preceding $r_{211}(q)$ operation belonging to t_2; hence the two write accesses of t_1 can be pulled back so that t_1's withdraw operation would completely preceed the withdraw operation of t_2. Then, once we have "disentangled" the two interleaved withdraw operations, the argument from the previous section is applicable: all object-level operations are isolated and commute with each other, so that the entire schedule is equivalent to either of the two possible serial orders of t_1 and t_2.

The key abstraction in the entire reasoning, disregarding the operations inside isolated subtrees, boils down to reducing such a subtree to its root. By adding an appropriate *tree pruning rule* to the notion of commutativity-based reducibility, we finally arrive at the desired generalized type of reducibility which we coin "tree reducibility," as it is indeed applicable to the entire class of object-model schedules.

tree reducibility

Definition 6.12 (Tree-reducible history)

An object-model history s=$(op(s), <_s)$ is *tree reducible* if it can be transformed into a total order of its roots by applying the following rules finitely many times:

1. *Commutativity rule:*
   The order of two ordered leaf operations p and q with, say, the order $p <_s q$, can be reversed provided that
   (a) both are isolated, adjacent in that there is no other operation r with $p <_s r <_s q$, and commutative, and
   (b) the operations belong to different transactions or, if they belong to the same transaction t_i, the reversal does not contradict the specified order $<_i$ within the transaction t_i, and
   (c) the operations p and q do not have ancestors, say p' and q', respectively, which are non-commutative and totally ordered (in the order $p' <_s q'$).
2. *Ordering rule:*
   Two unordered leaf operations p and q can (arbitrarily) be ordered, i.e., assuming either $p < q$ or $q < p$ if they are commutative (i.e., both are reads or they operate on different data items).
3. *Tree pruning rule:*
   An isolated subtree can be pruned in that it is replaced by its root.

An object-model schedule is *tree reducible* if its committed projection is tree reducible.

Applying the tree pruning rule results in trees that are, strictly speaking, no longer object-model schedules according to the definition of an object-model transaction given in Chapter 2. The reason for this subtlety is that our original definition has required the leaves of such transactions to be page-model read or write operations. Once we prune a tree, this condition is obviously no longer satisfied. To resolve this syntactic mismatch, we now relax the definition of object-model schedules so that they are allowed to contain transactions whose leaves are high-level operations (i.e., have pruned the original page-model leaves), provided that these high-level operations are guaranteed to appear as if they were indivisible operations. Indeed this is the whole point of pruning the tree: when we know that some lower-level interleaving has the same effect as if each of the corresponding high-level operations

were an indivisible unit, then and only then we can safely prune the descendants of such high-level operations. This is the key abstraction that we exploit in our reasoning about object-model schedules; so object-model schedules are now allowed to contain also *abstractions of object-model transactions* in the above sense. Note, however, that the abstraction cannot be reversed: it does not make sense to expand again leaves of pruned trees, which correspond to high-level operations, into "full" trees with page-model operations as leaves.

abstractions of object-model schedules

Compared to the earlier definitions of commutativity-based reducibility for page-model and flat object-model schedules, the commutativity rule in the above definition has become more sophisticated by adding the third condition regarding ancestors. The reason for this addition is a subtle, yet important one. Whenever two higher-level operations are totally ordered and do not commute, this ordering is a hard constraint for their descendants. Reversing the ordering of two leaves that belong to such higher-level operations and, in the end, possibly reversing even the order of higher-level operations is not legitimate, as this could lead to different effects of the higher-level operations themselves. As the higher-level operations are in conflict, a reversal of their order would in general not preserve the original return values. As an example consider two serial withdraw operations on the same bank account such that the first one is successful in withdrawing money, whereas the second one is unable to do so because of the account's overdraft protection. If we could rearrange the leaves of these two operations such that, ultimately, the second withdrawal preceded the first one, it would be impossible for the operations to have their original return values. So the fundamental problem is that reordering commutative leaf operation without carefully taking into account their ancestors would open a "backdoor" for implicitly (almost "unconsciously" so to speak) reordering these ancestors even if such reordering were unacceptable.

Such subtleties did never arise in the commutativity-based reasoning for the page model and for flat object schedules, as the only possibly affected ancestors are the transaction roots. Different transactions, however, are usually assumed to be "independent" in the sense of being commutative; this is exactly the reason why the page-model serializability theory considers *each possible* serial transaction order as acceptable.

As a second example for applying our new fundamental concept of tree reducibility, consider Figure 6.4 with record-level operations *store*, *fetch*, and *modify* inside a database server, which are dynamically translated into page accesses. The example assumes that records w, x, and y initially are all located on page p, and that all operations first look up some address translation table on page t, say for translating record identifiers into page numbers. During the execution of a $store(z)$ operation for a new record z, record x is moved from p to q, say for lack of space on page p, and the address translation table on page t is updated accordingly. Recall from our overview of database-system internals in Chapter 1 that such situations are commonplace inside a data server.

In this example, the interleavings of the *store* and *fetch* pair and of the last two *modify* operations can both be eliminated by pulling back the two page reads of *fetch(x)* so that they are placed in front of *store(z)*, and pushing forward the $r(t)$ operation issued by t_2's *modify(y)* operation. The resulting schedule has all object-level operations isolated, so that the page-level subtrees can be pruned, with the following outcome:

$$fetch_{21}(x)\ store_{11}(z)\ modify_{12}(y)\ modify_{13}(w)\ modify_{21}(y).$$

Now the only non-commutative pair of operations consists of the two *modify* operations on record y; so we rearrange the commutative operations (e.g., by pushing forward t_2's $fetch(x)$ operation to produce a serial order with all operations of t_1 preceding all of t_2. Now the transactions themselves, i.e., the roots in the forest, are isolated and we can prune their record-level operations as well. In this way we have finally proven that the execution in Figure 6.4 is can be reduced and thus is equivalent to a serial order of the transaction roots with $t_1 < t_2$.

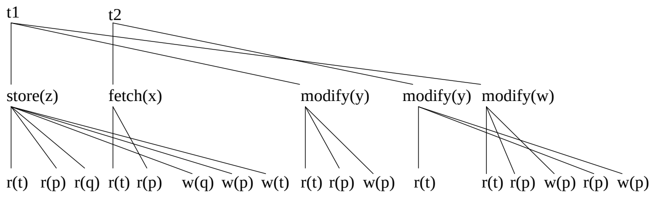


Figure 6.4: Example of a correct object-model schedule with record-level operations.

Note that the serialization order with t_1 preceding t_2 is the only feasible one in the example of Figure 6.4. This may appear somewhat strange, as this order would imply that the $fetch(x)$ operation of t_2 followed t_1's $store(z)$ and thus the movement of record x from page p to page q. So it seems that the $fetch_{21}(x)$ operation should have a read access to q among its children; however, this was not the case in the original schedule where x has still been read from p. The explanation for this seemingly confusing observation simply is that we do not claim to have an equivalent serial execution with every detail of the necessary page accesses. Rather, the abstraction

step of pruning these details once we have isolated subtrees cannot be reversed by later expanding the transactions with their original descendants. In the serial execution with $t_1 < t_2$, the $fetch(x)$ operation of t_2 would again issue its descendants dynamically depending on the current internal state of the underlying data pages. However, this does in no way affect the validity of our overall argumentation that the actual execution, as given by Figure 6.4, is equivalent to a serial execution from the perspective of the transactions. The point is that the low-level details such as the specific pages on which the various records reside do not matter at a higher level of abstraction, provided that no internal, i.e., storage-level inconsistency can arise.

To demonstrate that subtree isolation is all but trivial, consider another example with record-level operations shown in Figure 6.5. Here, because of the page-level conflicts between $w_{115}(p)$ and $r_{212}(p)$ on one hand and $r_{211}(t)$ and $w_{116}(t)$ on the other, it is impossible to rearrange the page-level operations towards isolating the $store(z)$ and $fetch(x)$ operations. Hence the schedule is not tree reducible. Note that it does not help in this situation that the subtree rooted at the $fetch(x)$ operation is already isolated right away (but not the other subtree rooted at $store(z)$). We could thus prune the $fetch$ subtree first rather than trying to commute leaves, but then it would be impossible to apply commutativity arguments to the $fetch(x)$ operation with regard to the r and w leaves of the $store$ subtree; so we could still not isolate the $store$ subtree. In fact, the shown interleaving of page accesses would result in an inconsistent view of the underlying storage structures: $fetch_{21}(x)$ would see the "old" contents of the address translation table, as of before the relocation of record x, and then erroneously access page p although the record has meanwhile been moved to q. It is of crucial importance to prevent such anomalies, and this is well reflected in the tree reducibility criterion.

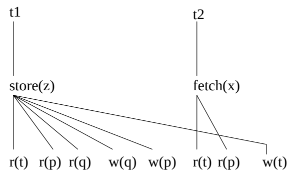


Figure 6.5: Example of a disallowed object-model schedule with record-level operations.

As a final example in this section, consider the non-layered schedule in Figure 6.6. As mentioned before, tree reducibility can be applied to this most general class of schedules as well. The example shows two transactions that issue operations for electronic purchases; these operations are then dynamically transformed into steps for checking the availability of the desired items, initiating their shipping, and handling the customer's payment. For simplicity, all parameters are omitted in the figure. The *CheckItem* operations are assumed to initate stock refillings by means of appending appropriate entries to a list of items with low quantity-on-hand. The two invocations of the *Payment* operation result in two different subtrees reflecting different types of payment; so this again illustrates the dynamic nature of the transaction trees which are spanned during the execution depending on actual parameters and the current states of objects. The *Payment* operations also put entries into some kind of audit trail, stored on page t, for tracking purposes. It is for the importance of tracking and reproducing the exact timing of financial aspects that we consider *Payment* operations to be non-commutative among themselves. In contrast, *Shipment* and *CheckItem* operations are assumed to be commutative with each other and also with *Payment*. Finally, *CheckCard* and *CheckCash*, the operations for verifying a credit card (or some form of cyber cash), are assumed to be commutative, whereas *Append* operations on a list do obviously not commute. Under all these (admittedly arbitrary, but reasonably natural) assumptions, it is now left to the reader to verify that the schedule shown is indeed tree reducible, arriving at the serial order $t_2 < t_1$.

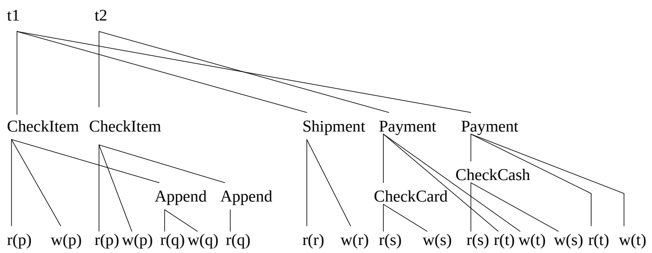


Figure 6.6: Example of a correct object-model schedule from an e-commerce scenario.

6.5 Sufficient Conditions for Tree Reducibility

The fundamental notion of tree reducibility is very intuitive, and it is easy to use for checking the correctness of a given schedule. However, it is not constructive in the sense that we can immediately derive from it a concurrency control algorithm that reconciles the higher concurrency that object-model schedules give us over page-model schedules with an efficient, low-overhead implementation. The purpose of this section is to introduce a number of sufficient conditions for tree reducibility that are more suitable for efficient scheduling algorithms. We will again focus on the special, but very important case of layered schedules first.

An obvious idea for layered schedules is to conceive concurrency control in the form of layers of conventional, conflict-driven schedulers, where each scheduler is modular in that it sees only operations of a particular level and knows about the parents of these operations but nothing else. For further elaborating this idea, the following notion of a level-to-level schedule is a convenient construct:

level-to-level schedule

Definition 6.13 (Level-to-level schedule)

Let $s = (op(s), <_s)$ be an n-level schedule with layers $L_0, \ldots, L_n$ (in bottom-up order). The *level-to-level schedule* from L_i to L_{i-1} with $i > 0$, or L_i*-to-*L_{i-1} *schedule* for short, is a 1-level schedule $s' = (op(s'), <_{s'}$ with

1. $op(s')$ consisting of the L_{i-1} operations of s,
2. $<_{s'}$ being the restriction of the extended order $<_s$ to the L_{i-1} operations,
3. the L_i operations of s as the roots, and
4. the parent-child relationship identical to that of s.

So a level-to-level schedule narrows down our view of the entire object-model schedule to a pair of adjacent layers, with all ordering relationships of the original schedule being fully preserved. In terms of operation semantics, such as commutativity, we focus on the leaves of the resulting 1-level schedule, i.e., on L_{i-1}. The roots of the 1-level schedule, i.e., the L_i operations, serve as transactions of the 1-level schedule in that they denote the contexts for the L_{i-1} operations. In other words, the new roots tell us which new leaf operations belong together. Note that we implicitly generalize our initial definition of an object-model schedule, as the latter would actually require all leaves to be read/write operations in the page-model sense. This syntactic restriction will be dropped when we consider level-to-level schedules that are derived from a full-fledged object-model schedule.

As an example, Figure 6.7 shows the two level-to-level schedules that can be constructed from our initial example in Figure 6.1. The operation subscripts are given explicitly to highlight the connection to the original schedule; the new roots of the L_1-to-L_0 schedule are cast into mere placeholders of the form t_{ij}, as we will not consider their semantics in this specific 1-level schedule. Note that the L_2-to-L_1 schedule has partially ordered leaves, and the order is explicitly indicated by the directed arcs in the figure. This partial order is exactly the original schedule's partial order of the L_1 operations as derived from the ordering of their leaves (i.e., the original L_0 operations). For simplicity, we do not draw these arcs for the L_1-to-L_0 schedule, as this one still has totally ordered leaves.

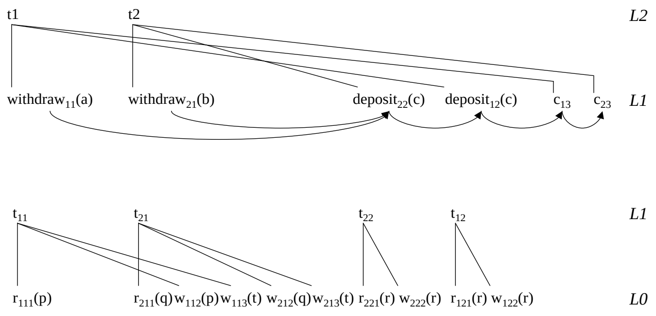


Figure 6.7: Level-to-level schedules constructed from the layered object-model schedule of Figure 6.1.

Now the obvious idea is to view each of the resulting level-to-level schedules as a flat object schedule and apply conflict serializability to each of them separately. To this end we need a conflict specification or, equivalently, its positive counterpart, a commutativity relation, for all operation types that belong to the same layer. As mentioned before, this cannot be derived automatically, say, from the code of the methods that belong to an encapsulated object, but intellectually defining a (possibly conservative) conflict/commutativity table does usually not pose any (unsurmountable) difficulties.

What many readers may now intuitively expect is that a layered schedule is tree reducible if (or possibly even if and only if) all its level-to-level schedules are conflict serializable when viewed as flat object schedules. This modular decomposition principle is formally captured in the following conjecture; as we will see in a short while, this conjecture is not true, but with some appropriate amendments it will point us towards the correct theorems.

Conjecture 6.14

Let s be an n-level object-model schedule. If for each i, $0 < i \leq n$, the L_i-to-L_{i-1} schedule derived from s is conflict serializable (CSR, see Chapter 3), then s is tree reducible.

In checking this criterion, a minor technical difficulty is that we may encounter conflicting but unordered operations in one of the level-to-level schedules, e.g., two withdraw operations on the same bank account. This is not the case in our example, but when such situations arise they are rather straightforward to rectify: as serial schedules have all operations ordered, an unordered conflict pair prevents a schedule from being equivalent to any serial schedule. In other words, unordered conflict pairs render a schedule non-serializable. In the example of Figure 6.7, however, both level-to-level schedules are conflict serializable: the L_1-to-L_0 schedule is equivalent to the serial order $t_{11} < t_{21} < t_{22} < t_{12}$, and the L_2-to-L_1 schedule is equivalent to $t_1 < t_2$ as well as $t_2 < t_1$. Intuitively, the serializability of the L_1-to-L_0 schedule is an argument for being able to prune the L_0 level, i.e., apply the tree pruning rule to the subtrees rooted at the L_1 operations of the original 2-level schedule. Then the ordering and the commutativity rules can be applied to the L_1 operations to arrive at either of the two possible serialization orders for t_1 and t_2.

The above argumentation works perfectly for our example. However, it is treacherous, and it turns out that there are cases where the conflict serializability of all level-to-level schedules does not imply tree reducibility. The problem is that reordering commutative leaves regardless of what their ancestors look like could have undesirable or even unacceptable "side effects" on these ancestors. Figure 6.8 shows such an example. In the example, we consider object methods f, g, and h where f and g are assumed to be in conflict when applied to the same object and h is assumed to commute with both f, g, and itself. The level-to-level schedule from L_2 to L_1 is conflict serializable, as each of t_1's two operations strictly precede, in terms of a total order, t_2's corresponding operations that lie in conflict with t_1; so the L_2-to-L_1 schedule is equivalent to $t_1 < t_2 < t_3$. In fact, t_3 can be arbitrarily ordered with regard to t_1 and t_2, as its operation h does not conflict with f or g. Between levels L_1 and L_0, on the other hand, we also observe a conflict serializable level-to-level schedule, equivalent to the serial order $t_{11} < t_{21} < t_{22} < t_{31} < t_{12}$. However, it is impossible to isolate all three subtrees, t_{31}, t_{12}, and t_{22}, by means of the commutativity rule; hence the schedule is not tree reducible due to part 1(c) of Definition 6.12.

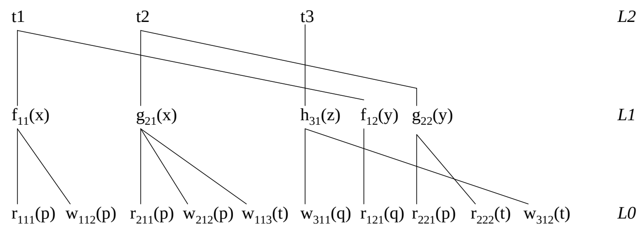


Figure 6.8: Non-tree-reducible layered object-model schedule with conflict serializable level-to-level schedules.

The decisive problem with the schedule of Figure 6.8 is that the serialization order for the L_1-to-L_0 schedule reverses the actual execution order of the two conflicting L_1 operations $f_{12}(y)$ and $g_{22}(y)$: $f_{12}(y)$ is completely executed before $g_{22}(y)$, but serialized the other way round as far as the underlying page accesses are concerned. This signals that there may be a major problem. If we knew that these two L_1 operations are serialized in the order with $g_{22}(y)$ preceding $f_{12}(y)$ we would no longer consider the L_2-to-L_1 schedule serializable, as this observation contradicts the conflict order on object x. The fundamental reason for this troublesome situation is that the serialization order for the L_1-to-L_0 schedule does *not* preserve the execution order at level L_1; that is, the level-to-level schedule from L_1 to L_0, albeit conflict serializable, is not order-preserving conflict serializable.

So we can obviously not completely modularize the concurrency control for layered schedules in that we consider level-to-level schedules in an absolutely separated, essentially "myopic" manner. But once we insist on order-preserving conflict serializability for each level-to-level schedules, we essentially add a "handshake" between successive level-to-level schedules by ensuring that the serialization order for the "lower" schedule is compatible with the execution order observed for the "higher" schedule. Indeed the following theorem holds:

OCSR level-to-level schedules imply tree reducibility

Theorem 6.15

Let s be an n-level object-model schedule. If for each i, $0 < i \leq n$, the L_i-to-L_{i-1} schedule derived from s is order-preserving conflict serializable (OCSR, see Chapter 3), then s is tree reducible.

Proof. Order-preserving conflict serializability from level 1 to level 0 allows us to apply the ordering and commutativity rules to isolate the subtrees rooted at level L_1 such that the execution order of L_1 is preserved. Then we can prune all operations of L_0 by the tree pruning rule. Now consider the new leaves of the pruned transaction forest (i.e., the operations at level 1). Any non-commutative operations among these new leaves must have been executed in total order, and because of the order-preserving serialization this original execution order is retained. So the outcome of pruning all operations of level 0 is a transaction forest whose new leaf ordering is compatible with the original ordering of the operations at level 1 in the sense that all "critical" orderings between non-commutative operations are included, some previously unordered but commutative operations may have become ordered, and the original ordering of some commutative operations may have been switched. Thus, since the original level-to-level schedule from level 1 to level 0 is assumed to be OCSR, this property also holds for the result of the first pruning round. Then the same argument can be iterated, and by induction on the number of levels, we see that we arrive at a total order of the transaction roots. □

The converse of the implication in the above theorem does not hold: there are schedules which are tree reducible without all their level-to-level schedules belonging to OCSR. Figure 6.9 shows an example. The L_1-to-L_0 schedule is not order-preserving conflict serializable. The only possible serialization would reverse the order of $h_{12}(y)$ and $h_{22}(y)$, contradicting their actual execution order. In this case, however, this does not cause any harm, as the two h operations are commutative so that any arbitrary serial order is acceptable.

As the above example shows, the order-preservation condition for all level-to-level schedules may sometimes be overly restrictive. What really matters is not order-preservation in general but to prevent that the serialization at one level reverses the actual execution order of conflicting operations of the next higher level. This unacceptable effect would be impossible if the non-commutative higher-level operations would lead to conflicting operation pairs also at all lower levels. The reason for this situation fixing our problem is that even non-order-preserving conflict serializability can never reverse the order of a conflict pair. We refer to this setting as *conflict-faithful* and define it formally as follows:

conflict faithfulness

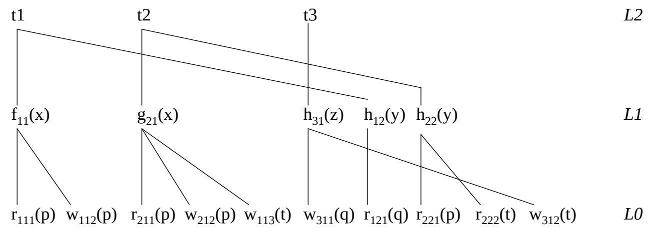


Figure 6.9: Tree-reducible layered object-model schedule with non-OCSR level-to-level schedule.

Definition 6.16 (Conflict faithfulness)

A layered object-model schedule $s = (op(s), <_s)$ is *conflict-faithful* if for each pair $p, q \in op(s)$ that are non-commutative and each $i > 0$ the corresponding sets of descendants with distance i to p and q contain at least one operation pair p', q' where p' and q' are descendants of p and q, respectively, and are in conflict.

Conflict faithfulness is a fairly natural property. What it boils down to is that some non-commutative operations at a semantically high level of abstraction must have conflicts on all lower levels of abstraction, where details such as storage representations are taken into account. For example, two withdraw operations on the same bank account must touch and, in fact, write common records and ultimately common pages. It is natural for low-level conflicts to become irrelevant at higher abstraction levels, but not the other way round. Proving that schedules are conflict-faithful is often fairly easy by studying properties of the underlying system layers rather than inspecting individual schedules. For example, all schedules that properly capture executions at the record and the page level of a database system must be conflict-faithful.

An alternative way of ensuring conflict faithfulness is to consider operations as conflicting only if they have at least one conflict at a lower level *and* are semantically non-commutative in that their return values and side effects are dependent on the order. So if operations are specified as generally non-commutative, but do

not exhibit a single lower-level conflict during their execution (as their behavior is dependent on the current states of the underlying objects), they would be considered as conflict-free. Obviously this condition would need to be checked for each schedule individually; from a system perspective, this would mean that the various subsystems across which a schedule extends (e.g., in a federated environment, see Chapter 1) need to report conflicts to their callers.

The nice property of conflict faithfulness is that it renders conflict serializability of all level-to-level schedules a sufficient condition for tree reducibility:

Theorem 6.17

A layered object-model schedule $s = (op(s), <_s)$ is tree reducible if it is conflict-faithful and all its level-to-level schedules are conflict serializable.

Proof. The proof is by induction on the number n of non-root levels. For $n = 1$, we consider a conventional single-layer flat-object-model or page-model schedule, and the claim trivially holds. Now assume that our claim holds for $n-1$ $(n > 1)$ non-root levels, and consider a schedule with n non-root levels. By the induction hypothesis, we can prune all subtrees rooted at level $n-1$ such that the resulting transaction forest preserves the original execution order of all non-commutative operations at level $n-1$. Because of this property and since the level-to-level schedule from level n to level $n-1$ is conflict serializable, we can isolate the operations at level n and once again prune subtrees (i.e., their children). What remains to be shown is that this pruning step must also preserve the execution order of any non-commutative operations at level n. So suppose that two such non-commutative level n operations, f and g with the original execution order $f < g$, are rearranged by the transformation steps for isolating the level n operations such that g precedes f in the resulting transaction forest. Now the property of conflict faithfulness tells us that f and g have children, say, f' and g', that are non-commutative at level $n-1$. Since the original execution order is $f < g$, f' must have preceded g' originally. Therefore, the (conflict serializable) serialization of the level-to-level schedule from level n to level $n-1$ must have preserved this ordering. So it is impossible to violate the original execution order of non-commutative operations at level n. $\square$

As examples for this sufficient condition for tree reducibility, consider again the two scenarios of Figures 6.8 and 6.9. In both examples, all level-to-level schedules are conflict serializable. The schedule in Figure 6.9 is conflict-faithful because the only L_1 conflict pair, $f_{11}(x)$ and $g_{21}(x)$, exhibits also page-access conflicts on page p. The argumentation for the schedule in Figure 6.8 is more subtle: here we would additionally have to consider the two L_1 operations $f_{12}(y)$ and $g_{22}(y)$ as non-conflicting simply *because* their execution did not produce any page-level conflicts.

So, although these operations may not commute in general, the executions in the particular context of the given schedule turn out to be commutative.

A final observation is that the conjunction of level-to-level serializability and conflict faithfulness is not a necessary condition for tree reducibility. The point is that conflicting higher-level operations may be allowed to run concurrently (i.e., unordered) as long as they are properly serialized at the underlying implementation levels. Such concurrency would not be allowed by a conflict serializable level-to-level schedule. Figure 6.10 shows a concrete example with conflicting, unordered record operations *fetch(x)* and *modify(x)*. The schedule is tree reducible to the transaction order $t_2 < t_1$.

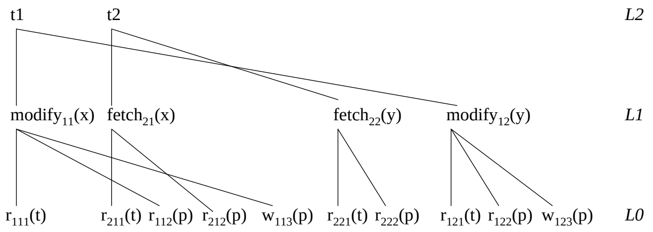


Figure 6.10: Tree reducible schedule with concurrent, conflicting L_1 operations.

6.6 Exploiting State-Based Commutativity

In this section we will discuss possible relaxations of general commutativity to further enhance concurrency. As this issue solely involves operations on the same object type, all previous considerations on the underlying operation invocation trees can be carried over without modification. Thus, for the ease of presentation, we will assume that operations on ADT (i.e., abstract data type) objects are already isolated and restrict the discussion to flat object schedules, sometimes not even bothering to show the isolated page-level subtrees underneath.

General commutativity is a powerful means for enhancing concurrency beyond what a pure page-level perspective would allow; however, it also has its limits. For example, the following totally ordered object-level schedule would be considered

non-admissible because the first two operations are in conflict with the ordering $t_1 < t_2$, whereas the second pair of deposit and withdraw operations is conflicting with the order $t_2 < t_1$ (based on the commutativity table from Section 6.3):

$$withdraw_1(x, 30)\ deposit_2(x, 50)\ deposit_2(y, 50)\ withdraw_1(y, 30)$$

In this example, we have stated the amounts of money withdrawn or deposited as explicit parameters (e.g., \$ 30 for the first operation). Knowing these additional parameters alone does not help much. However, in conjunction with knowledge about the original balance of the account (i.e., before t_1's first withdraw operation is invoked), we may be in a position to approve the above schedule as one that is equivalent to a serial execution. For example, if the original balance was known to be \$40 and the threshold for the overdraft protection was zero, then the first withdraw operation could be viewed as commutative with the two deposits, thus eliminating the conflict cycle and rendering the schedule equivalent to the transaction order $t_2 < t_1$. The key point is that this equivalence is given only in a specific state of the bank account against which the entire schedule is run. For example, if the initial balance of the account was \$20 only, the first withdraw operation would have signaled the overdraft protection and failed to obtain money, whereas it would be successful if it followed the first deposit operation. We refer to the first mentioned, positive case with the initial balance of \$40 as a *state-dependent commutativity* argument.

state-dependent commutativity

Definition 6.18 (State-dependent commutativity)

> Two operations p and q on the same object are *commutative in object state* σ if for all possible sequences of operations ω, the return parameters in the concatenated sequence $pq\omega$ applied to state σ are identical to those in the sequence $qp\omega$ applied to σ.

So from a state-dependent commutativity viewpoint, the sample schedule above is acceptable, whereas it cannot be allowed under general, state-independent commutativity. Unfortunately, exploiting state knowledge poses severe difficulties in a real system environment. First, the notion of a state is not easy to define, given that operations may have side-effects on more than one object, some of which may be observed much later. Second, even if we had a clean definition of object states, we would still need an algorithm for inspecting states. With encapsulated objects these methods may not be available, and even if they could, in principle, be added, they could be highly inefficient or even considered as a breach of the encapsulation and modularity (ultimately violating the information-hiding principle of good software engineering practice). With bank account objects, for example, we may internally use the *getbalance* method for testing the object state upon each withdraw method, but this could result in inefficiencies and may no longer be sufficient if the internal state of objects included also an audit trail of cash flows say in the form of a list of balance modifications. In fact, the latter situation may render withdraw and deposit operations again non-commutative in every possible state.

So rather than actually inspecting states, we restrict ourselves to observing only operation parameters, but we can extend this approach to observing result parameters as well. In the case of withdraw operations, the two possible outcomes — successful withdrawal or overdraft protection — would yield two different return values *ok* or *no*. Obviously, these return values are related to the object state against which an operation runs; so this provides a means for making particular facets of the object state visible without any additional steps. Commutativity arguments should then refer not just to operation types and their input parameters but also to the return values of the invoked operations.

This idea leads to the following definition of what we call *return-value commutativity* and what can be considered as an indirect kind of state-dependent commutativity that respects the object encapsulation. To this end, we will now denote operation executions in the form $f(\downarrow x_1, \ldots, \downarrow x_m, \uparrow y_1, \ldots \uparrow y_n)$ where the x_i are input parameters and the y_j output parameters, or in the simpler form $f(x_1, \ldots, x_m) \uparrow y$ when there is only a single output parameter, the *return value*.

return-value commutativity

Definition 6.19 (Return-value commutativity)

An operation execution

$$p(\downarrow x_1, \ldots, \downarrow x_m, \uparrow y_1, \ldots, \uparrow y_n)$$

is *return-value commutative* with an immediately following operation execution

$$q(\downarrow x'_1, \ldots, \downarrow x'_{m'}, \uparrow y'_1, \ldots, \uparrow y'_{n'}),$$

if for every possible sequence of operation executions α and ω such that p and q have indeed yielded the above return parameters in the sequence $\alpha\ p\ q\ \omega$, all operations in the sequence $\alpha\ q\ p\ \omega$ yield identical return parameters.

Note that the restriction of the considered sequences α is essential for enhanced concurrency: we consider only sequences α, and thus implicitly the resulting states, such that p and q yield the given return values when executed in the order $p < q$ right after α. This is a subset of all possible α sequences. For example, when

p is a $withdraw(x, 30) \uparrow ok$ operation execution, only α sequences are relevant that resulted in an account balance of at least \$30, since otherwise the withdrawal would have returned "no" rather than "ok". Further note that this subtle but important restriction of the implicitly observed initial states makes the notion of return-value commutativity asymmetric. The set of relevant α sequences for the ordered pair $p\ q$ may differ from that for the ordered pair $q\ p$. As an example assume $p = withdraw(x, 30) \uparrow ok$ as above and $q = deposit(x, 50) \uparrow ok$. For the execution order $p\ q$ the set of α has the above restriction, whereas for $q\ p$ we implicitly remove this restriction, and Definition 6.19 thus refers to all possible α sequences for deposits return "ok" in every possible state.

Return-value commutativity for a given object type can be stated in the form of a table, similar to general commutativity tables but with the difference that the entries refer to operation executions including return values rather than operation invocations with input parameters only. An example for the bank account object is given below (ignoring the getbalance operations for simplicity), where *withdraw* has two different possible return values and *deposit* always returns "ok". The rows of the table are the p operations of Definition 6.19, and the columns are the q operations. Note the asymmetry that we discussed above.

	$withdraw$ $(x, \Delta_2) \uparrow ok$	$withdraw$ $(x, \Delta_2) \uparrow no$	$deposit$ $(x, \Delta_2) \uparrow ok$
$withdraw(x, \Delta_1) \uparrow ok$	+	−	+
$withdraw(x, \Delta_1) \uparrow no$	+	+	−
$deposit(x, \Delta_1) \uparrow ok$	−	+	+

The entry for the two successful withdrawals, for example, has the following intuitive explanation: since both operations returned "ok" in the considered α sequences, there must have been enough money on the account to cover the sum of Δ_1 and Δ_2; hence reversing the order of the withdrawals would lead to the same result. The entry for the unsuccessful withdrawal and a subsequent deposit, on the other hand, must state non-commutativity, as the withdrawal could have become successful if the deposit preceded it (without changing anything in the prior history α).

The table for return-value commutativity could be further refined by adding conditions on the actual values of the input parameters. For example,

$$withdraw(x, \Delta_1) \uparrow ok$$

could be considered commutative with a subsequent $withdraw(x, \Delta_2) \uparrow no$ if $\Delta_2 > n\Delta_1$ and $\Delta_1 \geq x^{(0)}/m$ with n and m being constants such that $m \leq n$, where $x^{(0)}$ denotes the initial account balance (before the schedule begins) and we assume an overdraft protection threshold of zero. Namely, in this case we can infer that in the original schedule the value $x^{(0)}$ of the account balance before the first withdrawal must have been at least as high as Δ_1, but the resulting balance, $x^{(0)} - \Delta_1$ must have been smaller than Δ_2. All these inequalities together yield $x^{(0)} \leq m\Delta_1 < \frac{m}{n}\Delta_2$, and we can infer $x^{(0)} < \Delta_2$ because of $m \leq n$. So the second withdrawal operation would have been unsuccessful even if it was executed first. Such a kind of reasoning is, however, fairly involved. Therefore, we will not consider explicit constraints in a commutativity table any further, and will rather restrict ourselves to entries with plain + or − statements.

return-value commutativity for counters

The manipulation of account balances by withdraw and deposit operations is merely an example of the general concept of incrementing and decrementing *counters*. Counters arise in many highly relevant applications such as stock inventory where the quantity-on-hand of sales items is tracked, or reservation systems for airlines, hotels, rental cars, etc. where the numbers of available seats, rooms, and cars need to be maintained. So general considerations for highly concurrent counters surely pay off over a wide variety of applications. The approach is quite simple: counters are viewed as ADT objects with $incr(x, \Delta)$ and $decr(x, \Delta)$ as operations typically used in the operational transactions (e.g., for reserving seats on a flight) and an additional $getvalue(x)$ that will infrequently be used to read the actual counter value. If counters were absolutely unconstrained, then *incr* and *decr* would generally commute. However, in almost all applications, counters have lower bounds (e.g., zero in reservation systems) or upper bounds (e.g., the stock capacity in inventory management) or both. Since we lose general commutativity in such a setting but observe "uncritical" situations such as "plenty of seats still available" most of the time, state-dependent or return-value commutativity is the intriguing resort. In fact, both *incr* and *decr* can be designed and implemented in such a way that they either return "ok" or "no" depending on whether they were successful in modifying the counter or hit a lower or upper bound and then did not modify the counter at all.

Working out the corresponding return-value commutativity table is left as an (easy) exercise to the reader (see Exercise 6). Here we merely give an example of a highly concurrent schedule on objects x and y of type counter, both with a lower bound of 0 and an upper bound of 50. Figure 6.11 shows this example schedule. The actual, encapsulated and thus usually non-observable intermediate states of the counters are shown for clearer illustration. This layered schedule is tree reducible when return-value commutativity is allowed at the ADT level, namely, equivalent to the serial execution with t_1 preceding t_2, but it would not be feasible under general, state-independent, commutativity.

return-value commutativity for queues

Another practically important standard ADT for which exploiting return-value commutativity is feasible and may be worthwhile are FIFO, i.e., first-in-first-out *queues*. Such queues are of great benefit in implementing message handling systems

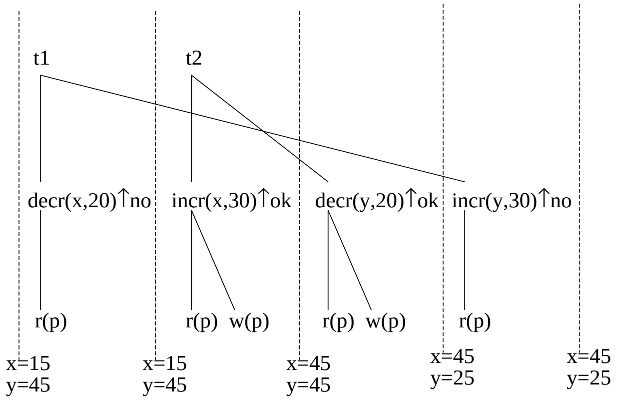


Figure 6.11: Example schedule on counter objects.

such as mail servers or even distributed workflow management systems as mentioned in Chapter 1. Queues provide operations for enqueueing entries (e.g., messages) at the tail of the queue and for dequeueing entries from its front. Transactional concurrency control is not necessarily required for messaging applications, but it can be of great benefit in order to ensure certain consistency guarantees such as preserving the order of a multiple-messages multicast to a number of recipients, and other things along these lines. It should be emphasized at this point that messaging systems often need to serve a very large number of clients concurrently and that the underlying queue objects may well become "hot spots" in terms of their update frequency. Therefore, a liberal, yet consistency-preserving concurrency control for queues may be crucial.

The two most important queue operations, *dequeue* (or *deq* shortly) and *enqueue* (or *enq*) do obviously not commute in general. However, when the queue already contains at least one entry, then the order of an enqueue and a subsequent dequeue operations does not matter in that the two different orders cannot be distinguished by the application. As it would again be problematic to directly inspect the actually encapsulated queue state for the purpose of higher concurrency, we rather enrich the two operations by appropriate return values: Dequeue returns "ok" if it finds the queue non-empty and removes the front entry, otherwise it returns "empty". Enqueue returns "ok" when it adds an entry to an already non-empty queue, otherwise it reports the special case of an initially empty queue by returning "one" (for a new queue length of one). With these modifications we can construct the following return-value commutativity table (where the trivial input parameters are omitted):

	$enqueue \uparrow ok$	$enqueue \uparrow one$	$dequeue \uparrow ok$	$dequeue \uparrow empty$
$enqueue \uparrow ok$	−	*impossible*	+	*impossible*
$enqueue \uparrow one$	−	*impossible*	−	*impossible*
$dequeue \uparrow ok$	+	−	−	−
$dequeue \uparrow empty$	−	−	*impossible*	+

The table contains some combinations that can impossibly occur and which are marked as such. Note that without the exploitation of return-value commutativity, all queue operations would be in conflict with each other, as both operations update the queue state. Thus, now allowing dequeueing and enqueueing on a non-empty queue to proceed without raising a conflict is already a great win in terms of the achievable concurrency of transactions.

Note that two enqueue operations are still non-commutative regardless of whether we attempt to exploit return-value information or not. The simple reason is that two entries enqueued in a specific queue will be dequeued in exactly this order; so a different order would later be observable. Here we simply touch on the essence of

a FIFO queue. Further improving the concurrency among enqueue operations can only be achieved by relaxing the FIFO semantics.

semi-queues

Such a relaxation is indeed possible by re-defining the dequeue operation such that it nondeterministically selects and removes an arbitrary entry from the queue. A relaxed queue ADT is sometimes called a ***semi-queue*** or "weak queue" in the literature. In fact, it could as well be regarded simply as a set (or bag) of entries since the ordering of entries is now irrelevant; in conjunction with messaging applications using a special name may be desirable. We leave a discussion of the return-value commutativity of semi-queues as an exercise (see Exercise 8).

6.7 Lessons Learned

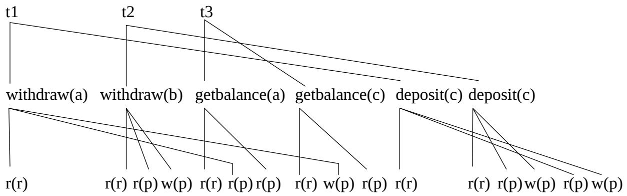
This chapter has made a major step in going beyond and generalizing the classical page model of transaction theory. We have introduced the technical details of a very powerful model for reasoning about transactions in semantically richer object-style settings, where an object may itself invoke operations on other objects to implement an operation provided at the object's interface. The resulting caller-callee relationships have been modeled as transaction trees, and the concurrent execution of several such trees has led us to a notion of transaction forests, the most general form of object-model histories.

Transaction trees can be specialized in various ways, the practically most relevant one being a layered form where objects correspond to the implementation layers of a system. An even more specialized case is the restriction to a single layer, with the underlying assumption that the operations of this layer are executed one at a time, so that it is indeed justified to disregard the implementation of an individual operation. This argument has guided us to the general principle of reasoning about a transaction forest being equivalent to a sequential execution of the transactions: we first need to show that concurrently executed operations can be isolated in the sense of being transformable into a sequential execution, and only then we can abstract from these operations' implementation details and prune their corresponding subtrees. The technical correctness criterion that captures this kind of reasoning is the notion of tree reducibility. In addition, we have introduced a couple of simpler criteria that provide us with sufficient conditions for a layered object-model schedule to be correct. These criteria will be the main tool for showing the correctness of several practically relevant protocols that we are going to present in the next chapter.

We have seen in this chapter that the classical notion of conflict, or non-commutative pair of operations, is still the key to our correctness reasoning, but it now captures more semantics of operations and needs to be applied at different levels of object-model transaction forests. We have also shown that the traditional state-independent form of commutativity can be further generalized for appropriate operations, bearing potential enhancements of the allowable concurrency.

Exercises

1. Consider the following 2-level schedule consisting of two funds-transfer transactions and a balance-lookup transaction. Assume that the getbalance operations do not commute with withdraw operations nor with deposits. Is this schedules tree reducible? If so, which is the equivalent serial execution? If not, how would the schedule have to be changed (as little as possible) so as to render it tree reducible?



2. Consider a data server with an SQL-style interface (see Chapter 1) and a database with a *Person* table whose rows contain a unique *Name* attribute and a *City* attribute. The server executes operations like

   - `Select * From Person Where City = c`, where `c` is a parameter, for looking up all persons that live in a given city, and
   - `Update Person Set City = c Where Name = n`, for recording that a given person has moved to a new city.

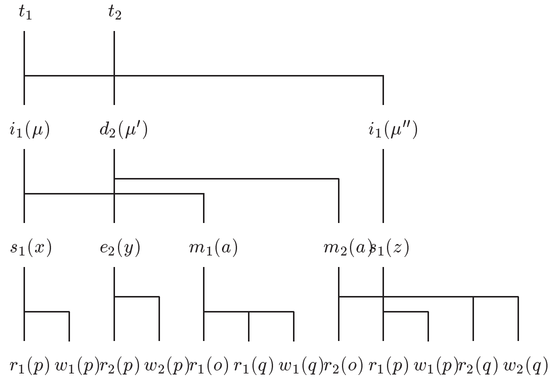
   Let us abbreviate these two operation types as *Select*(c) and *Update*(n, c), respectively. The server executes these operations by decomposing them into index lookups (*search*(*key*)), record fetches (*fetch*(*rid*)), record modifications (*modify*(*rid*)), and index maintenance steps (*insert*(*key*, *rid*) and *delete*(*key*, *rid*)). Assume that B$^+$-tree index structures exist for each of the two relevant attributes, and that both of these trees have depth two, i.e., consist of a root node and a leaf level. All of the mentioned record and index operations are finally transformed into page reads and writes.

   Now consider a transaction that finds all persons from two different cities, say Los Angeles and New York, and a second transaction that records the moving of a couple, say Liz and Jerry Smith, from Los Angeles to New York. Model the executions of both transactions as 3-level transactions, and discuss possible schedules for them. Give (non-trivial, e.g., non-serial) examples for 3-level schedules that are (a) tree reducible and (b) not tree reducible and thus not admissible.

3. Consider the two non-layered schedules of Figure 6.2. Are these tree reducible?

4. Consider again the example of Figure 6.6. Give the necessary transformation steps, using the commutativity and the tree pruning rules, for proving that the schedule is tree reducible.

5. Consider the layered schedule given below, where i means (record) insert, d stands for (record) delete, s means store, m stands for modify, and e is for erase. Which of its level-to-level schedules are CSR, which ones are OCSR, and which ones satisfy the conflict faithfulness property? Is the schedule tree reducible?



6. Consider a counter object with operations with lower and upper bounds as constraints. The counter interface supports three operations: *Increment* adds a specified value Δ to the counter and returns "ok" if the upper bound is not execeeded; otherwise it returns "no" and leaves the counter unchanged. Analogously, *decrement* subtracts a given value from the counter or returns "no" if the lower bound became violated. Finally, *getvalue* returns the counter value. Give a return-value commutativity table for these three operations. Discuss possible improvements of concurrency by taking into account explicit constraints among operation parameters.

7. Construct a sample schedule with operations on FIFO queues that demonstrate the higher concurrency that return-value commutativity can achieve over general commutativity.

8. Construct the return-value commutativity table for a semi-queue. Show, by means of sample schedules, that semi-queues allow higher concurrency than FIFO queues.

9. Consider the following execution of operations on an initially empty queue q where a, b, and c are entries added to the queue:

$$enq_1(q, a)\ enq_2(q, b)\ deq_3(q)\ enq_1(q, c)\ deq_3(q)$$

Discuss whether this schedule is serializable assuming (a) general commutativity, (b) return-value commutativity for queues, and (c) return-value commutativity for semi-queues with non-deterministic selection of entries by dequeue operations. In the last setting, which entries should be selected by the two dequeue operations of t_3 to produce an intuitively correct execution?

10. Design an abstract data type "mailbox" with operations like *send* (i.e., add a message to the mailbox), *receive* (i.e., extract a message from the mailbox), and so on. Discuss the operations' commutativity relation, for (a) general (i.e., state-independent) commutativity and (b) return-value commutativity. Devise an implementation of mailbox objects in terms of queues, records, and ultimately pages. Sketch the resulting transaction trees. Give an example of a concurrent execution, with as much concurrency as possible, for two transactions where one sends two messages to two different mailboxes and the other receives all messages from one of the two mailboxes.

11. Even if two operations do not commute in the strict sense, the different effects that result from the two possible orderings may be considered as "effectively equal" from an application viewpoint. One could then simply declare the two operations as *compatible* and treat them as if they were commutative. An example would be a deposit operation together with a new style of withdraw operation that allows overdrafting but claims a penalty of, say, $10 each time the account balance is below the specified threshold and money is withdrawn. So the "code" for this relaxed variant of withdraw would be:

```
relaxed withdraw (x, Δ):
  x.balance := x.balance - Δ;
  if x.balance < 0 then
    x.balance := x.balance - 10 fi;
```

Give a sample schedule with these operations that is conflict serializable relative to the compatibility of the operations, but is not conflict serializable under a strict interpretation of commutativity. Discuss the acceptability of the resulting (worst-case) situations from an application viewpoint, of both the bank and the customer.

Bibliographical and Other Remarks

The formal notion of an object-model schedule and the fundamental concept of tree reducibility is from Beeri et al. (1983, 1989), originally introduced under the name of "serializability for nested transactions". This seminal work also contains a detailed proof of Theorem 6.15. Informal ideas along these lines can be traced back to the highly inspiring, albeit rather semi-technical papers by Bjork (1973) and Davies (1973, 1978), Gray (1981), and Traiger (1983), under the names of *"spheres of control"* and "open nested transactions".

The practically most relevant special case of layered schedules, also known as *"multilevel transaction schedules"*, has been intensively investigated by Weikum and Schek (1984) and Weikum (1986, 1991), Moss et al. (1986), as well as Beeri, Schek, and Weikum (1988). The conceptual separation of logical and physical operations in a two-level concurrent system has been studied also by Bernstein et al. (1981, 1983) and Minoura (1984). The general case of non-layered schedules has been addressed by Muth et al. (1993) and Alonso et al. (1997, 1999a, 1999b).

The simpler case of flat object schedules with semantically rich operations beyond read/write operations has first been studied by Weihl (1983, 1988, 1989a, 1989b) as well as by Schwarz and Spector (1984). The subject is also treated in great detail in the textbook by Lynch et al. (1994). Interesting extensions can be found in Badrinath and Ramamritham (1992). The advanced form of exploiting even state-based or return-value commutativity dates back to Weihl's work as well, where our notion of return-value commutativity has been called "backward commutativity," see Weihl (1989b), and has also been further studied by Herlihy and Weihl (1991). The specific, but practically highly relevant example of increment and decrement operations on counters has been addressed especially in the work of Reuter (1982) and O'Neil (1986). An even more far-reaching approach to exploit both state-based commutativity and logical pre- and postconditions of operations has been proposed by Agrawal, El Abbadi, and Singh (1993) and further refined into a form of "assertional concurrency control" by Bernstein et al. (1998).