Selective Recovery for Long-Duration Transactions in Object-Oriented Database Management Systems

Axel Meckenstock
Detlef Zimmer

C–LAB*
Fürstenallee 11
D-33102 Paderborn
{axel|det}@c-lab.de

Rainer Unland

Universität-GH Essen
Schützenbahn 70
D-45117 Essen
unlandr@informatik.uni-essen.de

Abstract

Long-duration transactions play an important role in non-standard database applications like design environments or office automation. This paper deals with recovery for long-duration transactions. We start with a discussion of approaches presented in literature and show why these solutions are not sufficient. We then introduce *selective recovery*. The main advantage of this approach is that only that part of work is considered that is directly affected by a failure (first step). In order to guarantee consistency, however, it must be checked if further work is indirectly affected and therefore must be invalidated, too (second step). This is decided by evaluating information about operations performed on the data and information about the relationships between objects. The approach is especially tailored to object-oriented database systems.

Keywords: Non-Standard Transaction Management, Recovery, Object-Oriented Database Management Systems

1 Overview

Long-duration transactions play an important role in non-standard database applications like design environments or office automation. The inability of traditional transaction management [BHG87] to support long-duration transactions has already been observed by [Gra81] and was the motivation for the development of a number of non-standard transaction models (for an overview, see, e.g., [BK91] or [Elm92]).

This paper focuses on *recovery* for long-duration interactive transactions. The goal is to guarantee that as little work as possible is lost if a failure occurs without sacrificing consistency. However, this means to skip the simple and elegant concept of atomicity which is the base of the traditional transaction model.

The idea we present will be called *selective recovery*. With this approach, first that part of work is considered that is *directly* affected by the failure. Other work, even if performed after the erroneous work, will be preserved if possible. In order to assure consistency, however, it must then be checked whether this work is *indirectly* affected by the failure and, therefore, must be invalidated, too. This is decided by evaluating information about operations performed on the data and information about the relationships between objects.

Selective recovery is a new recovery technique for long-duration and interactive transactions. It also allows to support *cooperation* between transactions more effectively since it minimizes the negative ef-

*Cooperative Computing & Communication Laboratory (Siemens Nixdorf Informationssysteme AG, Universität-GH Paderborn)

fects of cascading rollback. The application area we assume are design applications [MZU94] which rely on object-oriented database management systems (OODBMS). The concept has been implemented in a prototype called PODEST ("**P**rogrammable **O**bject-oriented **Des**ign **T**ransaction System").

First, we discuss how other approaches deal with the problem of recovery and why the presented solutions are not sufficient. We then introduce our approach of selective recovery (section 3). In particular, we discuss several kinds of dependencies which have to be examined in order to achieve consistency. In section 4 we show how the model can be extended in various directions.

2 Related Work

Conventional recovery [HR83, BHG87] handles different kinds of failures, e.g., process crashes or disk failures. The techniques developed there (e.g., logging) can also be applied to long-duration transactions. However, logical transaction failures like exceptions within an application, erroneous user inputs or intentional rollbacks require more sophisticated approaches which are the topic of this paper.

A fairly traditional way which can also support long-duration transaction recovery is the so-called **partial rollback** [HR87]. Here, a savepoint is set to which a transaction can be rolled back in case of a failure. The problem with this approach is that the rollback invalidates *all* the work performed after the savepoint (figure 1). It is not possible to restrict the rollback to that part of work that is *really affected* by the failure and to preserve the other work.

From the papers on long-duration transactions only few discuss recovery issues in more detail. Instead, most of the work focuses on concurrency control.

Some approaches propose to **roll back single objects only** (e.g., [KLMP84, RRR+88]). This has the advantage that only the work

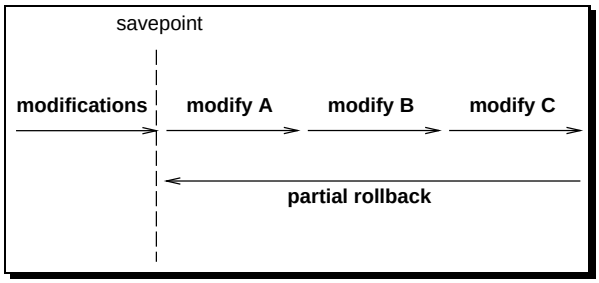

Figure 1: Partial Rollback

on the respective object is lost but no other work. However, objects are not simple entities that are independent of each other. First, objects are connected to other objects by relationships, especially in data models like the entity-relationship [Che76] or the object-oriented model [ABD+89]. Objects can also be related due to complex operations which read or write more than one object (figure 2). For these reasons, a rollback of single objects can lead to severe inconsistencies which, in these transaction models, have to be recognized and treated by the user. A dual problem is the **premature release of single objects** which was proposed to support cooperation between transactions [KSUW85]: because of the above reasons a transaction accessing a released object may get an inconsistent view of the database. When a transaction aborts which had already released some objects, either cascading rollback has to be performed, or the transaction can only roll back part of its work leaving all released objects valid which again may lead to an inconsistency.

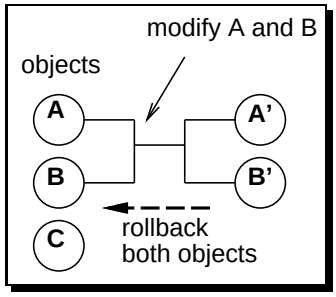

Figure 2: Rollback of Objects

The concept of **split transactions** [PKH88] divides the objects a transaction has accessed

into subsets and assigns them to independent transactions which can individually commit or abort. Thus, it is possible to roll back a part of the original transaction and commit the other part. The subsets have to be disjoint, i.e., it must again be taken into account that objects can be related in some way or that operations spanning more than one object were performed. The model does not explicitly provide mechanisms for checking these conditions.

The approach of [NRZ92] works by describing the consistency requirements for transactions through **grammar rules**. Single operations can be rolled back which is done recursively until, finally, the grammar rules are fulfilled again. Thus, consistency is guaranteed not only for normal work but also after recovery. While this is a very elegant approach it has the disadvantage that it requires a lot of specification effort. In order to guarantee consistency, the workflow and the constraints on data must be specified precisely. This specification, however, is very difficult for open-ended interactive transactions.

Some approaches (e.g., [KLS90, WS92]) propose **compensation** in order to undo certain operations while keeping the results of other operations. This works well as long as it can be guaranteed that future operations observe certain commutativity constraints, which is a very strong restriction. If this restriction cannot be guaranteed, as, e.g., in the *ConTracts* model [WR92], cascading compensation may occur. Here, a step (say S) within a long-duration transaction (ConTract) is annotated with a precondition. When the precondition gets invalidated because of some other event in a concurrent ConTract step S must be rolled back. However, if the ConTract has already performed further steps after S, the whole ConTract must be rolled back to that point, since there may be causal dependencies between steps (i.e., there is a control flow within the ConTract). Causality must be assumed because there is no further information about how steps are based on each other. Thus, recovery may lead to the loss of whole ConTracts or at least of large parts. Additionally, the specification of preconditions requires ConTracts to be pre-planned which is often not desirable for applications as the ones considered in this paper.

In the following we will show how our approach to recovery for long-duration transactions differs from the above solutions.

3 Selective Recovery

3.1 Main Idea of Selective Recovery

Recovery for long-duration transactions obviously requires to give up atomicity, i.e., to deal with a finer granularity than a transaction. In order to guarantee consistency this requires to evaluate additional information about how the objects resp. the operations of a transaction are related, an aspect not considered by most other transaction models. The additional information may be derived from existing information sources or may have to be specified explicitly.

Our approach offers two mechanisms. First, it permits to perform recovery in a selective way, i.e., to roll back only parts of a transaction, and still guarantees that the consistency constraints are obeyed (by evaluating additional information). Second, it permits to specify application-specific information in order to influence the recovery process. These two mechanisms are meant to be used by programmers who write tools (like editors) for design environments. Those programmers may then offer the end user, i.e., the designer, flexible ways to recover from errors. Technically, in our prototype PODEST we have developed a class library for an object-oriented programming language. Thus, the tool programmer can make use of design transaction management functionality in the same language used to implement the tool.

3.2 Basic Assumptions

We start with a simple model which will be extended later: a flat design transaction performs methods on objects of an OODBMS. The recovery technique we consider is the rollback of modifications[1]. The design transaction does not exchange data with other transactions (cooperation is deferred to section 4.3).

As an example we use a simplified scenario from the area of software design. We assume that the design objects manipulated within the software design environment are *programs*, *modules*, *procedure interfaces* and *procedure implementations* (figure 3). These objects are connected by relationships: a program `contains` several modules, a module `contains` procedure interfaces and procedure implementations, a procedure implementation `belongs-to` a certain procedure interface and may `use` other procedure interfaces. Thus, we have to consider complex objects (`contains`-relationships) and objects linked by arbitrary relationships.

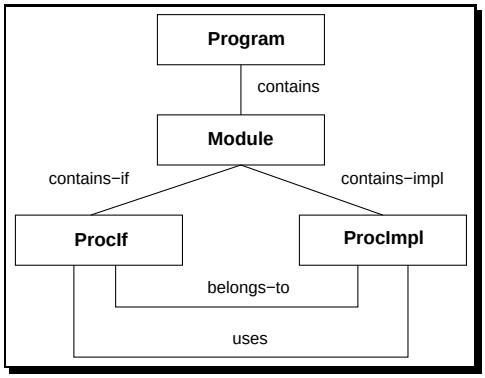

Figure 3: Example Scenario

The specific architecture of the OODBMS is not relevant for our discussion. However, the OODBMS is assumed to provide an interface integrated with an object-oriented programming language. In this paper, we use the C++-embedding as defined by the ODMG-Standard [Cat94]. Thus, the schema is defined by C++ classes (with some extensions), and methods are written as normal C++ code.

[1]Other possibilities are, e.g., the execution of alternatives or manual/automatic repair of inconsistencies.

3.3 Introduction to Selective Recovery

Now we discuss the concept of *selective recovery*. Most transaction models assume that an operation is always *causally dependent* on all earlier operations of the same transaction. For this reason, a rollback has to be performed completely or partially "from the end". Especially in interactive environments, the assumption of causality is too strong. A designer may execute a set of operations which are not necessarily dependent on each other.

Example 3.1: A designer changes two independent procedure interfaces `A` and `B` (figure 4). Then he adapts procedure implementation `C` to the changes of `A`. A rollback of procedure interface `A` invalidates procedure implementation `C`, but not procedure interface `B`. A rollback of procedure interface `B` does not invalidate procedure implementation `C`.

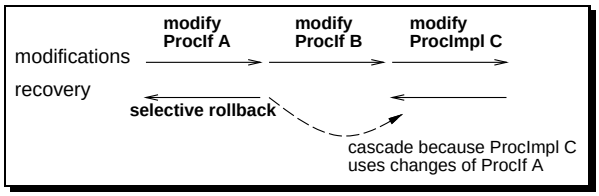

Figure 4: Causality of Modifications

The goal of selective recovery is to roll back only those parts of a transaction that are *really affected* by the failure, i.e., a transaction as a whole is no longer the unit of recovery. This leads to two questions:

- What are suitable **recovery units** instead?
- What **consistency** guarantees can be given under such circumstances?

Two **recovery units** are conceivable:

- **object-based recovery**
  The state of an object is rolled back to an earlier point in time. All work on the object performed after this point in time is lost. For this reason we also call this approach *partially-selective recovery*.
- **operation-based recovery**
  An operation is rolled back by executing an inverse (compensating) operation. Changes performed later on the affected objects are only lost if they (recursively) depend on the compensated operation. Thus, this approach can be described as *fully-selective recovery*.

These two perspectives seem to be relatively independent. However, as will be seen in the following, they have common characteristics. For example, object-based recovery has to consider the operations performed on the object. We will discuss object-based recovery in this section and shortly mention operation-based recovery in section 4.4.

In contrast to the classical transaction model where the transaction is the unit of recovery *and* consistency, the two recovery units mentioned above (objects and operations) cannot generally serve as **consistency units** because

- objects may be related to other objects,
- there may be dependencies between operations performed within a transaction.

Thus, it is necessary to perform a kind of cascading rollback *within* a transaction in order to reach a consistent state. Therefore, we consider the following two consistency requirements:

- **operation consistency**
  Whenever operations are executed which affect more than one object it should not be possible to roll back in such a way that parts of the operations survive while others are rolled back.

  Example 3.2: When the designer executes a global substitute for all procedures of a module in order to change a variable name, a rollback of a single procedure would lead to a semantic inconsistency.

- **schema consistency**
  It should not be possible to roll back objects selectively in such a way that the schema consistency is violated.

  Example 3.3: If a new procedure was inserted into a module, and the `belongs-to` relationship between procedure interface and implementation was established, a rollback of the interface only would violate referential integrity because the implementation still references the interface.

Conventional transaction management guarantees[2] these kinds of consistency by rolling back transactions completely or partially (through the assumption of causality). Other transaction models (section 2) offer more flexibility but ignore the above consistency requirements. Our approach permits a selective rollback of parts of a transaction and exploits further information in order to determine if more parts of the transaction are affected by the failure. In other words, the system decides if the assumption of causality is really valid. This is the fundamental difference to other transaction models (section 2).

Information about causality is partly present in the system anyway (e.g., information about relationships between objects). Additional information may be provided by the tool programmer explicitly. If the information is not precise enough, heuristic decisions are made. In such a case the designer has to correct potential inconsistencies manually which is mostly acceptable in interactive environments. However, the system uses as much information as possible in order to determine the causality automatically.

[2] if transactions are programmed correctly

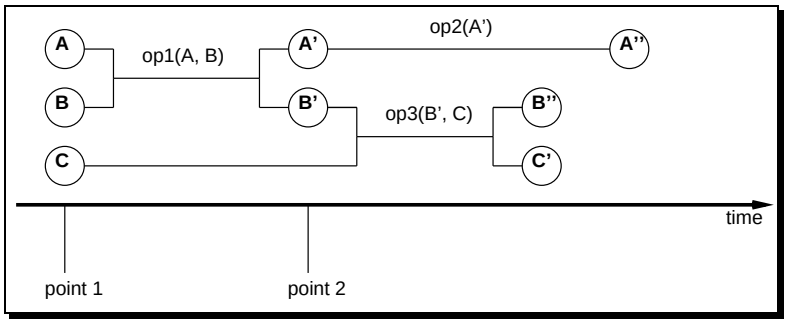

Figure 5: Example for Complex Operations

The information relevant for selective recovery is represented by **dependencies**. Dependencies are either uni- or bidirectional. In the first case, an object is dependent on another one but not vice versa. An object is invalidated if the object it depends on is invalidated. In the second case, invalidation takes place in either direction.

We now show in more detail how to determine dependencies and perform cascading rollback within a long-duration transaction.

3.4 Dependencies caused by Complex Operations

A long-duration transaction performs a sequence of operations. In an object-oriented environment, an operation is a method on an object which may call methods on the same or other objects. Whenever an operation touches more than one object, we call it a *complex operation* (e.g., `op1` and `op3` in figure 5). Such operations should be executed atomically, i.e., either completely or not at all. Thus, the execution of such an operation links together the participating objects in such a way that they have to be treated as a unit by the recovery management. This not only holds when the operation is executed (which is the conventional atomicity property), but also *after* it has terminated successfully.

Example 3.4: In figure 5, three modification operations (each of which forms an atomic unit) are performed. If `A"` is rolled back to point 1 (state `A`), `B"` must be rolled back, too (state `B`), which finally leads to a rollback of `C'` (state `C`). If `A"` is rolled back to point 2 (state `A'`), neither `B'` nor `B"` nor `C'` are concerned. If `B"` is rolled back to point 2 (state `B'`), `C'` must be rolled back, too, but `A"` can be kept.

When writing the methods, the tool programmer has to specify which operations form an atomic unit. Therefore, we introduce language elements to begin and to end a complex operation. Examples are listed in figure 6.

Complex operations may cause uni- or bidirectional dependencies between two or more objects.

- **unidirectional dependencies**
  An operation reads an object (which was modified by an earlier operation) and writes another object. If the first object is rolled back, the second object must be rolled back, too. However, if the second object is rolled back, the first object does not have to be considered, even if it was modified later on.

  Example 3.5: A procedure interface to which a new parameter was added by an earlier operation is read, and a procedure implementation using this interface is adapted by adding a new actual parameter. When the interface is rolled back, the implementation becomes invalid. A rollback of the implementation, however, is not relevant for the interface.

```
begin_complex_op();

proc_if.read_parms(...);
proc_impl.add_new_parm_to_proc_call(...);

end_complex_op();

-----------------------------------------------

begin_complex_op();

for all proc_impl contained in module
        // scan all procedure implementations
{
    proc_impl.substitute(name1, name2);
}

end_complex_op();
```

Figure 6: Unidirectional and Bidirectional Dependencies

- **bidirectional dependencies**
  An operation modifies several objects. A rollback of one of these objects must lead to a rollback of the other objects and vice versa.

  Example 3.6: A global substitute of a variable name by another name is performed on all procedure implementations of a module. A rollback of a single procedure implementation invalidates the other ones.

The system records the dependencies between the considered objects. Since it knows whether objects are read or written it can decide if the dependencies are uni- or bidirectional. In case of the rollback of an object to a point before the execution of a certain complex operation the system performs cascading rollback of all objects which are dependent on the object because of this complex operation. The additional logging and the determination of dependent objects in case of selective rollback are the main differences to conventional recovery.

Complex operations may also modify relationships between objects. Since a partial modification of a relationship may lead to schema violations, these operations, by default, have to be treated in an atomic way. However, there may be further dependencies caused by relationships which will be discussed next.

3.5 Dependencies caused by Object Relationships

The schema permits to define different kinds of relationships between objects. The most important one is the `contains`-relationship which leads to the construction of complex objects (figure 7).

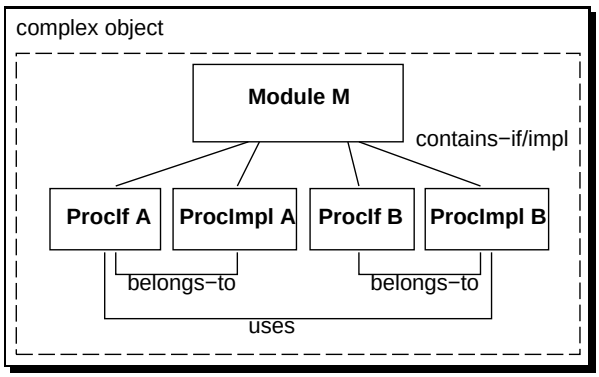

Figure 7: Example for a Complex Object

If an object is rolled back that contains other objects, it makes sense to roll back the inner

```
Module::add_procedure(procIf, procImpl)
{
    contains-if += procIf;              // set the relationship
    set_dependency(REL-UNIDIRECTIONAL,
                   this, procIf);       // set the dependency
    contains-impl += procImpl;          // set the relationship
    set_dependency(REL-UNIDIRECTIONAL,
                   this, procImpl);     // set the dependency
    procImpl.set_belongs_to(procIf);
}

ProcImpl::set_belongs_to(procIf)
{
    belongs-to = procIf;                // set the relationship
    set_dependency(REL-BIDIRECTIONAL,
                   this, procIf);       // set the dependency
}
```

Figure 8: Example for Specifying Dependencies

objects, too. This behaviour may also be desirable for arbitrary relationships which combine objects to logical units. Thus, it must be specified which relationships lead to a treatment of objects as a unit.

Relationships can be the source of unidirectional and bidirectional dependencies:

- **unidirectional dependencies**
  The semantics of the relationship is asymmetric, i.e., it holds in one direction but not in the other one.

  Example 3.7: If a procedure implementation `uses` a certain procedure interface, a rollback of the interface may invalidate the modifications of the implementation. On the other hand, a rollback of the implementation does not influence the interface because the `uses`-relationship is asymmetric. The same holds for the `contains`-relationship: if a module containing procedures is rolled back, the procedures should be rolled back, too, but not the other way round.

- **bidirectional dependencies**
  The semantics of the relationship is symmetric, i.e., it holds in both directions.

  Example 3.8: The relationship `belongs-to` requires procedure interface and implementation to be consistent, e.g., with respect to the number of formal parameters. Thus, if a transaction modifies both objects, the relationship may be taken as a hint to always roll back both objects.

The specification of dependencies is done on the language level, again. The tool programmer has to set the desired dependencies whenever relationships are inserted or deleted. This is done by inserting special calls into the code (or more conveniently by overloading the operators dealing with relationships). For example, in figure 8 the relationship `contains-if` and `contains-impl` between a module and a new procedure interface/implementation are set, leading to unidirectional dependencies. Afterwards, the relationship `belongs-to` between procedure interface and implementation is set, leading to a bidirectional dependency.

When the system executes an object-based rollback, it uses the specified dependencies in order to determine which further objects have to be rolled back. In case of a module, it will roll back all contained procedure interfaces and implementations, too. In case of a procedure interface, it also rolls back the procedure implementation and vice versa.

The above solution allows the tool programmer to define dependencies in a dynamic way. For example, the current state of a certain object may be evaluated. The tool programmer may also integrate an interactive interface, e.g., in order to let the designer define the relevant objects dynamically.

For complex objects, the question arises how to treat common subobjects of overlapping complex objects (e.g., two programs contain the same module, figure 9). Depending on the semantics of the complex object, both alternatives are meaningful: When one complex object is rolled back, the common subobject (if it was modified by the transaction) is rolled back, too, or the common subobject is left intact[3]. Since the system cannot determine this, the tool programmer has to specify how this situation should be handled (by setting appropriate dependencies).

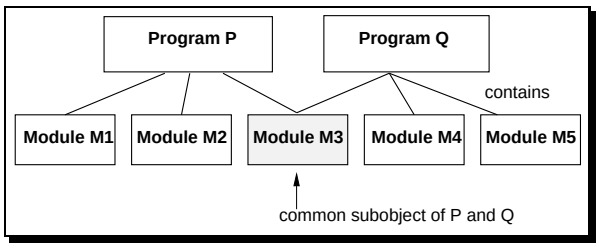

Figure 9: Example for Overlapping Complex Objects

To consider the schema is relevant if objects are not already related by complex operations. This is especially important for interactive environments like editors where a designer executes a sequence of simple operations which cannot be recognized automatically as belonging together.

Example 3.9: The designer modifies some procedure interfaces and some implementations which `use` them. From these operations, the system has no information about how the modifications belong together semantically.

[3]Note that this problem is similar to the problems of deleting or copying complex objects with shared subobjects.

In order to derive this information, it is again possible to set a dependency whenever a certain relationship (like the `uses`-relationship) exists between the corresponding objects.

3.6 Treating Dependencies

Whenever a rollback of an object is caused by some event, the system analyzes which other objects are dependent on this object. Both criteria, i.e., complex operations and object relationships, are checked. This is done in a transitive way, i.e., a cascading rollback of an object may cause further objects to be invalidated and so on. Thus, the main algorithm for selective recovery (which is implemented within the class library) looks as sketched in figure 10 (assuming the state of the objects was saved before in a suitable way).

In the appendix (section 6) we describe a complete application scenario using the approach of selective recovery.

3.7 Implementation Aspects

As described before, the specifications of complex operation dependencies and object relationship dependencies are done on the language level by adding special code (e.g., beginning a complex operation). This is useful for two reasons:

- Typically, the underlying OODBMS only provides generic methods to manipulate objects or pages but does not maintain user-defined methods. Thus, for dealing with the latter, it is necessary to work on the language level. Predefined methods (e.g., adding a relationship) are handled automatically.
- A specification on the language level permits the tool programmer to define application-specific dependencies. An alternative to this explicit way would be to extend the schema description language. This, however, would not permit an individual handling of single instances of

```
Object::rollback(earlier-state)
{
    if object already rolled back to earlier-state // avoid cycle
    {
        return;
    }

    roll back object to earlier-state;

    for all dep_obj related by complex operation dependencies
    {
        dep_obj.rollback(earlier-state);
    }

    for all dep_obj related by object relationship dependencies
    {
        dep_obj.rollback(earlier-state);
    }
}
```

Figure 10: Main Algorithm for Rolling Back an Object

a class. An extension of our approach is to exploit certain information from the schema automatically, e.g., by assuming that a `contains`-relationship always causes a dependency.

The integration of selective recovery into the transaction management can be done in two ways: Either the modifications are (partly) integrated into the transaction management of the OODBMS. This requires availability of the source code of the database management system, but permits efficient and safe access to internal information (e.g., the log). The other possibility is to work on top of the OODBMS, i.e., on the language level. Then the management of long-duration transactions has to be implemented based on the short transactions provided by the OODBMS. This implies disadvantages regarding efficiency and safety, but is easier to implement. It is also more portable if a standard database interface is used.

For these reasons, the latter approach is taken by our prototype implementation PODEST. PODEST provides (as a class library) classes for *design objects* and *long-duration transactions* (figure 11). The latter are further divided into *design transactions* for modeling design tasks and *tool transactions* for realizing tools. For the purpose of implementing certain protocols (e.g., two-phase locking), a *protocol* class is used.

The class library defines the basic recovery and concurrency control algorithms. For recovery, mechanisms for total, partial and selective rollback (e.g., the `rollback` method described above) and the commit handling are provided. For concurrency control, protocols like two-phase or non-two-phase locking and flexible lock modes are offered.

In order to consider object-specific information, the protocols access the design object class which describes a default behaviour of design objects. Every design object implemented by the application must be derived from this design object class. The default behaviour may be changed by specialization.

By deriving subclasses of the protocol and transaction classes, transaction types with application-specific concurrency control and recovery mechanisms can be realized. Transactions of different types can be combined to form a heterogeneous hierarchy. For example, within a hierarchy conventional protocols (guaranteeing serializability) and more flexible

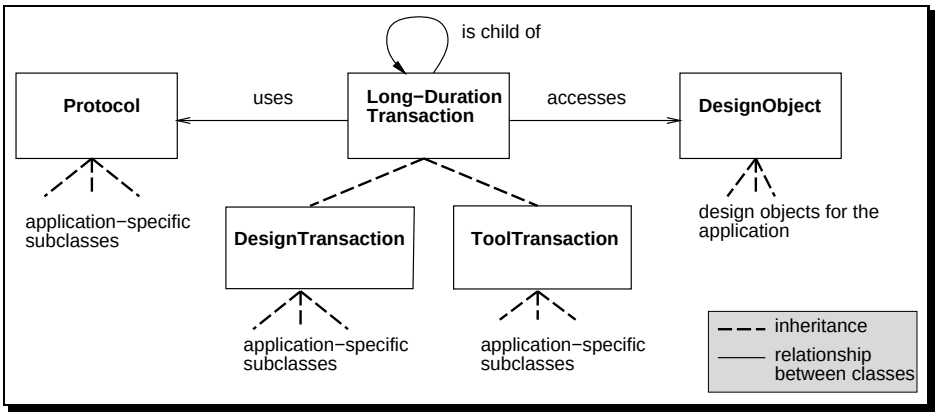

Figure 11: Class Hierarchy

protocols (permitting cooperative work) can be combined, depending on the requirements of a project or subproject.

The approach exploits object-oriented concepts for realizing a very flexible transaction system which is integrated into an object-oriented programming language (in our case C++ [Str86]) and an ODMG-compliant OODBMS (C–LAB's implementation *OpenDM* [RBB+95, Cad95]). The approach also uses concepts from the transaction toolkit [US92].

4 Extensions

4.1 Further Consistency Requirements

We now discuss further aspects and extensions of selective recovery. First, it may be useful to specify arbitrary consistency constraints on objects. Examples are cardinality constraints for the relationships between objects or rules for the construction of derived data. Consistency constraints have to be enforced whenever modifications are done, either by preventing illegal modifications or by making automatic corrections. This enforcement has to be done in case of recovery, too. Thus, consistency constraints may either lead to cascading rollback of other objects or to the execution of automatic reactions to correct inconsistencies.

Example 4.1: If a consistency constraint requires that a graphical software specification is always consistent with the textual source code, a rollback of the specification may either lead to a rollback of the source code or – if possible – to an automatic regeneration.

Consistency constraints may also be bidirectional (i.e., the mutual consistency of several objects is specified by a constraint) or unidirectional (i.e., the consistency of one object depends on the state of another object but not vice versa).

4.2 Releasing Objects

Some approaches (e.g., [KSUW85]) suggest to permit a long-duration transaction to release single objects (figure 12) such that other transactions can access them before the end of the transaction (which in fact is a prerequisite for cooperation).

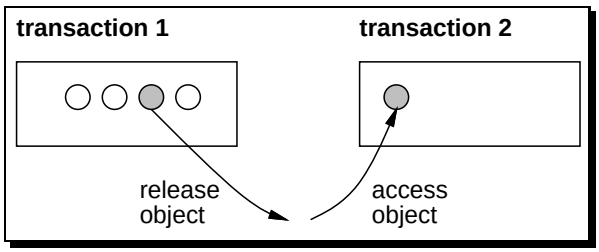

Figure 12: Releasing Objects

The release of objects can be handled in two ways:

- The transaction unlocks the object, but keeps the right to roll back its modifications. This is the traditional way applied, e.g., by the non-strict two-phase locking protocol [KMSL90]. When the transaction aborts, all modified objects – including those which were released – have to be rolled back in order to guarantee atomicity. This can cause cascading rollback of other transactions.
- It is often useful to release and commit immediately parts of the work of a transaction in order to make it available to other transactions without the risk of cascading rollback. Thus, the transaction decides that this work is already in a stable state and gives up its right to abort these objects. This effect may also be achieved by splitting a transaction and committing one of the resulting transactions (cf. section 2).

When single objects are released and committed (we call it *selective commit*), the same problems occur as with selective rollback: objects which belong together because of some dependencies (e.g., due to complex operations) must be treated as a unit. Thus, the release of one object may have to be accompanied by the release of other objects (if that is not possible the release is prevented). Note that certain unidirectional dependencies are handled just the opposite way as in the case of rollback: if an object is released, objects it depends on are released, too.

Example 4.2: If a transaction has read an interface and has adapted an implementation to use some new features of the interface, the implementation must not be released if the corresponding state of the interface was not released before. On the other hand, the interface can be released without the implementation which makes sense, e.g., if other transactions are meant to adapt further procedures to the new interface.

For complex objects, it must again be defined by the tool programmer whether common subobjects are released together with the containing object or not. In most cases, the first alternative is meaningful since the containing object in some way *uses* the subobject.

4.3 Nested and Parallel Transactions

Selective recovery can also be applied to nested and parallel transactions. After rollback has been performed within a (sub-)transaction, the effects on other transactions (parent, children, siblings or arbitrary transactions) are examined. If objects are rolled back which were accessed by other (sub-)transactions afterwards, a cascading rollback occurs. However, in contrast to the traditional approach where complete transactions are rolled back, selective recovery can be applied to the other transactions, too. Thus, loss of work is minimized even in case of a cooperation between transactions. Consequently, selective recovery is suitable for cooperative environments.

Example 4.3: If a transaction has read an interface which was implemented by another transaction, a rollback of the latter invalidates the interface. Thus, the transaction has to rollback all the modifications which rely on this interface. Selective rollback allows the transaction to keep other changes which are not dependent on the interface.

4.4 Operation-based Recovery

We already mentioned that operation-based recovery is an alternative to object-based recovery. While we cannot discuss this aspect in detail in this paper, we can point out that similar problems occur: if an operation of a transaction is compensated by an inverse operation, there can be cascading effects on later operations of the same or other transactions (figure 13).

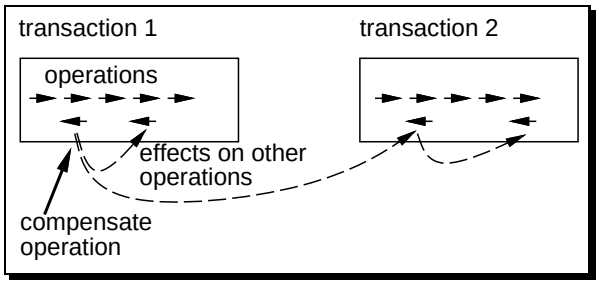

Figure 13: Compensation of Operations

In order to avoid such effects, most approaches (e.g., [WS92, MRKN92]) only allow subsequent operations which are commutative with the compensated-for operation resp. the compensating operation [KLS90] (a restriction that is quite impractical for the environments we consider). [WR92] allow arbitrary operations and apply cascading compensation if necessary. In any case, later operations of the *same* transaction may be causally dependent on one of the compensated-for operations and have to be rolled back, too.

Example 4.4: If a transaction reads procedure A and modifies procedure B, these operations are commutative, since they touch different objects. However, the modification of B may rely causally on the read operation and may thus have to be invalidated if A's modification is compensated.

By using the concepts we have described, we can optimize this case (cf. [MUZ96]). Compensation can then be done in a selective way, i.e., the affected operation is compensated and the effects on later operations are analyzed by checking for the dependencies described in section 3.

5 Conclusion and Future Work

In this paper we have discussed the problem of recovery for long-duration transactions. If a total or partial rollback of a transaction is not desirable, the concept of selective recovery, introduced in section 3, is a suitable alternative. Some approaches in literature already permit a selective rollback of single objects, but have the disadvantage that they ignore the dependencies between objects resp. operations.

The concepts presented in this paper rely on state-based rollback of single objects. We distinguish two kinds of dependencies, caused by complex operations and by object relationships. These dependencies are used to perform cascading rollback *within* a transaction which is necessary to achieve consistency after a selective rollback of single objects.

We briefly presented our prototype PODEST which implements our ideas on basis of the ODMG-compliant OODBMS *OpenDM* [RBB+95, Cad95]. Furthermore we gave some hints how our ideas can be extended. In particular, we discussed the problems of selective commit of objects, cooperation between transactions and operation-based recovery (compensation). The last point is a very interesting issue to be investigated further since compensation in general seems too complex to be used as a general technique. However, compensation might show advantages in special situations.

Currently, we are performing some studies to get a better understanding of the advantages and disadvantages of our approach.

6 Appendix: Application Scenario

We sketch a scenario based on the schema from figure 3. A programmer implements an editor for modules and procedures. The editor uses the class `ToolTransaction` from the class library. The classes for `Module`, `ProcIf` and `ProcImpl` are derived from the class `DesignObject`. At the user interface, the editor provides – among others – the following commands (e.g., within a menu):

```
create-module
remove-module
create-proc
remove-proc
modify-proc
use-proc-if
replace-all
```

The commands (each defined as a complex operation) handle the three kinds of objects and the relationships defined in the schema, e.g., the relationship `belongs-to` and the `contains` relationships. The command `use-proc-if` establishes a `uses` relationship between a procedure interface and another procedure implementation. The command `replace-all` replaces a certain text by a new one in the module and all its procedures.

The editor commands call methods on the three design object classes `Module`, `ProcIf` and `ProcImpl` which are omitted here. The programmer also sets dependencies (figure 8) between the objects of the three classes: a unidirectional dependency between a `Module` and all its procedure interfaces and implementations, a bidirectional dependency between a `ProcIf` and its `ProcImpl`, and a unidirectional dependency between a `ProcImpl` and all `ProcIf` `used` by the `ProcImpl`.

Now a designer is working with the editor in an interactive way (figure 14). He calls editor commands on modules `A` and `B` with procedures `A1`, `A2`, `B1` and `B2` (we add a suffix `impl` or `if` for implementation resp. interface.). The commands are called within a single long-duration transaction.

```
create-module A
create-proc A1
create-proc A2
use-proc-if A1.impl A2.if
create-module B
create-proc B1
create-proc B2
use-proc-if B1.impl A2.if
----------------------
SAVE STATES OF OBJECTS
----------------------
modify-proc A1
modify-proc A2
modify-proc B1
modify-proc B2
```

Figure 14: Editor Session

At this point of the transaction, the designer finds an error and decides to roll back a certain object to the saved state. The recovery algorithm (figure 10) exploits the information provided by the dependencies and rolls back other objects, too. Some typical cases show the effects of rolling back a certain object. Behind the arrow, we list the objects which are rolled back by the algorithm and the reasons for the decisions:

- **roll back** `A1.if` →
  roll back `A1.impl` because of `belongs-to`.
  (`uses` is not relevant in this direction, i.e., `A2.if` is maintained.)
- **roll back** `A2.impl` →
  roll back `A2.if` because of `belongs-to`.
  roll back `A1.impl` because of `uses`.
  roll back `A1.if` because of `belongs-to`.
  roll back `B1.impl` because of `uses`.
  roll back `B1.if` because of `belongs-to`.
- **roll back** `A` →
  roll back `A1.if`, `A1.impl`, `A2.if`, `A2.impl` because of `contains`.
  roll back `B1.impl` because of `uses`.
  roll back `B1.if` because of `belongs-to`.
- **roll back** `B` →
  roll back `B1.if`, `B1.impl`, `B2.if`, `B2.impl` because of `contains`.

Now we add a last operation `replace-all A` to the above scenario. If the designer rolls back `A1.if`, `A2` has to be rolled back, too, because `replace-all` is a complex operation. Otherwise, the replaced text would remain in `A2`, but not in `A1`.

Note that with partial recovery, a rollback of `A1.if` leads to a rollback of all the four procedures in *any* case. This shows the advantages of selective recovery.

References

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proc. DOOD*, pages 1–18, 1989.

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

[BK91] N.S. Barghouti and G.E. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3):271–317, September 1991.

[Cad95] Cadlab. *OpenDM System Overview*, November 1995.

[Cat94] R.G.G. Cattell, editor. *The Object Database Standard ODMG-93 (Release 1.1)*. Morgan Kaufmann, 1994.

[Che76] P.P. Chen. The Entity-Relationship model - Towards a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[Elm92] A.K. Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

[Gra81] J. Gray. The Transaction Concept: Virtues and Limitations. In *Proc. VLDB*, pages 144–154, September 1981.

[HR83] T. Härder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[HR87] T. Härder and K. Rothermel. Concepts for Transaction Recovery in Nested Transactions. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 239–248, 1987.

[KLMP84] W. Kim, R. Lorie, D. McNabb, and W. Plouffe. A Transaction Mechanism for Engineering Design Databases. In *Proc. VLDB*, pages 355–362, August 1984.

[KLS90] H.F. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Proc. Conf. on Very Large Data Bases*, pages 95–106, August 1990.

[KMSL90] R. Kröger, M. Mock, R. Schumann, and F. Lange. RelaX - An Extensible Architecture Supporting Reliable Distributed Applications. In *9th Symp. on Reliable Distributed Computing, Huntsville*, pages 156–164, October 1990.

[KSUW85] P. Klahold, G. Schlageter, R. Unland, and W. Wilkes. A Transaction Model Supporting Complex Applications in Integrated Information Systems. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 388–401, May 1985.

[MRKN92] P. Muth, T.C. Rakow, W. Klas, and E.J. Neuhold. A Transaction Model for an Open Publication Environment. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 159–218. Morgan Kaufmann, 1992.

[MUZ96] A. Meckenstock, R. Unland, and D. Zimmer. Rolling Back in a Selective Way – an Approach to Recovery for Interactive and Long-Running Transactions. In *Proc. 2nd*

Int'l Baltic Workshop on DB and IS, June 1996.

[MZU94] A. Meckenstock, D. Zimmer, and R. Unland. A Configurable Cooperative Transaction Model for Design Frameworks. In *Proc. 4th Int'l Workshop on Electronic Design Automation Frameworks (EDAF)*, pages 11–20, November 1994.

[NRZ92] M.H. Nodine, S. Ramaswamy, and S.B. Zdonik. A Cooperative Transaction Model for Design Databases. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 53–85. Morgan Kaufmann, 1992.

[PKH88] C. Pu, G.E. Kaiser, and N. Hutchinson. Split-Transactions for Open-Ended Activities. In *Proc. VLDB*, pages 26–37, August 1988.

[RBB+95] E. Radeke, R. Böttger, B. Burkert, Y. Engel, G. Kachel, S. Kolmschlag, and D. Nolte. Efendi: Federated Database System of Cadlab. In *Proc. ACM SIGMOD*, page 481, May 1995.

[RRR+88] S. Rehm, T. Raupp, M. Ranft, R. Längle, M. Härtig, W. Gotthard, K.R. Dittrich, and K. Abramowicz. Support for Design Processes in a Structurally Object-Oriented Database System. In *Proc. 2nd Int'l Workshop on Object-oriented Database Systems*, pages 80–97, 1988.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986.

[US92] R. Unland and G. Schlageter. A Transaction Manager Development Facility for Non Standard Database Systems. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 399–466. Morgan Kaufmann, 1992.

[WR92] H. Wächter and A. Reuter. The ConTract Model. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 219–263. Morgan Kaufmann, 1992.

[WS92] G. Weikum and H.J. Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 515–553. Morgan Kaufmann, 1992.