2

A Configurable Cooperative Transaction Model for Design Frameworks

Axel Meckenstock
Detlef Zimmer

CADLAB*
Bahnhofstr. 32
D-33102 Paderborn
{axel|det}@cadlab.de

Rainer Unland

Westfälische Wilhelms-Universität
Grevener Str. 91
D-48159 Münster
unlandr@uni-muenster.de

1 Introduction

Traditional transaction management [Gra81] which supports concurrent work and failure handling (recovery) based on the *ACID*-properties[1] [HR83] plays a significant role in database management systems. In the field of design frameworks this concept is relevant, too. However, the set of requirements for transaction management in this area is much more diverse.

In the conventional model, **concurrency control** is done by executing transactions in an isolated way. Design processes, however, require different degrees of **cooperation** in order to support collaborative work. **Recovery**, which is traditionally handled by rolling back transactions, should consider that design activities are of long duration and that work losses should be minimized. **Consistency** is handled implicitly in the conventional model. In design environments, explicit consistency specifications and different levels of consistency should be supported.

The idea of design frameworks is based on the observation that "hard-wired" services are not flexible enough to meet the various requirements of applications. Instead, a framework should offer a high degree of configurability. This idea can also be applied for transaction management. Thus, we propose a **highly configurable transaction model** permitting the definition of different **transaction types**. Hereby, a heterogeneous transaction hierarchy can be formed [US92, MUZ94]. The transaction types can be assigned different characteristics with respect to concurrency control, cooperation, recovery and consistency management. This is done by defining **protocols** that are used by the corresponding transactions. It is the task of the framework administrator to define suitable transaction types fulfilling the application requirements. Framework users and/or applications needing transactions with certain characteristics can choose from the set of transaction types and need not be bothered with details of transaction management.

*Joint R&D Institute University-GH Paderborn / Siemens Nixdorf Informationssysteme AG.
[1]Atomicity, Consistency, Isolation, Durability

F. J. Rammig et al. (eds.), *Electronic Design Automation Frameworks*
© Springer Science+Business Media Dordrecht 1995

We shall start with a short review of related work. In section 3 we present the basic concepts of our transaction model. Section 4 describes the main idea, i.e., the configurability of the model. We conclude with a short summary and an outlook.

2 Related Work

Early approaches to design transaction management have concentrated on certain aspects. For example, [KLMP84] introduced the concept of workspaces and *checkout/checkin*, [KSUW85] added version management and some cooperation primitives, and [KKB88] presented a model consisting of four transaction types and defined concurrency control protocols permitting cooperation. [Ioc89] discusses recovery techniques in workstation-server environments for several design transaction models. [NRZ92] uses grammars as a programmable correctness criterion for cooperative transaction hierarchies.

A more recent approach, *Concord* [RMH+94], especially deals with cooperation. Although it uses similar notions of transactions as our approach, it differs, e.g., in the way the operations *checkout/checkin* are handled. Another model developed within the *JESSI Common Framework* project [BS94] describes primitives for design transactions on top of an object-oriented database system. Both approaches do not support the concept of typing of transactions.

The main benefit of our model lies in the ability to configure transactions. Hereby, heterogeneous transaction hierarchies can be built that satisfy various requirements of applications. This heterogeneity also allows to combine the best-suited concepts from other transaction models. Furthermore, the model is supposed to integrate the different aspects of transaction management, in particular concurrency control, cooperation, recovery and consistency management. In this way, we continue and generalize the *transaction toolkit* approach [US92].

3 The Transaction Model

3.1 Overview

In this section we present the basic concepts of our transaction model. In particular, we sketch three notions of "transactions" which is necessary since this term is overloaded in literature.

To illustrate our presentation we give a simple example that will be used throughout this paper. A chip design project has the task to build an *arithmetic-logical unit* (*ALU*). The task is subdivided into the *design* and the *simulation* of the *ALU*. The design of the *ALU* can be further partitioned into the design of submodules like *adders* and *multipliers*. Designers are supported by interactive or batch tools, e.g., a `schematic editor`, a `netlist generator`, or a `simulator`. These tools store design objects (e.g., schematics or netlists) in a database. They perform operations like reading a schematic into a main memory buffer, writing it back, inserting new modules into a schematic or adding a link into a netlist.

From these observations we can derive three kinds of transactions: **Design Transactions (DT)** are used to model certain design tasks, **Tool Transactions (TT)** represent the execution of tools, and **Atomic (Database) Transactions (AT)** perform the elementary operations on the database.

3.2 The Elements of the Model

We illustrate the model by the schema depicted in fig. 1 and the example in fig. 2.

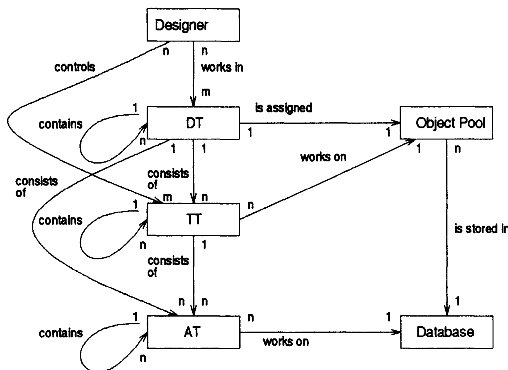

Figure 1: Schema for the Transaction Model

A **Designer** works on design tasks or controls their execution. If he is participating in several tasks, he is assigned several DTs. Vice versa, several designers can be assigned a single DT if they cooperate closely and if the task cannot be split in a reasonable way. Thus we get a n:m-relation between designers and DTs.

A **DT** represents a design task. Often design tasks are subdivided into subtasks, resulting in a hierarchy of DTs. The design objects manipulated by a DT are typically managed as local copies. This leads to a workspace concept that distinguishes between (semi-)public and private workspaces [KLMP84]. To realize such a concept, we introduce **object pools** [US92]. An object pool is assigned to a DT and serves as a (logical) container for all objects accessed by the DT. The operations *checkout* and *checkin* are used to copy objects between object pools. By using these operations DTs can cooperate *explicitly.* DTs are of long duration and typically do not satisfy the *ACID*-properties. Concurrency control is done by a persistent mechanism (e.g., persistent locks) spanning sessions. Recovery must be done in a flexible way since a total rollback of long-lived design activities is often not adequate. In case of a crash, DTs can be reconstructed and continued, because the actual work is done by TTs and ATs, which store their results in a persistent way.

An object pool contains the **design objects** manipulated by a DT. Optionally, objects may be versioned in order to represent the design history or variants. We distinguish between the object pool as the logical workspace and the **database** as the physical container. The fact that data will typically be distributed within a workstation-server environment is not relevant for the discussion in this paper and will therefore be ignored. We assume that objects are manipulated within transaction boundaries and that each elementary operation is performed by an AT which obeys the *ACID*-properties.

Within a DT **tools** like editors or simulators are executed. For simplicity we assume that a **TT** represents the execution of exactly one tool. The object pool of a DT serves as the logical data repository for the TT. The DT has to ensure (in charge of the user or a TT) that needed objects are available in the object pool with appropriate access rights. TTs can be of short or

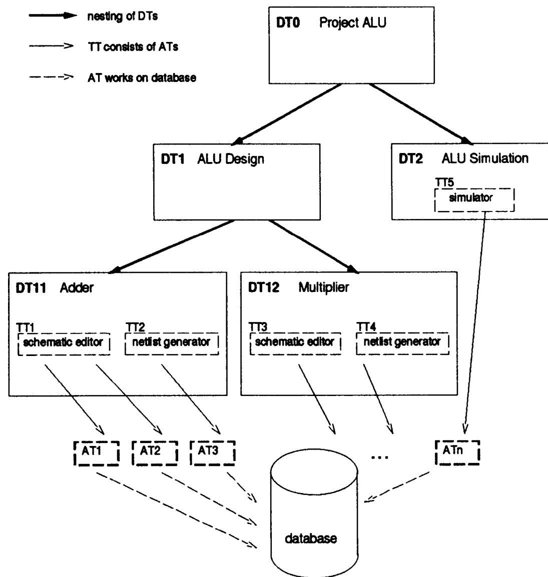

Figure 2: Example

long duration and may ensure the *ACID*-properties. If a TT spans several designer's sessions, it can be suspended and resumed later. TTs are not necessarily units of consistency since a single tool execution need not bring a design object into a consistent state. Several concurrent TTs working on one object pool are synchronized by a locking approach or by other, often tool-specific techniques (e.g., timestamps). Concurrency control information can be managed in a transient way as with conventional transactions, but must be made persistent when a TT is suspended. In contrast to DTs, TTs cooperate ***implicitly*** by accessing a single object pool, i.e., objects released by a certain TT are immediately available to other TTs without explicit cooperation mechanisms. Recovery can be handled by aborting TTs or rolling back partially to a savepoint. Other techniques are also conceivable, e.g., semantic undo of editor operations. If a TT performs its work in main memory buffers, a system crash can lead to a major loss of data. If it works on objects by executing ATs, the recovery mechanism for ATs guarantees that the results of successfully terminated ATs are durable.

ATs realize the *ACID*-properties. They can be used to implement TTs and certain administrative operations (e.g., starting DTs/TTs, ***checkout***, ***checkin***). Hereby, the conventional concurrency control and recovery concepts of a database can be exploited.

These kinds of transactions represent different levels of abstraction: DTs are realized by TTs and/or ATs, TTs are realized by ATs. Each of these levels can apply a different concurrency control or recovery algorithm. For example, DTs can be synchronized by persistent locks which survive failures or system shutdowns, TTs can use transient locks and ATs can employ an optimistic approach. However, there are dependencies between these mechanisms. A TT, e.g., can only acquire a lock if the object and the appropriate persistent lock are already available in the object pool the TT works on.

4 Configurability of the Transaction Model

4.1 Motivation

As was noted in the introduction, a transaction model for design applications has to cover a huge number of complex requirements. This especially holds in the framework area where different kinds of tools and different design methodologies are applied. No single "hard-wired" transaction model is flexible enough to meet all the requirements. Thus, in accordance with the principal idea of frameworks, we propose a transaction model that permits to individually configure a lot of different characteristics.

Example: We can observe different requirements in our *ALU* project:

- Cooperation is very intensive within a subproject, e.g., the development of the *ALU* as part of a *microprocessor*, but less intensive between subprojects. Thus, the concurrency control mechanisms applied within the subproject should be more "liberal".
- Recovery can be handled differently for interactive and batch tools. While a transaction rollback is not acceptable for interactive tools, batch tools like simulators can repeat their work automatically after a rollback.
- The consistency requirements are lower if design objects are passed within a subproject (e.g., a preliminary netlist of the *ALU* is passed to the *simulator*) than if they are released for use by other projects.

The following questions that arise in this context will be discussed in this section:

1. How can transactions be configured?
2. What are the characteristics that should be configurable?
3. How do transactions with different characteristics fit together if they are applied within *one* application or framework?

4.2 Typing of Transactions

We first discuss question 1. Transactions are configured by using a typing mechanism. This mechanism works similar to an object-oriented approach: It is possible to define **transaction types** and to specialize them by an inheritance concept. Transactions are instances of transaction types. Transactions of different transaction types can be combined in *one* transaction system in order to fulfill different requirements of applications. Similar to object-oriented class libraries, transaction type libraries can be built in advance as part of a framework and can be used and/or refined by users of the framework or by tool developers. Transaction types can be defined in a language resembling an object-oriented language or can be specified interactively. They mainly consist of methods defining the **protocols** to be used for the transaction.

Example: A transaction type defining *locking* as the concurrency control protocol can be refined into types applying *two-phase locking* and *non two-phase locking*.

4.3 Combining Transactions of Different Transaction Types

If transactions of different transaction types are used in one system (question 3) it is necessary that the protocols do not conflict. For example, if one transaction locks an object and another one accesses the object in an optimistic way an undesirable behaviour may result. Thus, an arbitrary combination of protocols is impossible.

The concept of object pools offers the possibility to define for each local workspace an individual protocol. Transactions accessing a single object pool have to use the same or at least "compatible" protocols. However, different object pools employ different protocols. Since an object pool is related to exactly one DT, typing of DTs assigns protocols to object pools.

For space reasons, we will restrict the following discussion to DTs and omit typing of TTs. Typing of ATs is not very meaningful since they are normally provided by a database system and are assumed to realize the *ACID*-properties.

As design tasks are often split into subtasks a nested DT hierarchy is built (fig. 3). The type of a certain DT defines the protocols to be used for the object pool of the DT. However, two rules have to be obeyed to make this concept work correctly.

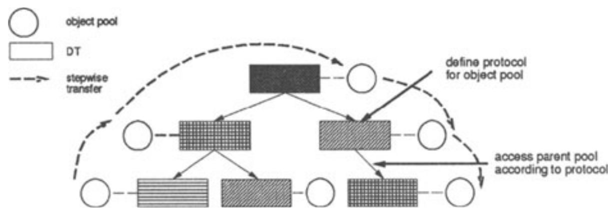

Figure 3: Hierarchy of typed DTs

First, when child DTs of a certain DT access the object pool of the DT by *checkout* or *checkin* they must apply protocols which are **compatible** with the protocols defined for that pool. For example, if a DT requires a two-phase locking algorithm for its object pool, the child DTs must use two-phase locking, too. However, they may apply specialized protocols, e.g. strict two-phase locking or two-phase locking with preclaiming.

Second, the transfer of objects between arbitrary DTs within the hierarchy has to be done **stepwise**, i.e., objects may only be transferred directly to a child DT or to the parent DT. This is important since each DT must check its protocols and may forbid certain operations. If objects were transferred directly as in the conventional nested transaction approach [Mos85] inconsistencies could result because protocols could be circumvented. The conventional approach works well only with strict two-phase locking but does not support arbitrary protocols. In particular, it does not support different protocols within one hierarchy.

Because of the stepwise transfer the subtree rooted at a certain DT builds a **sphere** with well-defined characteristics. DTs outside this sphere are not influenced by the protocols used within the sphere. This is important in order to control the information flow within the hierarchy.

We now describe in more detail some characteristics that can be configured (question 2). The list is not exhaustive, i.e., additional features can be defined. Of course, some characteristics are strongly related which means that they cannot be configured independently.

4.4 Concurrency Control Protocols

Concurrency control is important for two reasons: First, the concurrency control mechanism determines the degree of parallelism and thereby strongly influences the efficiency of the system. Second, it is a prerequisite to perform work in a cooperative way. Concurrency control requires the definition of a protocol. Locking protocols are the best-suited alternative. However, by using special lock modes we can also support a more liberal approach where objects are modified concurrently by using different copies and the changes are merged later on. To specify locking protocols two aspects must be defined:

- **Locking rules**, e.g.,
  - two-phase locking
    In this case, a DT may not acquire a new lock after it has released a lock. Two-phase locking assures the serializability property.
  - strictness
    A DT may not release any lock before it commits. This option avoids cascading rollbacks and is thus important for recovery.
  - preclaiming
    A DT must acquire all its locks at its beginning. This option avoids deadlocks.
- **Lock modes and their compatibility**
  A lock generally has two effects [Unl94]: First, the DT holding the lock is granted the right to perform certain operations (called the *internal effect*). Second, a lock specifies which operations competing DTs are allowed to perform (called the *external effect*). The distinction between these two effects enables the specification of lock modes in a very flexible way. A classical exclusive lock can be specified by an internal effect granting the right to perform all operations and an external effect granting no right at all. However, it is also possible to specify a lock where the holder may modify an object and competitors may still read it or derive new versions from it. In particular, it is possible to specify locks permitting parallel updates of different copies of a single object that are merged later on.

4.5 Cooperation Protocols

Cooperation between transactions can take place in different ways, e.g., by exchanging design objects, delegating work, sending notifications or working on common data. Cooperation must be enabled by the concurrency control protocol. Traditional protocols for *ACID*-transactions prevent cooperation, i.e., they force transactions to work in an isolated way. This is not acceptable for design environments where design is often performed in a cooperative way. The main problem with cooperation is that it allows information to flow between transactions even if it is still preliminary. Thus, it can be difficult to assure consistency and to minimize the effects of recovery actions. To deal with this problem, it should be possible to define which kinds of cooperation are permitted in a certain situation. Cooperation between DTs occurs by explicit operations based on *checkout/checkin*. We distinguish between the following cases:

- **Transferring objects**
  In this case, a DT passes a (possibly preliminary) object to another DT. The locking protocol must not be strict and – if objects are transferred back and forth between DTs – not two-phase.

- **Cooperation based on multiple copies**
  DTs check out objects into their pools and work on these copies concurrently. This is possible if appropriate lock modes are offered. Two aspects have to be configured here: First, it must be defined what happens if a DT has produced a new version of an object that is also of interest for other DTs (e.g., rereading the object, notifying the user). Second, it must be defined what happens when different updates of the same object are checked in (e.g., merging the updates).

4.6 Recovery Protocols

Recovery is the reaction of the transaction system to certain kinds of errors like system crashes, program errors or errors caused by a user. The traditional way to perform recovery, i.e. the rollback of transactions, is not flexible enough for design environments. A major problem for the recovery of DTs is cooperation: By exchanging objects, inconsistent information may be spread throughout the hierarchy which can lead to cascading recovery in case of a failure. For the configuration of recovery we apply three mechanisms:

- **Savepointing**, i.e., specifying when a savepoint is set.
- **Definition of recovery actions**
  It is possible to define how a DT reacts to failures. We distinguish between the following alternatives[2]:
  - The DT rolls back completely or – if possible – to the last savepoint (*partial rollback*).
  - The DT only rolls back the changes to the object that is affected by the failure (*selective rollback*).
  - The user or the application is notified and can perform manual or programmed recovery actions.
- **Prevention of cascading recovery**
  By restricting the flow of information in advance, it is possible to prevent cascading recovery actions. We can apply one of the following mechanisms:
  - The DT uses a strict locking protocol. In this case, no preliminary objects may be released and no other DTs may be affected by a recovery action.
  - Whenever the DT releases an object, it commits its changes to the object, i.e., it waives its right to roll back its changes later (*selective commit*).

4.7 Consistency Specifications

A main goal of transactions is to guarantee **consistency** in case of concurrency and failures. In the conventional model, consistency is defined implicitly, i.e., each transaction is assumed to be consistency-preserving. In a design environment, we should be able to define consistency in a more flexible, application-specific way and should support different degrees of consistency.

Consistency can be specified by defining a **control flow** or by defining **properties** of design objects. The first approach is used in several transaction models (e.g., [NRZ92, WR92]) and

[2] An interesting alternative not discussed here is semantic recovery by compensation [KLS90].

by design flow managers [KM92]. It fits well with our model in that control flow information is assigned to DT types. The second approach works as follows: In order to specify consistency, we use the *feature* concept [Kae91]. A feature is an arbitrary property of a design object that can have a (possibly ordered) number of values. A set of features defines the consistency state of a design object. We can use features in order to specify consistency requirements when objects are transferred. First, it is possible to specify the minimum consistency of an object that is checked out by a DT. In this way, the DT can guard itself against objects with an insufficient degree of consistency. Analogously, it is possible to specify the minimum consistency of an object that is checked in by a DT. In this way, the DT assures that is does not release objects with an insufficient degree of consistency.

Since *checkout* and *checkin* are also used for cooperation, it is possible to specify whether cooperation is permitted for objects with a certain degree of consistency.

4.8 Application Example

To illustrate our approach, we show some type definitions suitable for the DTs in our example.

Example:

- Concurrency control characteristics
  The objects in the pool of the *ALU Development* DT are versioned in a linear order. Concurrency control is done by non-two-phase locking. Valid lock modes are *share* for reading a version and *derive* for deriving a new version. In the lock compatibility matrix, *share* is compatible with *share* and *derive*, but *derive* is not compatible with itself.
- Cooperation characteristics
  The *ALU Development* DT may transfer (possibly preliminary) versions of netlists to the *ALU Simulation* DT (only for reading).
- Recovery characteristics
  The *ALU Development* DT creates an automatic savepoint for an object after a TT has modified it. The *ALU Simulation* DT rereads a netlist if it has been invalidated by a recovery action and restarts the simulation.
- Consistency characteristics
  The *ALU Development* DT may *checkin* only objects for which the simulation with certain test patterns was successful.

5 Conclusion

The transaction model sketched in this paper distinguishes itself by a high degree of configurability. This is achieved by defining transaction types using certain transaction primitives and combining them into a heterogeneous transaction system. In this way, the model is superior to other approaches and is open to model most of their features. Since the specification of transaction types is a complex task requiring knowledge in this area, a framework should provide a set of general purpose transaction types for tool developers or framework users. If necessary, a framework administrator can build new or refine existing transaction types.

An implementation of the model requires a specification mechanism for transaction types. There are two main possibilities: First, a language can be defined that should resemble an object-oriented language or be an extension of it. Second, transaction types can be defined interactively.

In our prototype based on the *JESSI Common Framework* [Ste92] we have chosen the interactive approach. However, we plan to design a suitable language because this seems more flexible.

There are some extensions that can improve our approach. First, it should be possible to use information from the data schema in order to specify protocols for certain types of objects. This leads to an integration of the language for specifying transaction types with the data definition language. Second, there should be a rule mechanism permitting the definition of automatic reactions in case of synchronization conflicts, errors or consistency violations and supporting certain forms of cooperation. We are currently investigating these topics.

References

[BS94] A. Bredenfeld and H. Streibel. Report and Demonstration of first Prototype of Complex Transaction Model. Technical Report JCF/GMD/006-01/30-Jun-94, Jessi-Common-Frame (ESPRIT Project 7364), June 1994.

[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.

[Ioc89] C. Iochpe. Database Recovery in the Design Environment: Requirement Analysis and Performance Evaluation. Dissertation, Universität Karlsruhe, 1989.

[Kae91] W. Kaefer. A Framework for Version-based Cooperation Control. In *Proc. 2nd Intl. Symp. on Database Systems for Advanced Applications (DASFAA)*, April 1991.

[KKB88] H.F. Korth, W. Kim, and F. Bancilhon. On Long-Duration CAD Transactions. *Information Sciences*, 46:73–107, 1988.

[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.

[KM92] T. Kathöfer and J. Miller. The JESSI-COMMON-FRAME Project – Sub-project Development –. In *Proc. 3rd IFIP Workshop on Electronic Design Automation Frameworks*, pages 253–269, March 1992.

[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.

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

[MUZ94] A. Meckenstock, R. Unland, and D. Zimmer. Flexible Support of Cooperative Design Environments by a Transaction Toolkit (in German). In *Proc. STAK '94*, pages 9–26, March 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.

[RMH+94] N. Ritter, B. Mitschang, T. Härder, M. Gesmann, and H. Schöning. Capturing Design Dynamics – The Concord Approach. In *Proc. IEEE Data Engineering*, February 1994.

[Ste92] B. Steinmüller. The JESSI-COMMON-FRAME Project - A Project Overview. In *Proc. 3rd IFIP Workshop on Electronic Design Automation Frameworks*, pages 227–238, March 1992.

[Unl94] R. Unland. Control of Collaboration within Intelligent and Cooperative Information Systems. In *Proc. CoopIS-94*, May 1994.

[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.