Cooperative Transactions in Interoperable Environments

Weihai Yu

Computer Science Department
University of Tromsø
N-9037, Tromsø, Norway

weihai@cs.uit.no

Abstract

So far, cooperative transactions and interoperable (multidatabase) transactions are rather independent research topics. On the one hand, various cooperative transaction models have been proposed in the literature. They attempt to weaken the traditional ACID properties to maximize sharing among on-going transactions. On the other hand, researches toward interoperable transaction management have been related to the local autonomy issues. In supporting cooperative transactions in interoperable environments, there is yet another issue concerning heterogeneity among transaction models themselves. However, this last issue has not yet attracted enough attention. In this paper, we present our approach in which transactional behavior is encapsulated into the abstraction called *transaction ally* (TA). The approach is based on the observation that abstraction (encapsulation) and type polymorphism, the two important properties of the object-oriented diagram, are essential to achieving interoperability with respect to data model heterogeneity. In our framework, TAs are typed and TA types have properties similar to type polymorphism. This makes a step closer toward transaction model interoperability.

1. The Issues

Some transaction models have been proposed in the literature to address the problems concerning cooperative (and potentially long) activities. Meanwhile, some other transaction models have been proposed for interoperable multidatabases that face the local autonomy problems. Here, there is a distinction between cooperation and interoperability. *Cooperation* means that concurrent transactions involved in a single application may share data to a great extent. *Interoperability* means that an application may involve different existing systems with originally different design. [6] includes some transaction models for both cooperation and interoperability. Figure 1 illustrates the research domain concerning cooperative and interoperable transactions.

We believe there is a potential demand for cooperation among different existing systems. Let's first look at a simple scenario that will be used throughout this paper.

Scenario. *Suppose we need an interoperable environment that allows cooperation. The environment is built up atop three existing systems which support the traditional flat transaction model, the nested transaction model and a cooperative transaction model respectively.*

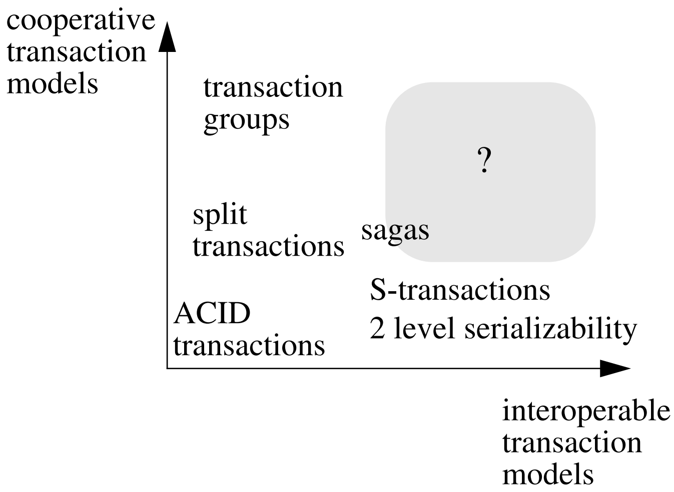

Figure 1 Cooperative transactions and interoperability

- *In the flat transaction model similar to [8], transactions have the ACID properties that are enforced using strict two phase locking and two phase commitment.*
- *In the nested transaction model [9], **top-level transactions** behave in a similar way as transactions in the flat transaction model, whereas locking and commitment with **nested-level transactions** are relative to their parent transactions.*
- *In the cooperative transaction model (modified from [1]), a **cooperative transaction** may join a **transaction group**, multiple transaction groups may merge into one, and a transaction group may split into multiple ones, all dynamically during execution.*

Now, several research groups which have been using one of the three systems respectively, want to cooperate somehow, whereas none of them want to give up the system they have been using.

The question is: what kind of transaction support can we obtain for cooperation in the global environment?

There has been so far no transaction support for this kind of cooperation. What can be done is simply that every group works on its own system and cooperation only occurs later without transaction support.

There are basically two issues concerning supporting cooperative transactions in an interoperable environment. The first issue is how a system can make its transactional behavior known to the global environment and therefore can take part in global activities. For instance, can the system supporting groups of cooperative transactions let the global environment know of this fact? The second issue is, when transactional behavior of local systems is known to the global environment, what global transactional behavior can be achieved, and how? For instance, can groups of cooperative transactions be supported in the global environment? What are the restrictions to those systems that do not provide this facility? Current research on multidatabase transaction management [3] is concerned with the second issue in the case that all local systems have certain traditional transactional behavior (thus not supporting cooperation).

This paper presents our approach toward cooperative transaction support in interoperable environments where different transaction models are involved. Our approach, like the one for data model interoperability such as in OMG/CORBA and ANSAware, is based on abstraction (encapsulation) and type polymorphism. In OMG/CORBA and ANSAware, an object-oriented data model is introduced as a canonical data model. Semantics of local objects can be encapsulated into well defined interfaces of the canonical data model and exported to the global environment. With type polymorphism, even thought the interface provided by the server object is not exactly the same as expected by the client, operations can still be called and executed, as long as the interfaces have the required relationship (subtyping, type conformance, etc.). Thus type polymorphism makes the exported interfaces more applicable in the global environment. In our approach, transactional behavior is encapsulated into the so-called *transaction allies* (TAs). TAs are typed. TA types can be exported through their interfaces; they are organized into hierarchies with properties similar to polymorphism.

This paper is organized as follows. Section 2 describes how transactional behavior of local systems can be encapsulated into TA types. Sections 3 describes how TA types can be organized hierarchically and presents the relevance of such organization to interoperable cooperative transacion support. Section 4 proposes the steps necessary to achieve interoperable cooperative transacions. Section 5 discusses related work. Section 6 concludes the paper and proposes future work.

2. Encapsulating Transactional Behavior

A *transaction* is a computational unit with certain *transactional behavior*. More precisely, a transaction *manipulates* the effects on objects according to some specific rules such as strict two phase locking, two phase commitment, etc..

The first step in our approach toward cooperative transactions in interoperable environments, is to encapsulate the various transactional behavior into *transactional types*. TA types presented in this paper are such transactional types. A *transaction model* contains one or more transactional types. That is, in a transaction model, different transactions may have different transactional behavior.

Transactional types are concerned with the manipulation of effects on a set of objects and thus are largely dependent on the behavior of the objects being manipulated. In our approach, transactional behavior is decoupled into two parts:

- Transaction-dependent behavior, including the termination decision (commit or abort) and execution dependencies between transactions.
- Object-dependent behavior, including object-level concurrency control and recovery.

This decoupling of behavior leads to the concept of transaction allies. When transactions of a certain type TAction (T can be a key word such as “Atomic”) are used in an application, the application contains TActions and TMable data objects[1]. A transaction and the manipulated objects constitute a *transaction ally* (*TA*). A TA is an instance of a *transaction ally type* T (or *TA type* T for short), which is a composition of the two types TAction and TMable[2]. A TAction may *manipulate* a number of TMable objects, meaning that operations on these objects comply with certain rules. In enforcing the rules, TActions are responsible for the transaction-dependent behavior; TMable objects are responsible for the object-dependent behavior.

Suppose Atomic TAs are used for the flat transaction model. In Figure 2, the interfaces marked with an “s” (server) are exported to the global environment. Typically, an application calls `be-`

1. TMable stands for <u>T</u>Action-<u>m</u>anipul<u>able</u>.
2. In a polymorphic environment where an object may have multiple types, TMable can be seen as one of the types.

gin() on an AtomicAction to start an Atomic TA. The application then calls lock() on AtomicMable data objects. These objects in turn call register() on the AtomicAction in order to "enter" the Atomic TA. When a data object is in the Atomic TA, the execution of all the operations, such as op1(), op2(), etc., complies with the rules of the Atomic TA. Finally, the application calls end() or abort() on the AtomicAction to terminate the Atomic TA. AtomicMable objects provide operations prepare(), commit() and undo() that are invokable by the manipulating AtomicAction. end() (abort()) of the AtomicAction ensures that *all* the objects in the Atomic TA execute commit() (undo()).

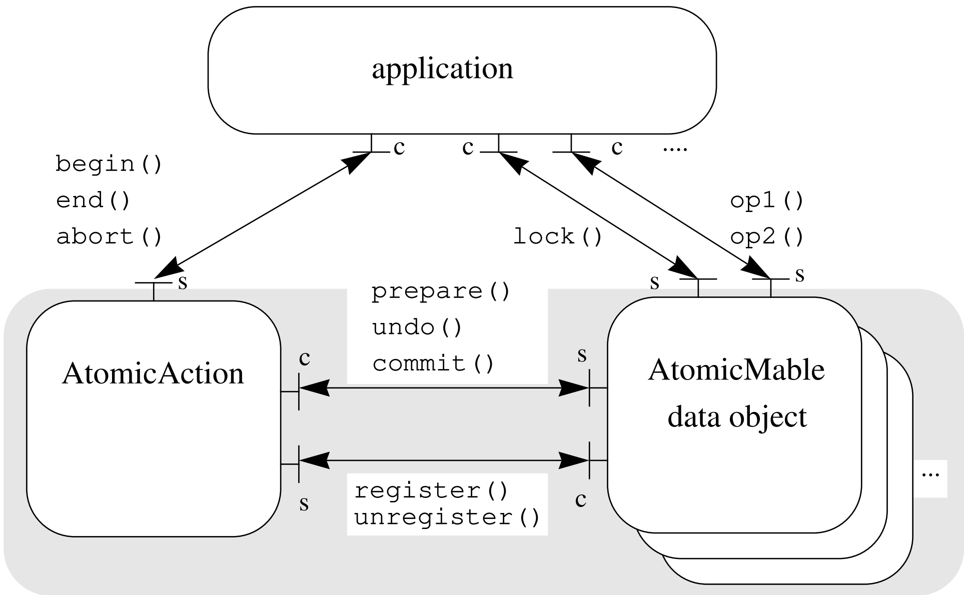

Figure 2. Interactions of an Atomic TA

The behavior of an Atomic TA is specified using the classes AtomicAction and AtomicMable. We choose to use a specification mechanism similar to Larch/C++ and VDM but more programming language oriented. Particularly, there is no precondition of an operation. The operation should be able to report undesirable effects via its returned values. However there is a when guardian which blocks the execution when it's not satisfied. Also there is no additional tier as in the Larch family. The effect of an operation may include that of other operations as nested invocations. We also use some primitives that are not available in programming languages such as C++ but can be supported by a class library. As an example, the semantics of the commit() operation of AtomicMable is defined below:

```
AtomicMable::commit(Tid tid) {
     effect     if (modified(tid))
                     then self'.stable = self.stable.tid.prepare
                and
                (∀lck in self.locks:
                     if (lck.owner == tid) then releaseLock(lck))
}
```

Every AtomicAction has a unique tid (transaction identifier). commit() is invoked by the AtomicAction with tid tid after all prepare()'s return true. An AtomicMable object has a volatile state self.volatile and a stable state self.stable. It may have some additional stable states, for example, self.stable.*tid* which is generated when the object is registered to the AtomicAction, and self.stable.*tid*.prepare which is generated after `prepare()` is executed on behalf of the AtomicAction. `self` and `self'` are respectively the states before and after the execution of the operation. See also the specification of the `lock()` operation in Section 3.1 for the usage of the different states.

3. TATypes in Hierarchy

With type polymorphism, even though the behavior exported by a local system is not the same as that required by some client in the global environment, exported operations can still be called and executed. With respect to transactional behavior, this means that although the transactional behavior of local systems is not the same as expected in the global environment, there is still some chance for the local systems to be involved in the global activities.

In order to achieve some kind of polymorphism, TA type are organized hierarchically.

3.1. Generalization vs. specialization

Within a type hierarchy, new derived types are extensions to existing base types. Typically in object-oriented software development, derived types (subtypes) are specializations of base types (supertypes). For example, rectangle can be a specialization of polygon, i.e., a rectangle is a polygon with exactly four edges. An instance of a subtype can be used wherever that of the supertype is expected.

The extension of transactional behavior is just the other way around. Extended transactional behavior is actually a generalization to the one on which it is based. For example, the nested transaction model can be seen as a generalizaion of the flat transaction model: a flat transaction can be seen as a top-level transaction with zero child transaction. Wherever a top-level transaction is expected, a flat transaction can be used, with restricted functionality (i.e., without being able to have any child transaction).

Basically, a generalized TA type increases concurrency. More control is needed, though, such that the increased concurrency does not lead to uncontrolled inconsistency. The additional control is achieved with more state information and operations in the classes of the new TA type.

Figure 3 shows the class hierarchy of the TA types in question and new operations introduced in TA types of the cooperative transaction model. *UDT*s (user defined types) are derived from concrete -Mable types. In the TA type hierarchy, there is an abstract TA type X which provides an abstract base for concrete TA types. It is neither a specification nor a generalization of any concrete TA type. Only the common behavior of most TA types is defined in X. More specific behavior is defined in concrete TA types derived from X.

As an example of an operation for all -Mable objects, `lock()` of XMable is defined below:

```
bool XMable::lock(XLock xl) {
     effect     mType = xl.owner.mableType(self, xl) and
                result = mType::setLock(xl) and
                if (result)
                then    self'locks = self.locks.merge(xl) and
                        xl.owner.register(self, mType)
                        if (xl.mode >= Write)
                        then self'.modified
                                  = self.modeified.add(xl.owner) and
                             self'.stable.tid = self.volatile
}
```

`lock()` aquires a lock and does the necessary initialization. In the specification, `mType` is the actual -Mable type of the data object. `mType` is determined by the type of the manipulating transaction, the -Mable classes of the data object, and the lock type used for locking. As we

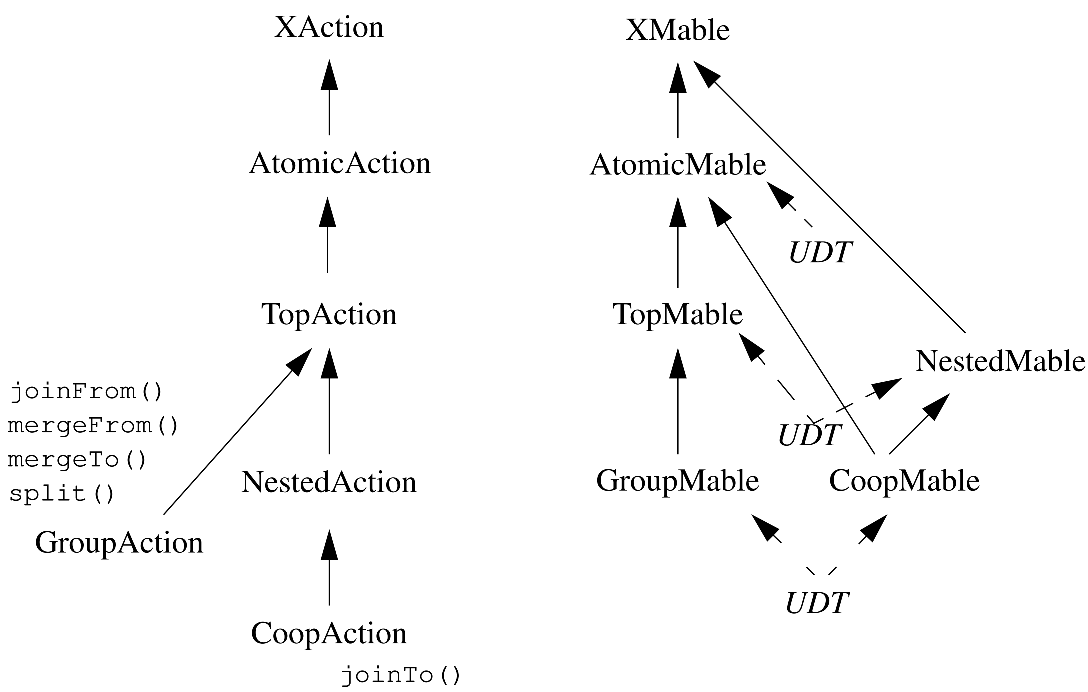

Figure 3. Class hierarchy of TA types

will see in Section 3.2.1, the actual -Mable type of a data object can be different from the type of the TA the objet is in. To aquire a lock, `setLock()` of the actual -Mable type is executed. Note that for each -Mable type there can be a lock type associated, for instance, the CoopLock type associated with CoopMable.

As another example, `delegate()` of XMable is used by many exetnded -Mable types:

```
XMable::delegate(Tid from, Tid To) {
     effect     ∀lk in self.locks:
                (if (lk.owner == from)
                then    self'locks = self.locks.merge(lk.delegate(to)) and
                        if (lk.mode >= Write)
                        then self'.modified
                              = (self.modified.remove(from)).add(to) and
                             self'.stable.to = self.stable.from)
                and
                to.register(self.tid) and
                from.unregister(self.tid)

}
```

A GroupAction behaves like a TopAction. Besides, a GroupAction may allow other CoopActions to become its new member; it may split into multiple independent GroupActions; and multiple GroupActions may merge into one GroupAction. A CoopAction behaves either as an independent AtomicAction, or, when it joins a GroupAction, as a NestedAction. Figures 4 and 5 show the specification of the TA types of the cooperative transaction model.

3.2. Coexistence of multiple transaction models

With the TA abstraction, objects are responsible for their own transactional behavior. Transactions of originally different models may coexist in the environment. There is neither a need for a centralized transaction manager in the global environment, nor a need for all the local trans-

```
class GroupAction: TopAction {
public:
      void mergeTo(Tid);
      Tid split(Tid_set, GroupMable_set);
      void joinFrom(Tid);
};

class CoopAction: NestedAction {
public:
      Tid begin();
      Tid begin(Tid);
      void joinTo(Tid);
};

class GroupMable: TopMable {
};

class CoopMable: TopMable, NestedMable {
public:
      virtual void commit();
protected:
      virtual bool setLock(CoopLock);
};

class CoopLock: NestedLock {
restricted:
      virtual ConfMode conflict(XLock);
      void delegate(Tid);
};
```

Figure 4. Interfaces of Coop and Group TA types

```
GroupAction::mergeTo(Tid ga) {
      effect     ∀ca in self.children: ca.joinTo(ga)
                 and
                 ∀obj in self.objs: obj.delegate(self.tid, ga)
}

Tid GroupAction::split(Tid_set cas, GroupMable_set objs) {
      effect     result = (new GroupAction).begin()
                 and
                 ∀obj in (objs * self.objs): // * set union
                                   obj.delegate(self.tid, result)
                 and
                 ∀ca in (cas * self.children): ca.joinTo(result)
}

GroupAction::joinFrom(Tid ca) {
      effect     addChild(ca) // defined in NestedAction
}
```

(to be continued)

Figure 5. Specification of operations of the Group and Coop TA types

```
Tid CoopAction::begin() {
     effect     result = TopAction::begin() and
                self'.parent = nil
}

Tid CoopAction::begin(Tid ga) {
     effect     result = NestedAction::begin(tid)
};

CoopAction::joinTo(Tid ga) {
     effect     ga.joinFrom(self.tid) and self'.parent = ga
}

bool CoopMable::setLock(CoopLock lck) {
     when       ∀lk in self.locks:
                        (lck.CoopLock::conflict(lk) != Conflict)
     effect     result = true
}

CoopMable::commit(Tid tid) {
     effect     if (tid.getParent() != nil)
                then NestedMable::commit()
                else TopMable::commit()
}

ConfMode CoopLock::conflict(XLock lck) {
     effect     if (self.owner.getParent() == nil)
                then    result = TopLock::conflict(lck)
                else    result = NestedLock::conflict(lck)
}

CoopLock::delegate(Tid tid) {
     effect     self'.owner = tid
}
```

Figure 5. Specification of operations of the Group and Coop TA types (continued)

action managers complying to the same standard. Transactions can manipulate objects in different local systems, as long as the objects can show the corresponding behavior.

Referring to the hierarchy in Figure 3, a NestedAction in a global environment can manipulate NestedMable objects in the local systems, regardless of whether they were originally designed for the nested transaction model or for the cooperative transaction model, since a UDT derived from CoopMable subsumes NestedMable behavior.

Also, a CoopMable object can be manipulated by a CoopAction and a NestedAction simultaneously, because it may behave either as NestedMable or as CoopMable. It is now the responsibility of the object to determine how it behaves in different TAs.

This later case, however, does not imply that a data object can be concurrently involved in TAs of arbitrary types, since some TA types may enforce conflicting rules. Basically, all objects in an application must have the same local atomicity property [13]. Furthermore, when an objects is in TAs of different types, there should be no conflicting operation scheduling orders within the object. More on correctness issues can be found in [15].

Therefore different transaction models may coexist in the global and local environments.

- A transaction may manipulate objects that were originally designed for different purposes.
- An object may be manipulated by transactions of different models.

3.2.1. Over-manipulating transactions

The hardest question is, however, whether the cooperative transaction model can be applied to local systems that do not support the model. That is, can a GroupAction or a CoopAction in the global environment manipulate local objects that were designed for the flat transaction model or the nested transaction model only. Surprisingly, the answer with our scenario is yes. Such GroupAction or CoopAction are called over-manipulating transactions.

Definition. Let BMable be a base type of DMable. A transaction DAction *over-manipulates* BMable objects if it can manipulate both DMable objects and BMable objects at the same time.

A discussion on which BMable objects a DAction can over-manipulate can be found in [15]. In the example, a GroupAction can over-manipulate TopMable objects; a CoopAction can over-manipulated NestedMable objects and AtomicMable objects.

Transaction over-manipulation is somewhat similar to polymorphism with subtyping. An instance of a subtype can be used wherever that of a supertype is expected. Since BMable can be considered a specialization of DMable, BMable objects can be applied in places where DMable objects are expected. That is, BMable objects can appear in D TAs.

Of course, a DAction that over-manipulates some BMable objects has restricted behavior, since these objects do not provide the full functionality as the usual DMable objects. In the example, when a GroupAction over-manipulates some TopMable objects, these objects cannot be split from GroupAction, neither can the GroupAction be merged into another GroupAction, because all these require the objects to be able to delegate from one TA to another (delegation is not allowed in TopMable, though it is supported in XMable). A CoopAction can over-manipulate AtomicMable (-only) objects when it is not in a transaction group (i.e., its parent is `nil`), and after it does so, it may not join a transaction group, because objects that are AtomicMable-only are not able to use the parent relation for locking and lock propagation. A CoopAction manipulating NestedMable objects, however, may dynamically join a transaction group, because CoopMable does not add any functionality to NestedMable.

4. Cooperative Transactions in Interoperable Environments

In this section, we propose the steps to achieve cooperative transactions in interoperable environments using the TA abstraction. The steps are similar to those taken by OMG/CORBA and ANSAware for information interoperability, given a canonical data model.

First of all, the transactional behavior of local systems should be abstracted into hierarchies of TA types. The hierarchies are then exported to the global environment. It is important that the hierarchies, rather than the interfaces of individual TA types, are exported, such that the global environment may make use of the polymorphism-like properties among different TA types in the hierarchies.

In the interoperable architecture such as OMG/CORBA and ANSAware, an interface repository maintains the necessary information about exported interfaces. The information, together with some polymorphism rules (such as type conformance rules), determine whether the invocations can be directed to certain interfaces. Similarly, a repository is needed for exported TA types. The information maintained in the repository should include that about over-manipulability of transactions.

In CORBA and ANSAWare, object adapters provide the primary way that local objects access services of the global environment, such as generation and interpretation of object references, method invocation, and so forth. Similarly, invocation of operations of TA types can be supported via object adapters. How much an object adapter can achieve in this case is largely dependent on the openness (for example, whether the commit message formates and protocols are public) and autonomy (which, as mentioned in Section 1, is a hot research topic for multi-database transaction processing) of local transaction systems. The scenario in this paper is simplified in the sense that the nested transaction model is a "pure" extension of the flat transaction model and the cooperative transaction model is a "pure" extension of the nested transaction model.

With all the above done, a transaction in the global environment can manipulate data objects in local systems; transactions of different models may coexist in the environment; cooperative transactions can be applied even though they are not fully supported by some local systems.

5. Related Work

First of all, we claim that our approach attempts to address the issues how different transaction management systems can understand each other and how they can be integrated in a consistent manner. It does not address the issues concerning local system autonomy. The approach is general enough, though, to apply to transaction models for these specific issues [3].

Standard interfaces of transaction managers and individual resource managers have been recognized as a key to open transaction processing (e.g., [14]). Different transaction managers and resource managers providing the same interfaces can be integrated in an interoperable environment. [11] goes a bit further by including object orientation and nested transactions. However, there is no standard for cooperative transactions. Our approach takes into account the behavioral aspects of different transaction models.

[12] is the only work we are aware of aiming at flexible transaction management in interoperable environments. [12], however, focuses on the flexibility of service policies, such as pessimistic or optimistic alternatives, instead of transaction models in general. Furthermore, neither of the issues whether different transaction types may coexist in an interoperable environment, nor how transactional features of existing systems can be exported to the environment, is taken into account.

ACTA [4] is a formal framework that attempts to characterize the whole range of structural and behavioural properties of different transaction types. Our work is a *practical* (thus less formal) attempt to supply a general framework for applying multiple transaction types. A formal framework like ACTA can be used as a guideline of organizing class hierarchies for TA type derivation.

There are researches on generic support for implementations of extended transaction models [7][2][1]. Georgakopoulos et al. [7], for example, proposed a programmable transaction environment. In this environment, the behavior of transactions is determined by transaction state dependencies (action-dependent behavior) and correctness dependencies (object-dependent behavior). The environment supports the enforcement of different classes of such dependencies. A transaction system can be instantiated using a combination of the supported dependencies. The TA abstraction, however, further clarifies the role of objects in transaction processing. Objects are responsible not only for concurrency control, but also for recovery. Furthermore, our framework is open-ended and allows software reuse.

In Arjuna [10], an atomic transaction is an object containing a list of "record" objects that enforce the rules of atomic transactions on accessed objects. Arjuna is similar to TAs in the ways transactions and objects interact. However, Arjuna does not specify in general how objects are involved in different transaction types. Though record objects can serve as "links" between atomic transactions and objects, objects are not completely unaware of the existence of atomic transactions. Objects are involved in transaction processing in a somewhat *ad hoc* way.

6. Conclusion and Future Work

So far, cooperative transactions and interoperable transactions are rather independent research topics. In this paper, we recognize the issues concerning cooperative transactions in interoperable environments, and present a novel approach. Our approach is based on the observation that abstraction (encapsulation) and type polymorphism of the object-oriented diagram are essential for interoperability with respect to data model heterogeneity. In our approach, transactional behavior can be encapsulated in the transaction ally (TA) abstraction. In a TA, the functionality of a conventional transactional manager is decoupled into transactions and objects. This makes data objects responsible of their own transactional behavior. TA types are organized in a hierarchical way. With the TA abstraction, transactional behavior of local systems can be exported to the global environment; different transaction models of the local systems may coexist; and finally, over-manipulation allows a transaction to manipulate objects in systems without the full transactional behavior expected.

Currently, we are considering prototyping the TA abstract in a non-interoperable environment. We have also considered the issues of applying the approach in an interoperable environment, Fril [5], which is under development in the department. The basic idea is using object adapters to map transactional behavior between global and local environments.

References

[1] R. Barga and C. Pu. A Practical an Modular Implementation of Extended Transaction Models. *Proceedings of the 21th VLDB Conference*. 1995.

[2] A. Biliris, S. Dar, N. Gehani, H. V. Jagadish and K. Ramamritham. Asset: A System for Supporting Extended Transactions. *Proceedings of ACM SIGMOD '94*, pages 44--53. 1994.

[3] Y. Bretbart, H. Garcia-Molina and A. Silberschatz. Overview of Multidatabase Transaction Management. *VLDB journal.* 1(2):181--239, 1992.

[4] P. K. Chrysanthis and K. Ramramritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. *Proceedings of ACM SIGMOD '90*, pages 194-203. 1990.

[5] F. Eliassen and R. Karlsen. Providing Interoperability using Functional Programming Concepts. *Proceedings of EurOpen '91*, pages 249--266. 1991.

[6] A. K. Elmagarmid (editor). *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, 1992.

[7] D. Georgakopoulos, M. Hornick, P. Krychniak and F. Manola. Specification and Management of Extended Transactions in a Programmable Transaction Environment. *Proceedings of the 10th International Conference on Data Engineering*. Feb., 1994.

[8] J. N. Gray and A. Reuter. *ransaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[9] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* The MIT Press, Cambridge, Mass, 1985.

[10] S. K. Shrivastava, G. N Dixon and G. D Parrington. An Overview of Arjuna: A Programming System for Reliable Distributed Computing. *IEEE Software.* 8(1):66-73, Jan., 1991.

[11] P. Taylor, V. Cahill and M. Mock. Combining Object-Oriented Systems and Open Transaction Processing. *The Computer Journal.* 37(6):487--498, 1994.

[12] M. B. F. Toledo and G. S. Blair. Transaction Support for an ANSA-based Platform. *Proceedings of the 10th International Conference on Data Engineering*, pages 209--216. 1994.

[13] W. E. Weihl. Local Atomicity Properties: Modular Concurrency Control Abstract Data Types. *ACM Trans. Prog. Lang. Systems.* 11(2):249-282, April, 1989.

[14] X/Open. *Distributed Transaction Processing Reference Model: The XA Specification.* X/Open Company Limited, 1991.

[15] W. Yu and F. Eliassen. Flexible Transaction Management in an Interoperable Database Environment. T. Cheung, J. Fong and B. Siu (editor), *Database Reengineering and Interoperability.* Plenum Publishing Corporation, 1995.