Concurrency Control in *Transactional Drago*[⋆]

M. Patiño-Martínez[1], R. Jiménez-Peris[1], J. Kienzle[2], and S. Arévalo[3]

[1] Technical University of Madrid (UPM), Facultad de Informática,
E-28660 Boadilla del Monte, Madrid, Spain, {rjimenez, mpatino}@fi.upm.es
[2] Swiss Federal Institute of Technology in Lausanne, Department of Computer Science, Lausanne, Switzerland, Joerg.Kienzle@epfl.ch
[3] Universidad Rey Juan Carlos, Escuela de Ciencias Experimentales, Móstoles, Madrid, Spain, s.arevalo@escet.urjc.es

Abstract. The granularity of concurrency control has a big impact on the performance of transactional systems. Concurrency control granularity and data granularity (data size) are usually the same. The effect of this coupling is that if a coarse granularity is used, the overhead of data access (number of disk accesses) is reduced, but also the degree of concurrency. On the other hand, if a fine granularity is chosen to achieve a higher degree of concurrency (there are less conflicts), the cost of data access is increased (each data item is accessed independently, which increases the number of disk accesses). There have been some proposals where data can be dynamically clustered/unclustered to increase either concurrency or data access depending on the application usage of data. However, concurrency control and data granularity remain tightly coupled. In *Transactional Drago*, a programming language for building distributed transactional applications, concurrency control has been uncoupled from data granularity, thus allowing to increase the degree of concurrency without degrading data access. This paper describes this approach and its implementation in Ada 95.

Keywords: transactions, locking, distributed systems, databases.

1 Introduction

Transactions [GR93] were proposed in the context of database systems to preserve the consistency of data in the presence of failures and concurrent accesses. Transactions are also useful as a way of organizing programs for distributed systems. Their properties simplify the development of correct programs, hiding the complexity of potential interactions among concurrent activities and the failures that can occur in a distributed system. Transactions provide the so-called ACID properties, that is, atomicity, consistency, isolation and durability. A transaction is executed completely or its effect is as it never had been executed (atomicity). If a transaction ends successfully (it commits), its effects will remain even in the advent of failures (durability). If a transaction does not commit, it aborts. In case

⋆ This research has been partially funded by the Spanish National Research Council, CICYT, under grant TIC98-1032-C03-01.

of an abort, atomicity guarantees that all the effects of a transaction are undone, as if it had never been executed. Consistency ensures that the application state is updated in a consistent way. The effect of executing concurrent transactions is as if they were executed sequentially in some order (serializability), that is, a transaction does not see intermediate results from other transactions (isolation).

Concurrency control techniques are used to ensure the isolation property of transactions. Read/write locking is one of the most popular concurrency control techniques. Two locks conflict if they are requested on the same data item, by two different transactions, and at least one is a write lock. A lock in the appropriate mode must be requested before a data item is accessed by a transaction. Locks are released when a transaction finishes. More concurrency can be achieved by defining locks on other operations instead of just read/write locks [BHG87].

The granularity of a data item is its relative size. The concurrency control granularity is the unit to which concurrency control (locking) is applied. In general, data granularity and lock granularity are the same. Data granularity has a big impact on the performance of a transactional system. If a data item is big (for instance, a file), concurrency decreases since the probability of conflicts is higher. That is, if two transactions write the same data item, one of them will not be able to proceed even if they access different parts of the data item. Since transactions only lock those data items they access, more concurrency can be achieved with a finer granularity (for instance, records instead of files). On the other hand, the time taken to load a data item of size N from disk is less than the time needed to load N data items of size 1. Therefore, performance decreases as more disk accesses are needed to access several data items (records) of the original data item (the file). Locks in transactional languages are requested at data item level. That means that the programmer must choose either a coarse granularity to improve data access in detriment of transaction concurrency, or a fine granularity in detriment of data access efficiency.

In this paper we present the approach adopted in *Transactional Drago* [PJA98]. *Transactional Drago* is an Ada extension for programming distributed transactional applications. *Transactional Drago* offers a locking scheme where locking and data granularity have been uncoupled. With this locking scheme the programmer first chooses the data granularity to improve data access. Then, she/he decides the locking granularity for that data item. Locking granularity can vary from the coarsest level (the whole data item) to the finest (each indivisible component of a data item). Although some transactional programming languages also allow changes in the locking granularity (see Section 6), programmers must program the concurrency control, and in some cases they also have to program the recovery mechanism needed to provide the atomicity property of transactions. In *Transactional Drago* none of these error-prone tasks needs to be done. The granularity of concurrency control is declarative, and programmers just decide where locks are applied.

The paper also describes how *Transactional Drago* is translated into *Optima* [KJRP01], the evolution of the Ada transactional framework *TransLib* [JPAB00].

Additionally, the *Optima* facilities for concurrency control and the implementation of lock-based concurrency control in Ada 95 are presented.

This paper is structured as follows. First of all, we present a brief description of *Transactional Drago*. In Section 3 we present how the granularity of locks is defined. The translation of *Transactional Drago* into *Optima* is presented in Section 4. Some details about the implementation of the concurrency control in *Optima* are given in Section 5. Finally, we compare our work with other approaches in Section 6 and present our conclusions in Section 7.

2 *Transactional Drago*

Transactional Drago [PJA98] is an extension to Ada [Ada95] for programming distributed transactional applications. Programmers can start transactions using the `begin-end transaction` statement or *transactional block*. Transactional blocks are similar to block statements in Ada. They have a declarative section, a body and can have exception handlers. The only difference is that the statements inside the block are executed within a transaction.

All data used in transactional blocks are subject to concurrency control (in particular, locks) and are recoverable. That is, if the corresponding transaction aborts, data will be restored to the value they had before executing that transaction. Data items can be volatile or non-volatile (persistent).

A transactional block can be enclosed within another transactional block, leading to a nested transaction structure. A transaction that is nested within another transaction (parent transaction) is called a subtransaction [Mos85].

Transactional Drago implements *group transactions* [PJA01], a new transaction model. One of the novel aspects of the model is that transactions can be multithreaded, i.e., a transaction can have several threads (tasks) that run in parallel. As a result it is possible to take advantage of the multiprocessor and multiprogramming capabilities. Threads working of behalf of the same transaction can cooperate by accessing the same data, i.e., they are not isolated from each other. Since locking is only intended for *inter-transactional* concurrency (logical consistency), latches [MHL$^+$98] are used to provide *intra-transactional* concurrency control (physical consistency) in the presence of concurrent accesses from threads of the same transaction. Latches provide short-term concurrency control (they last for a single operation) in contrast with locks that are long-term concurrency control (they are not released until the transaction finishes).

In *Transactional Drago* both concurrency control mechanisms are implicitly handled by the run-time system, hiding all the complexity from the application programmer. Programs access transactional data (atomic data) just as regular non- transactional data. Programmers do not set and free neither locks nor latches. The underlying system is in charge of ensuring the isolation and atomicity properties and the physical consistency of the data.

3 Lock Granularity

Although programmers do not explicitly request neither locks nor latches, *Transactional Drago* allows them to specify concurrency control for each data item separately, thus increasing concurrency among transactions.

Let us illustrate this mechanism with a simple example. From now on we assume all data are persistent (non-volatile). For instance, if we have the (persistent) array declaration on Fig. 1(a), in transactional languages, locks are applied to the whole array. So, if two transactions update a component of the array, no matter whether they access the same or different components, one of the transactions would be blocked until the other one finishes. By default, this is the semantics of *Transactional Drago*. However, in *Transactional Drago* the programmer can define the granularity of locks to be array components (thereby uncoupling data granularity and locking granularity). In this case two transactions updating different components can be executed concurrently. The resulting effect is just as if each component of the array were separate data items. *Transactional Drago* goes even further by allowing concurrency control to be applied at each field of the record inside the array. Therefore, two transactions can update different fields of the same component concurrently. This flexibility does not induce a penalty in data access as it does in other approaches: the array does not have to be split in smaller pieces.

```
type mytype is atomic array
     (Array_Index_Type) of
      record
         a: integer;
         b: float;
      end record;
```

(a) array level locking

```
type mytype is array
     (Array_Index_Type) of
      atomic record
         a: integer;
         b: float;
      end record;
```

(b) record level locking

```
type mytype is array
     (Array_Index_Type) of
      record
         a: atomic integer;
         b: atomic float;
      end record;
```

(c) Field level locking

Fig. 1. Different locking granularities

The definition of the locking granularity is declarative in *Transactional Drago*. The programmer simply specifies locking granularity just using the keyword `atomic`. This keyword is used in the data item type declaration before the type declaration where locks are applied. In the previous example, if the keyword is used at the beginning of the array definition (Fig. 1(a)), locks will apply to the whole array. Because this is the normal behavior in transactional languages, the keyword can be omitted. Locking granularity at record level for variables of the `mytype` type is defined using the keyword atomic just before the definition of each component of the array (Fig. 1(b)). Field level locking is specified using the keyword atomic before the type of each field (Fig. 1(c)).

The *Transactional Drago* compiler checks for each data item (variable) whether the locking granularity has been appropriately defined by applying the following rules:

– The atomic keyword has not been used in the type declaration. By default, locking granularity is set to the whole data item (the coarsest granularity).
– The atomic keyword is used. The keyword should be found once and only once in each path from the leaves to the root of the type tree.

A type tree is built upon the type declaration of a variable. Each node in the tree is a type. The root is the type of the variable and its children are the types this type is made of. For instance, the different trees associated with the `mytype` type are shown in Fig. 2(a). If the keyword is placed before the array or record keywords (Fig. 2(b)), it cannot be placed anywhere else. All the paths from the leafs to the root will contain the keyword exactly once. If the keyword is placed before the integer type, it must also be placed before the float type (Fig. 2(c)).

Locks and latches in *Transactional Drago* are requested implicitly by the system in the appropriate mode and automatically released when the transaction finishes. To change the granularity of locks, only the type definition must be changed, since locking in *Transactional Drago* is specified in a declarative way. There is no need to modify any application code.

Pointers are considered like any other simple type (integer, boolean...), that is, they always have read/write locks. Dynamic data also have concurrency control, which is specified in the same way than for static data.

4 ***Transactional Drago*** Translation

Transactional Drago can be translated into Ada 95 and invocations to either *TransLib* [JPAB00] or its evolution *Optima* [KJRP01], two adaptable OO libraries supporting transactions. Both libraries provide user-defined commutative locking. With commutative locking [GM83] two operations conflict if they do not commute. For instance, let's consider the `set` abstract data type with the following operations: `Insert(x)`, adds x to the set, `Remove(x)`, extracts an element from the set, and the membership test for x, `IsIn(x)`. The corresponding commutativity or conflict table is:

	Insert(x)	Remove(x)	IsIn(x)
Insert(y)	Yes	$x \neq y$	$x \neq y$
Remove(y)	$x \neq y$	Yes	$x \neq y$
IsIn(y)	$x \neq y$	$x \neq y$	Yes

Using commutative locking, pairs of the same operation always commute (e.g., `Insert/Insert`), and pairs of different operations commute if they have different parameters (e.g., `Insert(5)/Remove(7)`). That is, it does not matter the order in which two concurrent transactions perform insert operations, the operations commute and the final value of the set will be the same. On the other hand, if a transaction performs an insert and a concurrent one a remove operation, the operations commute if the parameters are different. Otherwise, the operations do not commute and the final value of the set depends on the order of execution of the two operations. Using read/write locks, the only compatible operations would be pairs of `IsIn` operations. Therefore, commutative locking significantly reduces the number of conflicts by taking advantage of semantic information.

Optima provides the `Lock_Type` abstract class to support user-defined commutative locking. It defines two abstract functions `IsCompatible` and `IsModifier`. A concrete subclass must be derived from this `Lock_Type` class for every data item that is accessed from within a transaction. The class must provide a means to store all the information that is necessary to determine whether two operations invoked on the data item conflict or not. In the `set` example the kind of operation, i.e., `Insert(x)`, `Remove(x)` or `IsIn(x)`, and the parameter `x` must be stored. Based on this information and the commutativity table shown above, the `IsCompatible` function is able to detect conflicts. The `IsModifier` function returns true if the operation modifies the state of the object.

```
package Locks is
   type Lock_Type is abstract tagged private;
   type Lock_Ref is access all Lock_Type'Class;
   function Is_Modifier (Lock : Lock_Type) return Boolean is abstract;
   function Is_Compatible (Lock : Lock_Type;  Other_Lock : Lock_Type)
      return Boolean is abstract;
private
   type Lock_Type is abstract tagged null record;
end Locks;
```

The power and flexibility of commutativity locking can be used to implement the user-defined locking granularity of *Transactional Drago*. The mapping from *Transactional Drago* data items to *Optima* commutative data items is performed as follows. For each data item a locking scheme must be defined. A locking scheme is defined by providing a concrete `Lock_Type` class. The constructor of the concrete `Lock_Type` class always has a boolean parameter which indicates whether it is a read or a write lock. The compatibility function is automatically generated and considers two lock values as compatible according to the following rules: (1) the constructors of the two locks have been called with different parameters; (2) the constructors of the two locks have been called with the same parameters, but both are read locks.

The array level locking type tree for the array example is shown in Fig. 2(a). For this type tree, a `Lock_Type` class with one constructor (`Array_Lock(Boolean)`) is generated. The Boolean parameter indicates whether it is a read or a write lock. Such a lock is equivalent to traditional read/write locks for that type.

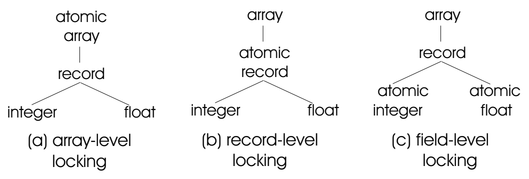

Fig. 2. Type trees for different locking granularities

If the granularity of locks is set at record level (Fig. 2(b)), a constructor with an additional parameter is generated, Record_Lock(Boolean, Array_Index_Type). The new parameter is the array index. In this case, accesses (reads or writes) to different elements of the array will be compatible. Two locks only conflict if they refer to the same array component and one of them is a write lock.

Finally, the type tree shown in Fig. 2(c) illustrates locking at field level. The constructor, Field_Lock(Boolean,Array_Index_Type,Positive), has an additional parameter that represents the declaration order of the field in the record. The following code shows the subclass that is created for the type tree in Fig. 2(c).

```
package Locks.FieldLocks is
   type Field_Lock_Type is new Lock_Type with private;
   function Field_Lock (Modifier : Boolean; Index : Array_Index_Type;
                        Field_Position : Positive) return Field_Lock_Type;
   function Is_Modifier (Lock : Field_Lock_Type) return Boolean;
   function Is_Compatible (Lock : Field_Lock_Type;
                            Other_Lock : Field_Lock_Type) return Boolean;
private
   type Field_Lock_Type is new Lock_Type with record
      Modifier : Boolean;
      Index : Array_Index_Type;
      Field_Position : Positive;
   end record;
end Locks;
```

The Field_Position parameter of the constructor Field_Lock represents the declaration order of the field in the record. The constructor creates a new Field_Lock_Type instance and assigns the parameters to the corresponding record fields. The Is_Modifier function simply returns the value stored in Modifier. The Is_Compatible function is implemented as follows:

```
function Is_Compatible (Lock : Field_Lock_Type;
                        Other_Lock : Field_Lock_Type) return Boolean is
begin
   return (Lock /= Other.Lock) or else
      (not Lock.Modifier and not Other_Lock.Modifier);
end Is_Compatible;
```

In general, a parameter (the array index or field position) is added to the constructor of the corresponding lock for each array/record found in the path from the root of the type tree to each node tagged atomic (not including it).

5 Locking Implementation

This section presents how the advanced concurrency features of Ada 95 have been used to implement long-term and short-term concurrency control in the *Optima* framework.

In *Optima*, the Lockbased_Concurrency_Control protected type implements lock-based concurrency control. One object of this type is associated to each data item during the translation of *Transactional Drago*. This protected type provides four operations, Pre_Operation, Post_Operation, Commit_Transaction and Abort_Transaction. When translating *Transactional Drago* code, every access to a data item is automatically surrounded by calls to Pre_Operation and Post_Operation of its associated concurrency control. Obviously, several Ada tasks might attempt to call these operations simultaneously. To handle this situation correctly, the concurrency control has been implemented in the form of a protected type as shown in Fig. 3. Since Pre_Operation has to be able to suspend the calling task in case of conflicts, the operation is implemented as an entry. The private part of the specification contains three private entries and some attributes, e.g. the list of currently active locks named My_Locks, and a boolean and a natural variable that are used to implement the multiple readers/single writer paradigm.

```
protected type Lockbased_Concurrency_Control is
   entry Pre_Operation (Lock : Lock_Ref; Trans : Trans_Ref);
   procedure Post_Operation;
   procedure Transaction_Commit (Trans : Trans_Ref);
   procedure Transaction_Abort (Trans : Trans_Ref);
private
   entry Waiting_For_Transaction (Lock : Lock_Ref; Trans : Trans_Ref);
   entry Waiting_For_Writer (Lock : Lock_Ref; Trans : Trans_Ref);
   entry Waiting_For_Readers (Lock : Lock_Ref; Trans : Trans_Ref);
   My_Locks : Lock_List_Type;
   Currently_Writing : Boolean := False;
   Currently_Reading : Natural := 0;
   To_Try : Natural := 0;
end Lockbased_Concurrency_Control;
```

Fig. 3 Lock based concurrency control

Before a data item is accessed from within a transaction, a Lock_Type object instance is created as described in the previous section, and Pre_Operation of the associated concurrency control is called, passing the transaction context and a reference to the Lock_Type object as parameters. The Pre_Operation code is shown in Fig. 4. First, long-term concurrency control must be handled. To guarantee isolation, the concurrency control must determine if the operation to be invoked conflicts with other invocations made by still active transactions. This check is performed in the Is_Compatible function of the Lock_List_Type ($\boxed{1}$). Successively, the new lock is compared to all previously granted locks by calling the Is_Compatible function of the new lock. If a conflict has been detected, the calling task is suspended by requeuing the call on the private entry Waiting_For_Transaction until the transaction having executed the conflicting operation ends ($\boxed{2}$). If, on the other hand, the access does not create any conflict, then the new lock is inserted into the list of granted locks ($\boxed{3}$), and the short-term concurrency control phase is initiated by requeuing on the private entry Waiting_For_Writer ($\boxed{4}$). The two entries Waiting_For_Writer and

Waiting_For_Readers implement the multiple readers/single writer paradigm. Starvation of writers is prevented by keeping readers and writers on a single entry queue. Requests are serviced in FIFO order. Inside Waiting_For_Writers, the nature of the operation, i.e. read or write, is determined by calling the Is_Modifier ($\boxed{6}$) operation of the new lock. Read operations are allowed to proceed, until a write operation is encountered. If there are still readers using the data item, then the writer is requeued to the Waiting_For_Readers entry ($\boxed{7}$). This closes the barrier of the Waiting_For_Writer entry, since the latter requires the Waiting_For_Readers queue to be empty ($\boxed{5}$). After the invocation of the actual operation on the data item, the run-time automatically calls Post_Operation, which decrements the number of readers or unsets the writer flag.

```
entry Pre_Operation (Lock : Lock_Ref; Trans : Trans_Ref) when True is
begin
   if not Is_Compatible (My_Locks, Lock, Trans) then                     1
      requeue Waiting_For_Transaction with abort;                        2
   else
      Insert (My_Locks, Lock, Trans);                                    3
      requeue Waiting_For_Writer with abort;                             4
   end if;
end Pre_Operation;

entry Waiting_For_Writer (Lock : Lock_Ref; Trans : Trans_Ref)
      when not Currently_Writing and Waiting_For_Readers'Count = 0 is    5
begin
   if Is_Modifier (Lock.all) then                                        6
      if Currently_Reading > 0 then
         requeue Waiting_For_Readers with abort;                         7
      else
         Currently_Writing := True;
      end if;
   else
      Currently_Reading := Currently_Reading + 1;
   end if;
end Waiting_For_Writer;

entry Waiting_For_Readers (Lock : Lock_Ref; Trans : Trans_Ref)
        when Currently_Reading = 0 is
begin
   Currently_Writing := True;
end Waiting_For_Readers;

procedure Post_Operation is
begin
   if Currently_Writing then
      Currently_Writing := False;
   else
      Currently_Reading := Currently_Reading - 1;
   end if;
end Post_Operation;
```

Fig. 4 Implementation of lock based concurrency control

Now let us go back to the first phase and see what happens to the calls queued on the `Waiting_For_Transaction` entry. Tasks queued here have requested to execute an operation that conflicts with an operation already executed on behalf of a still active transaction. Each time a transaction ends, this situation might change. When a transaction aborts, the operations executed on behalf of the transaction are undone, and hence the acquired locks can be released. This is illustrated in Fig. 5. `Transaction_Abort` calls the `Delete` operation of the granted lock list ($\boxed{\text{1b}}$), which results in removing all operation information of the corresponding transaction from the list. The same is done upon commit of a top level transaction ($\boxed{\text{1a}}$). If the commit involves a subtransaction, then the locks held so far by the subtransaction must be passed to the parent transaction. This is achieved by calling the `Pass_Up` operation of the `Lock_List_Type` ($\boxed{2}$). In any case, the auxiliary variable `To_Try` is set to the number of tasks waiting in the queue of the entry `Waiting_For_Transaction` ($\boxed{\text{3a}}$ and $\boxed{\text{3b}}$). As a result, all queued tasks are released and requeued to `Pre_Operation`, thus getting another chance to access the data item.

```
     entry Waiting_For_Transaction (Lock : Lock_Ref; Trans : Trans_Ref)
        when To_Try > 0 is
     begin
        To_Try := To_Try - 1;
        requeue Pre_Operation with abort;
     end Waiting_For_Transaction;

     procedure Transaction_Commit (Trans : Trans_Ref) is
     begin
        if Is_Toplevel (Trans.all) then
[1a]       Delete (My_Locks, Trans);
        else
[2]        Pass_Up (My_Locks, Trans, Get_Parent(Transaction.all));
        end if;
[3a]    To_Try := Waiting_For_Transaction'Count;
     end Transaction_Commit;

     procedure Transaction_Abort (Trans : Trans_Ref) is
     begin
[1b]    Delete (My_Locks, Trans);
[3b]    To_Try := Waiting_For_Transaction'Count;
     end Transaction_Abort;
```

Fig. 5 Implementation of lock based concurrency control

6 Related Work

Argus [LS83,Lis88] was the first programming language providing transactional semantics. It provided a set of atomic types and mutexes. Atomic types have predefined concurrency control and recovery. Hence, the granularity of locks for these data types cannot be changed. Programmers can define new data types based on atomic types to increase concurrency, uncoupling data and concurrency control granularity. However, they have to implement concurrency control (using mutexes and atomic types) for those new types, which it is not a trivial task.

Hybrid atomicity [LMWF94] is used in *Avalon* [EMS91] for concurrency control to increase the degree of concurrency. However, the programmer has to program both the concurrency control and the recovery of the new type. This increases dramatically the complexity of programming.

In *Arjuna* [PSWL95] the granularity of locks applies to objects. Hence, to increase efficiency (decreasing the number of disk accesses) the state of an object should be big, but in order to increase concurrency, objects should be decomposed into smaller objects, which implies changing the code of all the applications that use the original object. [WS94] proposes a run-time clustering-declustering support for the *Arjuna* system where data granularity can be changed at run-time. However, a change in data granularity provokes a change in locking granularity.

Transactional C [Tra95] is the programming language provided by the *Encina* transaction processing monitor. Latches (or mutexes in the *Transactional C* terminology) and locks are explicitly set in *Transactional C*, which complicates programmers' task. Latches are only valid within the scope of a transaction. If a thread starts a subtransaction, the physical consistency of the data is not guaranteed.

7 Conclusions

We have presented the mechanisms for controlling the concurrency control granularity in *Transactional Drago*. The main contribution of this paper is that concurrency control granularity has been uncoupled from data granularity. This enables to increase the transaction concurrency without introducing any penalty on data access and/or recovery.

This approach contrasts with other approaches where an increase in concurrency penalizes data access and/or recovery. Furthermore, the approach of *Transactional Drago* is declarative, thus easy to use for programmers. They simply specify the data and concurrency control granularity separately. Changing the locking granularity of a data item is therefore straightforward, and does not require changing the code of the program that uses the data item.

Finally, we have shown how to map the locking granularity of *Transactional Drago* to the locks provided by *Optima*, an Ada transactional library, and presented how the implementation of *Optima* that handles these locks takes advantage of the advanced concurrency features offered by Ada 95.

References

[Ada95] *Ada 95 Reference Manual, ISO/8652-1995.* Intermetrics, 1995.

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

[EMS91] J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann, 1991.

[GM83] H. García-Molina. Using Semantic Knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[JPAB00] R. Jiménez-Peris, M. Patiño-Martínez, S. Arévalo, and F.J. Ballesteros. TransLib: An Ada 95 Object Oriented Framework for Building Dependable Applications. *Int. Journal of Computer Systems: Science & Engineering*, 15(1):113–125, January 2000.

[KJRP01] J. Kienzle, R. Jiménez-Peris, A. Romanovsky, and M. Patiño-Martínez. Transaction Support for Ada. In *Proc. of Int. Conf. on Reliable Software Technologies, LNCS 2043*, pages 290–304. Springer, 2001.

[Lis88] B. Liskov. Distributed Programming in Argus. *cacm*, 31(3):300–312, 1988.

[LMWF94] N. Lynch, M. Merrit, W. E. Weihl, and A. Fekete. *Atomic Transactions.* Morgan Kaufmann, 1994.

[LS83] B. Liskov and R. Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM TOPLAS*, 5(3):382–404, 1983.

[MHL$^+$98] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. In *Recovery Mechanisms in Database Systems*, pages 145–218. Prentice Hall, 1998.

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

[PJA98] M. Patiño-Martínez, R. Jiménez-Peris, and S. Arévalo. Integrating Groups and Transactions: A Fault-Tolerant Extension of Ada. In *Proc. of Int. Conf. on Reliable Software Technologies, LNCS 1411*, pages 78–89. Springer, 1998.

[PJA01] M. Patiño-Martínez, R. Jiménez-Peris, and S. Arévalo. Group Transactions: An Integrated Approach to Transactions and Group Communication (in press). In *Concurrency in Dependable Computing.* Kluwer, 2001.

[PSWL95] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little. The Design and Implementation of Arjuna. *USENIX Computing Systems Journal*, 8(3):255–308, 1995.

[Tra95] TransArc Corporation, Pittsburgh, PA 15219. *Encina Transactional-C Programmers Guide and Reference*, 1995.

[WS94] S. M. Wheater and S. K. Shrivastava. Exercising Application Specific Run-time Control Over Clustering of Objects. In *Proc. of the Second International Workshop on Configurable Distributed Systems* , March 1994.