# Making Progress in Cooperative Transaction Models

Gail E. Kaiser
Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027
kaiser@cs.columbia.edu

Dewayne E. Perry
AT&T Bell Laboratories
Room 3D-454
600 Mountain Avenue
Murray Hill, NJ 07974
dep@allegra.att.com

In the classical transaction model, transactions are consistency preserving units: a transaction is made up of a series of actions which, when executed in isolation in a reliable environment, transforms the database from one consistent state to another [2]. A classical transaction management system guarantees the *appearance* of isolation and reliability, even though multiple transactions execute concurrently and hardware and software components fail. It does this by enforcing atomicity and serializability: atomicity means that either an entire transaction apparently executes to completion or not at all, while serializability means that the effects of the transactions are viewed as if the transactions had executed in some serial order, one completing before the next begins. This is accomplished by considering the objects read and written by concurrently executing transactions, and ensuring that either all updates are completed or none are, and that the read and write dependencies among the set of transactions correspond to some serial order of the transactions.

There have been many proposals for *extending* the transaction model from its original data processing applications to software development, CAD/CAM and other forms of cooperative work. The notion of ‘‘transaction’’ is intuitively appealing in these domains, since *forward progress* often depends on making a related set of changes to a program, design or document in such a way that it is transformed from one consistent state to another. As with data processing applications, ‘‘consistency’’ depends on the requirements of the domain, such as a new system configuration passing regression and acceptance tests before the changed modules can be committed.

However, it is well-known that atomicity and serializability are inappropriate for interactive cooperative work [9, 13]. A task that might be treated as a ‘‘transaction’’, such as responding to a modification request, may be of *long duration* — hours to days to weeks — while information must be *shared* among the perhaps large numbers of personnel who participate in the concerted effort during the process of making the changes. Such a task is typically broken down into several subtasks, one per developer, that are carried out in parallel, and together bring the software system under development from one consistent state to another.

But even these subtask ‘‘transactions’’ may be very long and need to exchange information while they are in progress, and in any case one subtask is unable by itself to maintain global consistency. Thus the conventional understanding of transactions as failure recovery and serialization units is unacceptable for cooperative work applications. In particular, recovering from failures by rolling back to the beginning of a (sub)task and starting again is rarely appropriate for human developers who would then have to redo much of their work, while serializing developer (sub)tasks does not permit cooperation while they are in progress.

The notion of *cooperative transactions* has been devised (under many different names) not only to provide the same intuition as conventional transactions but also to support the requirements of cooperative work. The gist of many cooperative transactions models is that transactions are assigned to

groups, where the transactions within a group employ a different concurrency control policy among themselves than with respect to transactions in other groups [4, 14]. Concurrency control is typically *relaxed* within a group, *e.g.*, allowing simultaneous updates to multiple versions of an object or allowing reads of uncommitted updates. Among groups, however, a stricter policy such as serializability is common. Cooperative transaction groups allow collaboration among the designated members of a group, through exchange of partial results, but enforces isolation among groups. See [1] for a survey of the literature on concurrency control policies and mechanisms for cooperative work.

We are concerned in this paper with one particular shortcoming of nearly all the cooperative transaction schemes. The shortcoming we have in mind is due to the problem of *human management* of *in-progress* software development processes. Cooperative transactions are designed to isolate groups from other groups and individuals, to allow cooperation among the members of a group while they carry out their tasks, and to implement the notion that the finalized set of updates made by group members is completed and released atomically. The *external view* outside the group sees only the released system, and not the partial results of in-progress work. However, the human managers of a software development project cannot wait for the released version of the system! They must be able to determine progress at a much finer granularity, dependent on the policies of the organization.

One possible approach would be to place the managers and all the personnel they manage in the same group, but for large organizations this defeats the purpose of concurrency control. Everyone may arbitrarily overwrite everything, without satisfying the consistency constraints of *forward progress*! Alternatively, each manager could be placed individually in every relevant group, as is possible in the participant transactions model which permits overlapping groups [7], but the symmetric cooperation implied between these managers and the other group members may be undesirable. Not only can the manager see the up-to-the-minute work of the developers, but the developers may inspect whatever the manager is doing; this is not usually acceptable for real projects.

Further, it would be preferable to provide the managers with an abstract view concerned with the software development *process* as opposed to the details of development *products* [10]. Managers are generally concerned with the results that have been accomplished thus far, and are not terribly interested in which versions of which files are currently being edited. Thus we propose that any cooperative transaction model should be augmented with a distinct *internal view* that supports the human management process, rather than trying to impose such access within the particular cooperative transaction model.

We have devised a general solution to this problem by analogy to the internal view of conventional transaction management systems that implement the classical transaction model. A transaction manager monitors the status of the currently in-progress transactions as well remembering the updates made by the previously committed transactions. The transaction manager interacts with various resource management systems, since transactions compete for computation cycles, primary memory, etc., and maintains internal structures to guarantee atomicity and serializability, including locks, logs, queues, shadow copies of data items, timestamps, distributed transaction coordinators and cohorts, and so on. The transaction manager sees a highly dynamic database, while an end-user of even a cooperative transaction system sees a relatively static database, since the database appears to change only when an update to an individual data item is completed (e.g., cooperating users probably would not share editing buffers, but only saved files).

We propose to solve the *impedance mismatch* between cooperative transaction models and the requirements of human management by *unveiling* the internal transaction management view at an appropriate level of abstraction. Like the transaction management system, the human manager can then use this internal view to manage resources, detect errors, recover from faults and in general monitor progress, but in terms of the software development process rather than data management structures. That

is, the human manager manages human and machine resources, detects erroneous interface and work breakdown assumptions by development staff, recovers from these faults by reassigning responsibility and correcting misconceptions, and in general keeps track of how far behind the schedule has slipped.

The human manager can go further than most transaction management systems, to *restructure* and change the direction of software development tasks, for example, to dramatically scale down the planned software product. On the other hand, transaction management systems can *guarantee* certain invariants, such as ‘‘all faults will be recovered’’ and ‘‘all deadlocks will be detected’’, while human management is itself fallible.

More specifically, we notice that every traditional transaction management system has two views, the *transaction view* of committed transactions and the *transcendent view* of in-progress transactions. The transaction view is explicit and externally visible; the transcendent view is implicit and internal to the transaction manager itself. In traditional applications, there is no reason to make the transcendent view explicit, and many good reasons to keep system implementation details hidden from applications programmers as well as end-users. The goal of a traditional transaction management system is that if the application programmer defines his program according to the transaction view using the primitives provided (*e.g.*, begin-transaction/end-transaction blocks), then the transaction manager will use its transcendent view to monitor global progress in order to guarantee atomicity and serializability. It is not necessary for the application programmer to be concerned with the details of how this is accomplished, and hiding this transcendent view is generally believed to ease the application programming effort.

A cooperative transaction manager should have similar transaction and transcendent views that support a cooperative work concurrency control policy and a consistency model specific to the particular cooperative work domain. Our solution to the human manager’s dilemma is to *extend* such systems to uncover a portion of the already existing transcendent view to make it explicit and visible to selected end-users (*i.e.*, the managers). In particular, we provide an *abstract view* of the internal structures and mechanisms that implement the cooperative transaction manager’s concurrency control protocol, while continuing to abstract away from the lower level layers that support the failure recovery protocol, physical data management, and so forth, which vary with the implementation. The result enables the human manager to monitor the progress (or lack thereof) of the software project, and take appropriate action, in the context of whatever cooperative transaction mechanism is employed.

Our preliminary ideas have been implemented in the INFUSE software development environment [6], which supports change management and integration testing. INFUSE is intended to support very large teams of developers, where it is crucial for the environment to enforce policies regarding the degree and style of cooperation among the developers [11]. This is achieved in INFUSE as follows.

A set of modules is selected in advance to be modified as part of a scheduled change. This change set is automatically partitioned into a hierarchy of what we call experimental databases using a simple module interdependency metric [8]. Developers make their changes to leaf experimental databases consisting of one or a few modules, invoke static semantic consistency checks and perform unit testing, and deposit their changes to the parent experimental database only when their work satisfies pre-specified constraints. Within a shared experimental database, the multiple developers *integrate* their changes, invoking inter-module static semantic consistency checks and applying regression and integration test suites, before the set of modules can be deposited to the next level. At the top of the hierarchy, acceptance tests must be passed before all the changes can be deposited to form the new baseline version of the system. The use of a module interdependency metric in forming the hierarchy follows the theory that changes to strongly coupled modules are more likely to affect each other, and thus should be integrated as early as possible, while changes to weakly coupled modules can be delayed until later on when higher levels of the

hierarchy are integrated.

Thus, INFUSE isolates individual developers and groups of developers following an isolationist concurrency control policy. The hierarchy is strictly partitioned, so it is never the case that more than one writer has access to a given module in a leaf experimental database, and readers who own sibling experimental databases cannot access the module until all changes have been completed in the leaf and deposited. Whenever it becomes clear at higher levels of the hierarchy that additional changes are required, the current experimental database is repartitioned below that point to enforce such isolated access while further changes are made.

However, INFUSE recognizes the pragmatic requirements of human managers to keep tabs on the progress of the scheduled change. Managers and other privileged users may make queries that cut across the hierarchy of experimental database, in order to determine the up-to-the-minute status of all modules in the change set. The managers can display a snapshot of the hierarchy at any given moment, or request a history of previous partitionings and repartitionings due to repeated changes (the ‘‘yo-yo’’ effect) required below a selected node in the hierarchy. Further, any developer may request creation of a workspace [5] that gathers together a selected set of disjoint experimental databases, with the permission of the other developers affected, in order to carry out early consistency checking and testing with respect to modules otherwise isolated from each other until higher levels of the hierarchy. This is useful when it is known that the scheduled changes will result in greater coupling among these modules or otherwise specifically involve these modules' interfaces.

Thus INFUSE supplies a transcendent view to managers of software development projects, to aid them in monitoring the *progress* of the software development process. This transcendent view is lacking in all other cooperative transaction models that we know of, even though it is clearly required for most practical applications. Unfortunately, the transcendent view we have developed is not sufficient in itself since the access it provides is read-only. In particular, it is not possible in INFUSE for a manager to modify the *organization* of in-progress transactions when a problem is discovered. Therefore, the transcendent view capability must be coupled with some facility for restructuring in-progress transactions, such as the split-transaction and join-transaction operations [12].

The split-transaction operation allows one on-going transaction to be split into two or more transactions as if they had always been independent, separating the data items accessed by the original transaction among the new transactions in a serializable manner. New developers may take over the new transactions, to improve progress towards the goal of a coherent working system. The join-transaction operations permits two or more on-going transactions to be joined into one, combining the data items accessed by the originally separate transactions as if they had always been part of the same transaction, so that the changes are released together.

The join-transaction operation is relatively easy to implement, but the split-transaction operation requires support to aid in the determination as to whether the desired split is valid. Both operations require aid from the software development environment in notifying affected developers of the potential changes to their work assignments and checking whether these changes make sense from the viewpoints of the individual developers. In INFUSE, the split-transaction and join-transaction operations would be implemented by dividing and merging experimental databases, respectively.

The transcendent view augmented with these transaction restructuring operations seems sufficient to support practical human management considerations towards making forward progress during software development. It seems possible to apply these ideas to a range of cooperative transaction models, not just INFUSE, and to support other cooperative work applications, not just software development. It should not

be very difficult to implement the transcendent view as part of a transaction manager already supporting a cooperative model, by making available an abstraction of the existing internal processes and structures. The feasibility of augmenting another transaction model with the split-transaction and join-transaction operations has previously been shown [3].

## Acknowledgments

Yoelle Maarek, Bill Schell and Bulent Yener worked with us on the implementation of INFUSE. We would also like to thank Andrea Skarra for her comments on a draft of this paper.

Kaiser is supported by National Science Foundation grants CCR-9000930, CDA-8920080 and CCR-8858029, by grants from AT&T, BNR, DEC, IBM, SRA, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

## References

[1] Naser S. Barghouti and Gail E. Kaiser. Concurrency Control in Advanced Database Applications. *Computing Surveys* , 1991. In press.

[2] Philip A. Bernstein, Vassos Hadzilacos and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading MA, 1987.

[3] Panayiotis K. Chrysanthis and Krithi Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In Hector Garcia-Molina and H.V. Jagadish (editor), *1990 ACM SIGMOD International Conference on Management of Data*, pages 194-203. Atlantic City NJ, May, 1990. Special issue of *SIGMOD Record*, 19(2), June 1990.

[4] Amr El Abbadi and Sam Toueg. The Group Paradigm for Concurrency Control Protocols. *IEEE Transactions on Knowledge and Data Engineering* 1(3):376-386, September, 1989.

[5] Gail E. Kaiser and Dewayne E. Perry. Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution. In *Conference on Software Maintenance*, pages 108-114. IEEE Computer Society Press, Austin TX, September, 1987.

[6] Gail E. Kaiser, Dewayne E. Perry and William M. Schell. Infuse: Fusing Integration Test Management with Change Management. In *COMPSAC 89 The 13th Annual International Computer Software & Applications Conference*, pages 552-558. IEEE Computer Society, Orlando FL, September, 1989.

[7] Gail E. Kaiser. A Flexible Transaction Model for Software Engineering. In *6th International Conference on Data Engineering*, pages 560-567. IEEE Computer Society, Los Angeles CA, February, 1990.

[8] Yoelle S. Maarek and Gail E. Kaiser. Change Management for Very Large Software Systems. In *7th Annual International Phoenix Conference on Computers and Communications*, pages 280-285. Computer Society Press, Scottsdale AZ, March, 1988.

[9] Erich Neuhold and Michael Stonebraker (editors). Future Directions in DBMS Research. *SIGMOD Record* 18(1), March, 1989.

[10] Dewayne Perry (editor). *5th International Software Process Workshop: Experience with Software Process Models*. IEEE Computer Society Press, Kennebunkport ME, 1989.

[11] Dewayne E. Perry and Gail E. Kaiser. Models of Software Development Environments. *IEEE Transactions on Software Engineering* , March, 1991. In press.

[12] Calton Pu, Gail E. Kaiser and Norman Hutchinson. Split-Transactions for Open-Ended Activities. In Francois Bancilhon and David J. Dewitt (editor), *14th International Conference on Very Large Data Bases*, pages 26-37. Los Angeles CA, August, 1988.

[13] Lawrence A. Rowe and Sharon Wensel (editors). 1989 ACM SIGMOD Workshop on Software CAD Databases. February, 1989.

[14] Andrea H. Skarra and Stanley B. Zdonik. Concurrency Control and Object-Oriented Databases. *Object-Oriented Concepts, Databases, and Applications.* ACM Press, New York, 1989, pages 395-421, Chapter 16.