Extended Transaction Models for Software Development Environments

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027

December 1988

CUCS-404-88

Abstract

This technical report consists of two papers discussing concurrency control facilities for multiuser software development environments. *A Marvelous Extended Transaction Processing Model* briefly sketches the previously developed commit-serializability model and then applies it to the MARVEL kernel for software development environments. *A Participant Semantics for Serializable Transactions* describes our first pass at a different extended transaction model that moves users *inside* the system, so certain users may participate in transactions and the interactions among transactions while all other users observe only a serial order for the transactions.

Prof. Kaiser is supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, DEC, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

A Marvelous Extended Transaction Processing Model

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027
kaiser@columbia.edu
212-854-3856

25 October 1988

Abstract

The key flaw in programming environment research is the lack of a transaction model supporting fault tolerance, concurrency control, consistent publication of changes and user-initiated rollbacks for software development activities. The atomicity properties of the classical transaction model make it unsuitable for industrial software development efforts. We sketch an extended transaction model with a *commit-serializability* semantics and describe the application of this model to the existing MARVEL architecture for programming environments. MARVEL realizes rule-based process modeling and integrates commercial-off-the-shelf tools *via* controlled automation.

Track: Software Engineering

This research is supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, DEC, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

1. Introduction

I was recently asked by a colleague (whose research area happens to be transaction processing) why it is that very few academic and laboratory results in the area of integrated programming environments seem to have been adopted for practical industrial software development. My response was that they have been applied successfully in certain communities, most notably commercial knowledge based systems, but primarily only single-user programming environments have seen practical use. Multiple-user programming environments seem to be an orphan technology, with apparently little impact of research on industrial development efforts.

In my opinion, the key problem with integrated programming environments for multiple users is the lack of a suitable transaction processing model. Research results in transaction processing have been adopted for commercial database and operating systems to provide fault tolerance, concurrency control, consistent commitment of changes and application program-initiated rollback facilities. Fault tolerance, concurrency control, consistent release of changes and user-initiated (or tool-initiated) rollback facilities are similarly crucial for multiple user programming environments. Individual software development tools provide some of these facilities, in the forms of, for example, checkpointing, version control utilities, system build, and undo/redo. The crippling problem with these mechanisms is that checkpointing, version control and undo/redo generally address only individual files, rather than the complete set of resources updated during the software development activity.

It is sometimes suggested that the transaction processing model successfully applied in databases and operating systems be applied directly to software development. There are several severe difficulties with this approach.

- Fault tolerance in traditional transaction processing models implies all or nothing, in the sense that if the machine crashes or there is some other failure, the atomic transaction is rolled back and retried. This seems the best approach for a transfer among bank accounts, but is entirely inappropriate when the 'transaction' consists of fixing a bug by browsing and editing a number of source files, compiling and linking, executing test cases and generating traces, *etc.*, which may easily take several hours and sometimes several weeks. No programmer would accept a system that threw out all his past work when the system crashed and expected him to start over!

- Concurrency control in traditional transaction processing implies that separate users or applications are effectively isolated from each other, and the transactions appear to have been executed in some serial order. When one transaction attempts access to a resource already read or written by another transaction that has not yet committed, the first transaction may be blocked until the resource is unlocked or rolled back to try again. Again considering our bug fixing 'transaction', it is not acceptable for a programmer to be locked out from editing a source file just because some other programmer had previously read the file but has not yet finished his changes to other files! It is equally inappropriate to break deadlocks by throwing out one or more of the programmers' efforts.

- In traditional transaction processing, consistent commitment implies simultaneously making all of a transaction's updates publicly available for reading and subsequent update by other transactions only when the first transaction completes and commits. In contrast, programmers must be able to release certain source modules so they can be viewed and/or compiled and linked by other programmers cooperating on the same subsystem, while continuing in progress work on other files. At the same time, it is necessary to prevent publication of the partial results outside the cooperating group.
- Finally, an abort of a traditional transaction in effect deletes all changes made during the transaction, so they are never available to other transactions; in some transaction processing models it is possible for other transactions to see the partial results, but then an abort triggers cascaded rollbacks. In fixing a bug, however, a programmer may realize that his original ideas about the cause of the problem were incorrect and decide to start over — but that does not mean he wants to throw away the incorrectly modified versions of the source modules! He may want to keep them available somewhere, even though they were not checked in, for reference on his second attempt.

Thus the traditional transaction model is not suitable for software development, at least where the 'transactions' are at the granularity of bug fixes, completion of a milestone, or release of a product. An *extended* transaction processing model is necessary to make integrated programming environments practical in the same way that research results in database and operating systems have achieved widespread commercial application.

This paper describes an admittedly incomplete model that is a step towards this goal, and applies this model to our MARVEL software engineering environment [Feiler 86, Kaiser 87a, Feiler 87, Barghouti 88, Kaiser 88a, Kaiser 88b, Kaiser 88c]. Our extended transaction model addresses concurrency control, consistent release and user-initiated rollback through a semantics for *commit-serializability* supported by two new transaction processing operations, *split-transaction* and *join-transaction*, that we recently introduced [Pu 88]. In this previous paper, we presented the new operations only in the context of programmed transactions, where here we consider both programmed transactions such as might be realized by 'process programming' [Osterweil 87] and interactive transactions initiated and controlled by a human user or a cooperating group of human users. We have not yet extended our semantics to fault tolerance, where some form of 'savepoints' will be necessary.

First we give an overview of MARVEL, then we describe the commit-serializability extended transaction model, and finally we apply the model to MARVEL. We conclude by summarizing related work and listing the contributions of this paper.

2. Overview of MARVEL Architecture

MARVEL's primary goal is to support realization of process modeling for controlled automation of software development activities. Such controlled automation eases integration of commercial-off-the-shelf (COTS) tools. The processing modeling language, called the MARVEL Strategy Language (MSL), is rule-based. Each activity, typically a tool invocation, is defined as having zero or one precondition and zero or more postconditions. The precondition corresponds to the condition of classical production systems [Waterman 86], while the actual activity plus the postconditions corresponds roughly to the action.

The precondition indicates those predicates that must be true in order to carry out the activity. Each postcondition indicates those assertions that are made true by completing the activity. There are multiple mutually exclusive postcondition to indicate the multiple possible results of many activities (for example, compilation produces either object code or error messages, but it is not possible to determine which without executing at least the front end of the compiler).

Automation is supported by forward and backward chaining on the rules. When the user attempts to initiate an activity, MARVEL checks whether its precondition is satisfied. If not, MARVEL attempts to satisfy it by backward chaining and consequent automatic initiation of activities. Satisfaction may not always be possible, in which case MARVEL is able to use its rules to explain the problem to the user (or provide help on how to use the tools required to perform the activities [Wolz 88]). Once an activity has completed, MARVEL asserts one of its postconditions. In the background, MARVEL uses forward chaining to automatically invoke activities whose preconditions are now satisfied. We call this application of forward and backward chaining *opportunistic processing*, because MARVEL automatically carries out activities as the opportunity arises.

Automation is controlled both by the rules and by *implicit queries* that MARVEL makes during both forward and backward chaining. When the cost of some automatic activity is likely to be over some threshold, MARVEL informs the user of the situation and requests confirmation before continuing.

COTS tools are supported by *envelopes* that interface between MARVEL's objectbase, which acts as the working memory of the rule-based system, and the actual input/output requirements of the tools. The envelopes also indicate to MARVEL which of the alternative postconditions should be asserted on working memory.

MARVEL's behavior is both user-selectable and user-programmable. A distinguished user, called the superuser, writes a number of MSL modules, called *strategies*. A strategy consists of a data model for the relevant portion of working memory, a collection of rules and a collection of tool envelopes. A strategy may import other strategies that provide some of the facilities it uses. Different strategies may support the same tools with entirely different rules reflecting different management policies; different rules may require different attributes for some data objects and different relationships among some data objects.

A default set of strategies is associated with each MARVEL objectbase when it is created, but this default set can be changed and the user can load and unload individual strategies at any time in effect changing MARVEL's behavior interactively. Any user can act as a superuser and create new strategies, to tailor MARVEL to his own favorite tools or preferred mode of behavior. In cases where policies must be enforced, this user-extensibility could easily be turned off — say be requiring a password to change strategies and/or to add new strategies to the strategy library.

For example, consider the task of building a programming environment for developing and maintaining software systems written in C. The tools used by the environment are those readily available on Unix: a text editor, a type checker, a compiler, a linker and a mail program. At a minimum, we would like the new environment to provide the following assistance.

- A manager decides to upgrade an existing software system and divides the changes that need to be made among the programmers in his group. He assigns each programmer a specific module (a C source file) to work on using a command like "assign <programmer> <module>". The environment responds to this command by displaying an error message if the module has already been assigned, and otherwise reserving the module for the programmer and sending mail notifying the programmer of his assignment. Each programmer is supposed to upgrade the module he is assigned to achieve the desired change.

- A programmer gives the "edit <module>" command. The environment automatically displays any known errors in the module before calling the editor. The programmer edits the module and saves the changes. The environment, knowing that the module has been modified, invokes the type checker and informs the programmer of (1) any errors detected previously that have not been corrected and (2) any new errors introduced. The programmer is expected to work on the module further to eliminate all static errors. (In a realistic environment, the programmer would also carry out unit testing, perhaps with a test management tool, as well as a debugger.)

- If the type checker does not detect any errors, the environment sends mail to the manager informing him that the programmer has completed his particular assignment. When all outstanding assignments have been completed, the environment automatically recompiles and relinks the program and sends mail to the manager and his programmers informing them that the upgraded system is ready for integration and acceptance testing.

MARVEL provides facilities to generate this environment in two phases. First, the superuser writes an MSL description that (1) specifies the organization of the database for the project in terms of entities, attributes and relations (*e.g.*, a C program consists of modules, each of which may contain macros, types, variables and functions), and (2) models the software development process for that particular project in terms of tools and rules (*e.g.*, a precondition of the editor is that the indicated module be assigned to the current programmer and a result of the editor is that the module's status is not-checked, implying it is necessary to invoke the type checker).

Second, a user starts up the MARVEL kernel and enters the load command to instantiate the kernel with this MSL description. He can then use this instantiated MARVEL to produce the target software system. In our example, the manager would probably give the load command and then save the instantiated MARVEL for later use by himself and his group.

Now consider a variant of this programming environment for C, where certain commands such as "assign" are restricted to managers and should not appear in the menus available to programmers. The superuser can enforce this by writing two MSL descriptions, one for programmers and one for managers, where each defines the same entities but the description for managers includes additional tools and rules not in the description for programmers. Whenever a manager uses MARVEL, he loads the manager-specific description and whenever a programmer uses MARVEL, he loads the programmer-specific description (or MARVEL could load the appropriate description automatically). The manager and his programmers would thus employ two different descriptions with respect to the same software development project and the same database.

This separation might lead to inconsistencies as new and improved tools become available, and the entities and rules are upgraded so the project can take full advantage of the new tools. The superuser must make sure the shared portion of the two distinct descriptions remains identical. Strategies ease the superuser's burden by modularizing the descriptions, so that MARVEL can be instantiated by a group of strategies that together define the full collection of entities, tools and rules. The superuser would define one basic strategy that gives the entities, attributes and relations shared between programmers and managers, as well as the tools and rules that support the programming process. The superuser would also define a separate strategy for managers that could be loaded along with the programmer-specific strategy to provide the full capabilities required by managers.

MARVEL as described in previous papers does not support multiple users. A previous implementation of MARVEL did actually support multiple users because it was built on top of Smile [Kaiser 87b], a multiple user programming environment for C developed as part of the Gandalf project [Habermann 86] at CMU. Smile had trivial support for multiple users, and in any case had a hard-wired objectbase. When MARVEL was reimplemented with its own general but 'quick-and-dirty' objectbase manager, we lost the multiple user capability. Rather than hack it back in, as was done for Smile, we decided to pursue a general extended transaction model applicable to a wide range of programming environment efforts. The application of commit-serializability to MARVEL described later is not yet implemented.

3. Commit-Serializability Transaction Model

The term *commit-serializability* is chosen to denote an extended transaction model where all committed transactions are in fact serializable in the standard sense, but these transactions may not correspond in a simple way to those transactions that were initiated. In particular, the initiated transactions may be divided during operation and parts committed separately in such ways that these transactions are not serializable.

To make this more clear, consider two in-progress transactions T1 and T2. T1 is divided under program or user control into T3 and T4, and shortly thereafter T3 commits while T4 continues. T2 may view the committed updates of T3, some of which were made by T1 before the division, and then itself commits. T4 may then view the committed updates of T2 before it commits. T2, T3 and T4 are serializable, but T1 and T2 are not. The originally initiated transaction T1 in effect disappears, and in particular is neither committed nor aborted.

Commit-serializability is supported by two new transaction processing operations, split-transaction and join-transaction, in addition to the standard *initiate-transaction*, *commit-transaction* and *abort-transaction* operations. The split-transaction operation supports the kind of division described above; the inverse join-transaction operation merges a completed transaction into an in-progress transaction to commit their results together.

The two-way versions of the split and join operations take the following arguments. We do not address n-way versions in this paper, but the extension is straight-forward.

```
Split-Transaction(A: ( AReadSet, AWriteSet, AProcedure ),
                  B: ( BReadSet, BWriteSet, BProcedure ))

Join-Transaction(S: TID)
```

When the split-transaction operation is invoked during a transaction T, there is a TReadSet consisting of all objects read by T but not updated and a TWriteSet consisting of all objects updated by T (alternatively, TReadSet could be all objects locked for reading by T and TWriteSet all objects locked for writing, whether or not they had actually been read or written). TReadSet is divided, not necessarily disjointly, into AReadSet and BReadSet. TWriteSet is divided disjointly into AWriteSet and BWriteSet. In the special case where A is immediately committed, say by a variant operation *split-transaction-and-commit*, objects in AWriteSet may also appear in either BReadSet or BWriteSet. In the case of a programmed transaction, AProcedure and BProcedure indicate the code for each transaction to execute following the split. In the case of a user-controlled transaction, these two parameters are omitted.

Say a programmer U has read modules M and N and updated modules N and O. He has compiled the changed N and O, linked them together with the old object code for M, and is in the process of debugging. Another programmer V requests access to module N. Since U is fairly sure he is done making changes to N, but needs to continue work on M and O, he splits and commits a transaction that updates N. V then reads N, decides to use this new version rather than the old one for testing his own changes to other modules, recompiles N and tests his subsystem. Later V commits N and U commits M and O.

It is possible to invoke an abort-transaction operation on transaction A or B resulting from a split-transaction. This does not automatically abort the other transaction, since they are now independent. However, if B is still ongoing when A aborts, it may be desirable to notify B that A has aborted and give B the option of subsequently aborting.

When the join-transaction operation is invoked during a transaction T, target transaction S must be ongoing. TReadSet and TWriteSet are added to SReadSet and SWriteSet, respectively, and S may continue or commit.

Say a programmer U has read modules M and N and updated modules N and O. He has compiled the changed N and O, linked them together with the old object code for M, and completed debugging. Another programmer V is working on other changes to the same subsystem. Since U is done, he joins M, N and O to V's resources, so all changes to the subsystem will be published together. U then goes on to his next task.

In the cases of both split-transaction and join-transaction, the originally initiated transaction T is divided or merged, respectively, so the net effect is as if it had never existed. The tables, logs, *etc.* used in by the transaction manager implementation are updated as necessary to expunge knowledge of T and replace it with knowledge of A and B or S, respectively.

Split-transaction and join-transaction may be used as part of *nested transactions* [Moss 81]. In the former case, both A and B have no parent or both have the same parent P, which was originally the parent of T, and the same set of siblings as T. In the latter case, either T has no parent, or T has the same parent P and the same set of siblings as S.

Again consider the possibility of invoking an abort-transaction operation on transaction A or B resulting from a split-transaction. When A and B are both nested inside the same parent P, then it is possible to notify P even though B has already committed. This may prompt P to issue a compensating transaction C, to undo the effects of B or take some other action.

Split-transaction and join-transaction may be invoked at different times during the same software development activity. Say a programmer U has read modules M and N and updated modules N and O. He has compiled the changed N and O, linked them together with the old object code for M, and is in the process of debugging. Another programmer V requests access to module M. Since U does not need it right now, he splits and commits a transaction that reads M. V then modifies M, recompiles and tests it, and then joins the updated M with U. Now U can make further changes to M, and the changes to M, N and O commit together.

4. Application of Commit-Serializability to MARVEL

When a user carries out a task using MARVEL, say to fix a bug, he first gives the **initiate-transaction** command. After the transaction commences, the user proceeds to browse through the MARVEL objectbase, looking at the bug report and some of the modules implicated in the report. He runs some test cases through the executable version of the system associated with the bug report.

So far, everything the user has done has been read-only in the sense of no obvious updates to software artifacts. However, he may have unwittingly caused changes to several objects due to forward and backward chaining by the rules associated with the activities he has carried out.

For example, the user's request to execute the system may have triggered backward chaining that ultimately compiled and linked the appropriate module versions adding to the derived object pool and updated the status attributes of the relevant module versions and system configuration versions. Commit-serializability would permit the user's transaction to split automatically so that the newly derived objects (presumably derived at some previous point but then deleted or garbage collected due to space limitations) are available to any other user that needs them.

A less intuitive example is that the user's request to read the bug report may have updated the status of the bug report and sent mail to his manager to indicate that the programmer had commenced on this task. Again the transaction should split, because an abort initiated by the programmer certainly does not negate the fact that he started to work on the task.

Continuing with our example, the user proceeds to edit several source files, which backward chaining causes to be checked out of the version control tool and forward chaining causes to be recompiled and relinked. There may be several cycles of editing as newly introduced syntactic errors are removed. Then the user continues running test cases and maybe inspects system execution using a debugger.

Sometime during these activities, another user operating in another transaction attempts to edit one of the source files already checked out of the version control tool. He is now given the choice of forking a version branch or requesting a split in the transaction that has locked the files. In the latter case, this split may be automatic or may require agreement of the original user. In general, the interactions with the transaction manager must be programmed in the tool envelopes, except that they are handled automatically by MARVEL in the cases of precondition checking and postcondition assertion.

5. Related Work

We know of only one integrated environment that realizes a transaction model, the Cosmos/Eclipse environment [Walpole 88] at the University of Lancaster. The transient versions and time domain addressing used for the multiple version implementation [Reed 78] of serializable transactions is replaced in Cosmos by immutable versions and domain relative addressing on configurations and configuration histories. The primary disadvantage of this scheme is the non-serializability of the committed transactions. We avoid this disadvantage with commit-serializability, since the committed transactions are in fact serializable although not atomic.

Sun Microsystem's Network Software Environment (NSE) [Sun 88], with its integration environments and components, and Imperial Software Technology's IStar [Dowson 87], with its contract databases, are relatively easy to reformulate with a transaction model but to my knowledge this has not been done by the developers. Our own Infuse change management system [Kaiser 87c] is similar to NSE, but enforces a policy of integrating strongly connected modules and subsystems first before weakly connected components. We may take advantage of either the NSE or Infuse support for group as well as individual isolation in the future MARVEL implementation of commit-serializability.

A number of integrated programming environments provide automation akin to MARVEL's opportunistic processing. ISI's CommonLisp Framework (CLF) [Balzer 85] is a notable example, and CLF strongly influenced our work on MARVEL. Several proposed environments plan to incorporate realization of process modeling; one eminent example is the work of the Arcadia consortium [Taylor 86].

6. Contributions

The primary contributions of this paper are a superior formulation of our previously published split-transaction and join-transaction operations, a presentation of a commit-serializability semantics for transactions, and the application of commit-serializability to a previously published research architecture for programming environments.

Acknowledgments

Calton Pu collaborated on the development of the split-transaction and join-transaction operations, with input from Norman Hutchinson. Commit-serializability in its current form was influenced by discussions with Dan Duchamp, Maurice Herlihy, Peter Wegner and Bill Weihl. The MARVEL project was initiated jointly with Peter Feiler at the CMU Software Engineering Institute, and a 'quick-and-dirty' implementation was done by Steve Popovich. Nasser Barghouti collaborated in a redesign and supervised the first large-scale implementation; Bob Schwanke along with Nasser collaborated on a subsequent redesign, and Nasser and Mike Sokolsky recently completed the second implementation. Wendy Dilliard, Russel Goldberg, Christine Hong, Wai Keung Hui, Qifan Ju, Christine Lombardi, Alexander Mogieleff, Joe

Milligan, Michael Sacks, Tam Tran, and Timothy Yuan also contributed to the implementation efforts.

References

[Balzer 85] Robert Balzer.
A 15 Year Perspective on Automatic Programming.
IEEE Transactions on Software Engineering SE-11(11):1257-1268, November, 1985.

[Barghouti 88] Naser S. Barghouti and Gail E. Kaiser.
Implementation of a Knowledge-Based Programming Environment.
In *21st Annual Hawaii International Conference on System Sciences*, pages 54-63. Kona HI, January, 1988.

[Dowson 87] Mark Dowson.
Integrated Project Support with IStar.
IEEE Software :6-15, November, 1987.

[Feiler 86] Peter H. Feiler and Gail E. Kaiser.
Granularity Issues in a Knowledge-Based Programming Environment.
In *Second Kansas Conference on Knowledge-Based Software Development.* Manhattan KA, October, 1986.
Available as CMU Software Engineering Institute, SEI-86-TM-11, September 1986.

[Feiler 87] Peter H. Feiler and Gail E. Kaiser.
Granularity issues in a knowledge-based programming environment.
Information and Software Technology 29(10):531-539, December, 1987.

[Habermann 86] A.N. Habermann and D. Notkin.
Gandalf: Software Development Environments.
IEEE Transactions on Software Engineering SE-12(12):1117-1127, December, 1986.

[Kaiser 87a] Gail E. Kaiser and Peter H. Feiler.
An Architecture for Intelligent Assistance in Software Development.
In *9th International Conference on Software Engineering*, pages 180-188. Monterey CA, March, 1987.

[Kaiser 87b] Gail E. Kaiser and Peter H. Feiler.
Intelligent Assistance without Artificial Intelligence.
In *32nd IEEE Computer Society International Conference*, pages 236-241. San Francisco CA, February, 1987.

[Kaiser 87c] 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. Austin TX, September, 1987.

[Kaiser 88a]	Gail E. Kaiser, Peter H. Feiler and Steven S. Popovich. Intelligent Assistance for Software Development and Maintenance. *IEEE Software* :40-49, May, 1988.
[Kaiser 88b]	Gail E. Kaiser, Naser S. Barghouti, Peter H. Feiler and Robert W. Schwanke. Database Support for Knowledge-Based Engineering Environments. *IEEE Expert* 3(2):18-32, Summer, 1988.
[Kaiser 88c]	Gail E. Kaiser and Naser S. Barghouti. An Expert System for Software Design and Development. In *Joint Statistical Meetings.* New Orleans LA, August, 1988. To appear.
[Moss 81]	J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, MIT, April, 1981. MIT LCS TR-260.
[Osterweil 87]	Leon Osterweil. Software Processes are Software Too. In *9th International Conference on Software Engineering*, pages 1-13. Monterey CA, March, 1987.
[Pu 88]	Calton Pu, Gail E. Kaiser and Norman Hutchinson. Split-Transactions for Open-Ended Activities. In *Fourteenth International Conference on Very Large Data Bases*, pages 26-37. Los Angeles CA, August, 1988.
[Reed 78]	David P. Reed. *Naming and Synchronization in a Decentralized Computer System.* PhD thesis, MIT, September, 1978. MIT LCS TR-205.
[Sun 88]	*Introduction to the NSE* Sun Microsystems, Inc., Mountain View CA, 1988.
[Taylor 86]	Richard N. Taylor, Lori Clarke, Leon J. Osterweil, Jack C. Wiledon and Michal Young. Arcadia: A Software Development Environment Research Project. In *2nd International Conference on Ada Applications and Environments*, pages 137-149. IEEE Computer Society, Miami Beach, FL, April, 1986.
[Walpole 88]	J. Walpole, G.S. Blair, J. Malik and J.R. Nicol. Maintaining Consistency in Distributed Software Engineering Environments. In *8th International Conference on Distributed Computing Systems*, pages 418-425. San Jose CA, June, 1988.
[Waterman 86]	Donald A. Waterman. *A Guide to Expert Systems.* Addison-Wesley Pub. Co., Reading MA, 1986.

[Wolz 88] Ursula Wolz and Gail E. Kaiser.
A Discourse-Based Consultant for Interactive Environments.
In *Fourth IEEE Conference on Artificial Intelligence Applications*, pages 28-33. San Diego CA, March, 1988.

A Participant Semantics for Serializable Transactions (Extended Abstract)

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027
Kaiser@columbia.edu
212-854-3856

7 October 1988

Abstract

The paper presents a new semantics for serializable transactions where certain human users participate in the transactions, and are thus aware of their interleaved and/or concurrent operation, while all other human users remain observers to whom the transactions appear to have been executed in some serial order. Participants perform certain actions within transactions, and thus may view their own and other users' partial results. This notion of participation is useful for applying the transaction concept to the open-ended activities supported by environments for CAD/CAM, VLSI design, office automation and software development.

keywords: complex objects, concurrency control, dependency theory, transaction management.

Kaiser is supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, DEC, IBM, Siemens and Sun, by the Center of Advanced Technology and by the Center for Telecommunications Research.

1. Introduction

The intent of serializability is that a set of transactions should appear to have been performed in some serial order with respect to every external observer, even though the actual execution of the actions within the transactions has been interleaved and/or concurrent. The external observers may include programs, but have always been assumed to include any human users interacting with the system. We introduce a new semantics of serializability where certain users may be designated as *participants* in a specific set of transactions, meaning the transactions need not appear to have been performed in some serial order with respect to these participants. Other users remain *observers*, and the set of transactions appears serial. A particular user may be a participant for some sets of transactions and an observer for other sets simultaneously executed within the same system.

This distinction between participant and observer is useful for applying the transaction concept to *open-ended activities*, such as are supported by environments for CAD/CAM, VLSI design, office automation and software development. Open-ended activities are characterized by

- Uncertain duration. Locating and fixing a bug in a software system may take from hours to months.
- Uncertain developments. The set of modules viewed, compiled and executed, as well as the set of test cases, may not be foreseeable at the beginning of the debugging activity.
- Interaction with other concurrent activities. In large software projects, several programmers cooperate to fix a bug — they must see the latest versions of each others modules even though the versions will not be publically released (committed) until the bug has been repaired.

In current environments, open-ended activities are typically supported by *ad hoc* mechanisms even though their requirements include the fault-tolerance, concurrency control and provision for user-initiated aborts collectively guaranteed by transactions. However, traditional transactions where all human users are treated as external observers are not appropriate in the context of uncertain duration, uncertain developments and interactions among concurrent activities. The latter characteristic of open-ended activities is the most troublesome — two-way dependencies among concurrent activities is inherently inconsistent with an observed serial order of the activities. Therefore, we separate the users into participants and observers: the participants are involved in the non-serial interactions while the observers see a serial order.

2. Contributions

The primary contribution of the paper is a semantics for user participation in what are externally observed as serializable transactions, including the concepts of participant serializability, participant non-serializability and user serializability.

We have previously defined open-ended activities in another paper [Pu 88], where we introduced a different semantics for transactions that we now call *commit-serializability*. The basic idea there was that all committed transactions appear to have been executed in serial order with respect to all users (*i.e.*, there was no notion of participant), but in-progress transactions may *split* to commit separately a subset of their resources or *join* to commit their resources together. Due to the split operation, the original set of transactions that began operation may not appear serializable as they may be committed in parts.

3. Participants

For each transaction **T**, there is a set of resources **R(T)** read or viewed by the transaction and a set of resources **W(T)** written or updated by the transaction. The intersection between **R(T)** and **W(T)** may be non-empty, and resources may be modified. Transactions may be of arbitrarily long duration between the start of the transaction and its commit (or abort).

In the context of open-ended activities, the goal of a transaction is typically to complete some task, such as design a VLSI circuit or write a quarterly report. In many cases, these tasks involve several tools and more than one human user. Sometimes the tools may proceed without human intervention and may even be invoked automatically by the environment, for example a document formatter, while others require human interactions, for example a word processor. Different users may be simultaneously working on different parts of the same task, such as writing different sections of the same report, but it is necessary for the users to view each others partial results — say to make sure they're not duplicating effort, for example by discussing the same material in different sections when it should only appear in one, and to negotiate and solve problems that arise, for example one part of the production plan has to be down-sized due to financial constraints that became clear only while developing another part. These users who thus interact within the same task, *i.e.*, the same transaction, are known as participants in the transaction. Users who see only the final results of the task are known as observers of the transaction.

For each transaction **T**, there is a set of users **P(T)** who are participants in the transaction. Each user in this set is designated as participant $\mathbf{p_i(T)}$ for some **i**. **P(T)** may be selected in advance before the transaction begins, or accumulated during the course of the transaction. All other users who are not participants are in the set **O(T)**, the observers of the transaction.

A participant in a transaction may perform some or all of the actions within the transaction, for example, drawing an illustration or invoking a VLSI layout tool. Some participants may not actually perform any actions, but view the results of these actions as part of some other dependent transaction, for example, reading the source code of one module in order to decide what changes to make in another module. Note that such viewing takes places before the transaction commits; after the transaction commits, any observer can of course read the resources updated by the transaction.

For any set of transactions, there is some group of users who cooperate to complete the task reflected in the set of transactions, for example, all the steps from designing through fabricating through testing a chip, where there may be feedback among the steps until the chip both meets its economic requirements and operates correctly. All other users are not directly involved and only see the committed results of the transactions as if they had been executed in serial order; in this example, they might see only the resulting chip, or they might see the final cost expenditures of each step broken down as if there had been no feedback.

For any set of transactions **S** equal to $\mathbf{\{T_1, ..., T_n\}}$, there is a set of observers **O(S)**, which is the set difference of the union of $\mathbf{O(T_i)}$ for some **i** and the set of participants **P(S)**, where **P(S)** is the union of $\mathbf{P(T_i)}$ for some **i**. If the $\mathbf{P(T_i)}$ are disjoint, then **S** is said to be *participant serializable*; otherwise, the set **S** is *participant non-serializable.*

For any set of transactions **S** equal to $\mathbf{\{T_1, ..., T_n\}}$ where there is a user who is a member of $\mathbf{O(T_i)}$ for every **i**, **S** is *user serializable* with respect to that particular user. If there is a user who is a member of $\mathbf{P(T_i)}$ for any **i**, then set **S** is *user non-serializable* with respect to that user.

In this abstract, we consider enforcing that certain sets of transactions are participant serializable while detecting that certain other sets are participant non-serializable. This can be accomplished most easily for hierarchical tasks. In the full paper, we describe the corresponding issues of participant serializability for non-hierarchical tasks, where user serializability becomes an important concern.

4. Hierarchical Tasks

In many applications, a set of tasks will be purely hierarchical, with a root task, a number of non-terminal tasks and a number of leaf tasks organized as a tree. For the definition of participant serializable transactions given above, the set of transactions S represents the set of subtasks of a non-terminal task. A leaf task is defined as a participant serializable transaction **T** when all users in the set **P(T)** are members of the sets $\mathbf{O(T_i)}$ for all other leaf tasks $\mathbf{T_i}$. Each task in a tree of tasks is a participant serializable transaction **T** when all users in the set $\mathbf{P(T^*)}$, the union of the sets $\mathbf{P(T_i)}$ for all tasks in the subtree rooted at **T**, are members of the sets **O(T-[i)]** for all other tasks $\mathbf{T_i}$ that are not in the subtree rooted at **T**.

Consider, for example, the development of a large software system. The root task is the development itself, and it commits with the first release of the system. The root task is broken down into several subtasks representing the stages in the lifecycle of the system: say, requirements analysis, functional specification, detailed design, coding and unit testing, integration testing, quality assurance, and deployment. Although in the waterfall model of software development these stages are purely sequential, the more modern spiral model assumes feedback among the various stages in order to improve the quality and economic viability of the product and the productivity of the process. Thus there must be human participants, the software development team, who view the transactions representing these stages as interacting while there are human observers, perhaps corporate management, that are external observers and see the transactions representing these stages in this serial order. The customers might view the entire task as a single transaction, resulting in the software product, and not observe any of the subtransactions reflecting the subtasks.

Within each of the stages there are more levels of subtasks. For example, for coding and unit testing, the software system is divided into subsystems with the responsibility for development of source code and internal documentation assigned to a particular group. Within a group, the subsystem is further divided into modules assigned to particular programmers. Although a programmer may create and modify only his own modules, he must view at least the specification part (imports and exports) of the other modules in the subsystem — and sometimes in other subsystems. Some senior programmers may sometimes modify modules assigned to other programmers to solve difficult coding problems or to handle tricky interactions among modules. To apply the transaction concept to such open-ended activities, we need the notion of participant to represent these users, since many users must view the non-committed updates to resources made by other users and some users must even modify the non-committed updates of other users.

In general, a *primitive transaction* **T** consists of a partially ordered set of actions, each denoted $\mathbf{A_i}$. Each action is atomic and consists of reading a resource in **R(T)**, writing a resource in **W(T)**, or modifying a resource in the intersection of **R(T)** and **W(T)**. Each action has exactly one participant $\mathbf{p_i(T)}$, but there may be multiple participants in **T**. Each participant $\mathbf{p_i(T)}$ may initiate an action $\mathbf{A_i}$ that reads, writes or modifies the resources read, written or modified by other actions $\mathbf{A_j}$ within the transaction all ordered $\mathbf{A_j}$ *lt* $\mathbf{A_i}$. The ability of any particular participant to initiate actions on specific resources is also governed by access control constraints and the goals of the task represented by the transaction, but this is ignored here. Each observer in **O(T)** is not aware of the internal actions, but may view only the final form of the committed resources.

A *non-primitive transaction* **T** consists of a partially ordered set of subtransactions and actions, each again denoted $\mathbf{A_i}$. Each action is as described above for primitive transactions. Each subtransaction is either a primitive transaction or a non-primitive transaction. In either case, it has a set of participants $\mathbf{P(A_i^*)}$, which is the union of all the participants of each action or lower-level subtransaction within the subtransaction $\mathbf{A_i}$. We assume that it is determined in advance whether the subtransactions of **T** must be participant serializable, or may be participant non-serializable.

In the case where participant non-serializability should be detected but not prevented, such as for a set of cooperating subtasks, each participant in **P(T)** is permitted to initiate an action or subtransaction $\mathbf{A_i}$ that reads, writes or modifies the resources read, written or modified by actions or subtransactions $\mathbf{A_j}$ within **T**. If there is any user that participates in more than one subtransaction within **T** that are not totally ordered (*i.e.*, actually serial), then the set of subtransactions is not participant serializable. Note that this does not mean that **T** itself is not participant serializable with respect to its parent task. Each observer in **O(T)** is not aware of the internal actions and subtransactions, but may view only the final form of the committed resources of the entire transaction **T**.

In the case where participant non-serializability among the subtransactions should be prevented, a participant in **P(T)** may attempt to initiate an action or subtransaction $\mathbf{A_i}$ that reads, writes or modifies the resources read, written or modified by actions (but not subtransactions) $\mathbf{A_j}$. This is permitted if all $\mathbf{A_j}$ *lt* $\mathbf{A_i}$. A participant may attempt to initiate an action or subtransaction A_i that reads, writes or modifies the resources read, written or modified by subtransactions A_j. This is permitted only if the transactions A_j have committed. This means that users not work on several subtasks at the same time and users may not see the partial results of another user, except when they both participated in a subtransaction where participant non-serializability was acceptable. Again, each observer in **O(T)** is not aware of the internal actions and subtransactions, but may view only the final form of the committed resources **W(T)** of the entire transaction **T**.

In the full paper, we describe the analogous notions for hierarchical but layered tasks, where certain users are participants at and below a certain level while others are observers throughout the tasks at and below that level. In some applications, a set of tasks will not be hierarchical nor layered. Then the best we can say is that there is a set of users who are not participants in any tasks in the set, and for them the set of tasks appears as transactions executing in some serial order.

5. Implementation

Participants can be implemented in terms of any of the standard mechanisms for implementing transactions. We describe an implementation for hierarchical tasks in terms of two phase locking. In the full paper we describe an implementation supporting non-hierarchical tasks.

First we address the case where participant serializable transactions are enforced. A transaction proceeds normally, except that each participant in the transaction has the same transaction identifier **tid**. When a user attempts an action on a particular locked resource, his **tid** is compared to the transaction identifier attached to the resource when it was locked. The action is permitted if and only if the **tid**'s match. In the case where participant non-serializability is permitted among the subtransactions, only the **tid** of the transaction at the root of the subtree is considered. Thus it is possible for a participant in one subtransaction to see uncommitted or perhaps even aborted results of another subtransaction. In the case of aborted subtransactions, some notification scheme is desirable to inform each user that has seen the rolled back resources of the abort, but is not required for participant semantics.

In those cases where participant non-serializable transactions are detected but not prevented, actions involving conflicting **tid**'s are permitted. Some confirmation scheme is desirable to inform the user that he is initiating an action that will introduce participant non-serializability, and give the user a change to abort the action — but this is also not required the semantics. Instead, it is necessary to mark the transaction as participant non-serializable as soon as the conflict occurs.

6. Related Work

The most closely related notions to participant semantics for transactions are nested transactions, nested objects, versions, undo/redo facilities, and long transactions. The participation semantics presented here are actually orthogonal to the first two of these concepts, but would likely cooperate with the third when implemented as part of an environment supporting open-ended activities. Undo/redo facilities, except as they occur in long transactions, normally apply only within a primitive action so we do not consider them here. Long transactions are an alternative, typically non-serializable approach to open-ended activities.

Nested transactions (*e.g.*, [Moss 81]) must be defined in advance, often require strict serializability of the subtransactions, and do not permit user participation in subtransactions. Operations on nested objects (*e.g.*, [Martin 88]) also must be defined in advance, do not require serializability of the lower-level operations on abstract data types — only of the top-level transactions, but do not permit user participation in the operations. A stricter form of this (*e.g.*, [Badrinath 88]) requires such lower-level operations to commute. Adding participation would enhance either nested transactions or nested objects and improve their suitability for open-ended activities.

Persistent versions (*i.e.*, distinct from the transient versions used for the multiple version implementation of serializable transactions, *e.g.*, [Reed 78]) with reserve/replace semantics have not yet been adequately formalized. This relatively *ad hoc* notion has been, however, extremely successful in practice (*e.g.*, [Tichy 85, Rochkind 75, Leblang 84]). *Ad hoc* versions do in fact address to some extent all three characteristics of open-ended activities: uncertain duration, uncertain developments and interaction among concurrent activities. Once a version branch has been reserved, an arbitrary length of time until the corresponding replace is not a problem. Versions of additional resources can be reserved at any time, and there is no requirement that all reserved resources be replaced together, permitting interaction with concurrent activities among participants. Access control can be used to limit on, effectively preventing inappropriate access by observers. In some systems, configurations can be treated as a single unit for version control (*e.g.*, [Sun 88]). However, versions alone do not effectively meet the requirements of fault-tolerance, concurrency control and user-initiated aborts.

Long transactions do meet the requirements of fault-tolerance, concurrency control and user-initiated aborts. Some work on long transactions (*e.g.*, [Garcia-Molina 87]) addresses the uncertain duration and uncertain developments characteristics of open-ended activities, but not interaction among concurrent activities. Our previous work on commit-serializability covers all three. At least one long transaction mechanism is based on a version model [Walpole 88]. The transient versions and time domain addressing used for the multiple version implementation of serializable transactions is replaced with immutable versions and domain relative addressing on configurations and configuration histories. The primary disadvantage of this scheme is the non-serializability of the transactions.

References

[Badrinath 88] B.R. Badrinath and Krithi Ramamritham.
Synchronizing Transactions on Objects.
IEEE Transactions on Computers 37(5):541-547, May, 1988.

[Garcia-Molina 87]
Hector Garcia-Molina and Kenneth Salem.
SAGAS.
Technical Report CS-TR-070-87, Princeton University Department of Computer Science, January, 1987.

[Leblang 84] David B. Leblang and Robert P. Chase, Jr.
Computer-Aided Software Engineering in a Distributed Workstation Environment.
In *SIGSoft/SIGPlan Software Engineering Symposium on Practical Software Development Environments*, pages 104-112. Pittsburgh, April, 1984.
Special issue of *SIGPlan Notices*, 19(5), May 1984.

[Martin 88] Bruce E. Martin.
Concurrent Nested Objects Computations.
PhD thesis, University of California at San Diego, Department of Computer Science and Engineering., June, 1988.

[Moss 81] J. Eliot B. Moss.
Nested Transactions: An Approach to Reliable Distributed Computing.
PhD thesis, MIT, April, 1981.
MIT LCS TR-260.

[Pu 88] Calton Pu, Gail E. Kaiser and Norman Hutchinson.
Split-Transactions for Open-Ended Activities.
In *Fourteenth International Conference on Very Large Data Bases*, pages 26-37. Los Angeles CA, August, 1988.

[Reed 78]	David P. Reed. *Naming and Synchronization in a Decentralized Computer System.* PhD thesis, MIT, September, 1978. MIT LCS TR-205.
[Rochkind 75]	M. J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering* SE-1:364-370, 1975.
[Sun 88]	*Introduction to the NSE* Sun Microsystems, Inc., Mountain View CA, 1988.
[Tichy 85]	Walter F. Tichy. RCS — A System for Version Control. *Software — Practice and Experience* 15(7):637-654, July, 1985.
[Walpole 88]	J. Walpole, G.S. Blair, J. Malik and J.R. Nicol. Maintaining Consistency in Distributed Software Engineering Environments. In *8th International Conference on Distributed Computing Systems*, pages 418-425. San Jose CA, June, 1988.