Interfacing Cooperative Transactions to Software Development Environments

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

2 April 1990

Abstract

Many cooperative transaction models are intended to support teams of programmers sharing a software development environment. However, papers on this topic often neglect to describe how the programmers might actually interact with the cooperative transactions in terms of user commands and corresponding responses from the environment. In this paper, we consider a particular cooperative transaction model, called participant transactions, and explain the user model that needs to be implemented by a software development environment.

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

keywords: Concurrency control, cooperative work, extended transaction model, large scale system, multi-user software development environment.

1. Introduction

We have developed an extended transaction model, called *participant transactions*, that is a member of the family of concurrency control models intended to support cooperation among groups of transactions [Klahold 85]. Our model is intended to support cooperation among teams of programmers working together to develop or maintain a large scale software system, where these programmers employ a common software development environment (SDE) that provides software development tools and a shared database, and supports a particular software development process.

A previous paper [Kaiser 90a] presents our participant transaction model from the viewpoint of what schedules it allows and disallows, and sketches some approaches to enforcing the model. In this paper, we discuss the pragmatic issues of embedding the participant transaction model in a multi-user SDE. We are concerned here primarily with how the model might be presented to the programmers through the user interface of the environment and how this user view might be supported by the environment.

We start with some background on the requirements for cooperative transactions for software development and briefly survey related work. We then describe our participant transactions model, first giving the gist and then in more depth explaining the user view of the model. Finally, we consider the interfaces between a transaction manager implementing the model and the other components of the SDE. These interfaces are discussed in terms of general requirements and the issues raised by the requirements with respect to a range of possible SDE architectures, rather than any particular environment.

2. Background

Multi-user software development environments must support data consistency, concurrency control and crash recovery. All of these facilities are provided in some form by the classical *atomic transaction* [Eswaran 76] implemented in commercial database management systems. However, SDEs also must support

- **Long duration operations**, since the reads and updates take the form of editing sessions, compilations, debugging sessions, execution of test suites, etc., which sometimes may continue for hours;
- **Interactive, open-ended tasks**, composed of many long duration operations, to take the software system from one semantically consistent state to another while developing a new system or responding to a modification request to a released

system; and

- **Cooperation among multiple programmers**, since for large systems it typically requires the concerted activities of many people to reach a consistent state defined by, e.g., compiling and linking successfully and passing a regression test suite.

These capabilities are not supported by the classical atomic transaction model, and it is generally recognized that an *extended transaction model* is needed [Neuhold 89]

There has been much progress towards developing a suitable extended transaction model, both in the data management community and in the software engineering community [Rowe 89], as well as in other areas with similar requirements, such as CAD/CAM and office automation. Two fundamental approaches have evolved for handling long duration transactions: extending serializability using additional knowledge, e.g., altruistic locking [Salem 87]; and relaxing serializability, e.g., sagas [Garcia-Molina 87]. The first approach addresses the long duration problem for traditional data management applications, such as printing bank statements for all customers at the end of the month, where it is possible to statically analyze the operations of long transactions to determine their access patterns and then conditionally release locks on objects[1] that are known will not be accessed again later. The second approach considers both long duration and (implicitly) interactivity, for example when a travel agent makes multiple reservations for a customer, by requiring a capability for compensating for previous operations when a long task is aborted. But neither approach deals with cooperation among multiple users.

A partial solution to the cooperation problem can be found in approaches to coordination among multiple programmers; most of these approaches support some form of ‘‘checkout’’ scheme. In pessimistic checkout, e.g., design transactions [Katz 84], each programmer obtains a lock on all the objects of interest (e.g., source code files, documentation files) and copies them to a private workspace. Once the programmer’s task has been completed, he or she copies any updated objects back to the shared database and releases the locks. In some schemes, some of the objects may be checked in early, while work continues on other objects. While this increases concurrency, it does not guarantee either serializability or semantic consistency (e.g., error-free compilation and linking) among the objects since subsequent changes may be made by other

---

[1]We do not intend to imply any particular style of data model or database, and use terms like object and object management system generically to include any kind of data item and database, resp., including a view of the native file system controlled by the SDE. However, we discuss some requirements placed on the object management system in Section 4.3.

programmers and/or the original programmer may abort changes to the remaining objects even though there may be dependencies with respect to the already checked in objects. In some schemes, multiple programmers can work simultaneously on the same object by creating multiple versions of the object, also increasing concurrency, but they do not see each others' updates until later when it is necessary to merge the changes. Such coordination prevents programmers from accidently modifying the same object at the same time, so information and effort is neither lost nor duplicated, but it is left entirely to the programmers to regulate consistency.

Some of these problems are solved in one optimistic checkout scheme, the copy/modify/merge paradigm implemented in Sun's Network Software Environments (NSE) [Adams 89]. Multiple programmers may simultaneously check out the same objects, but the environment detects cases where another programmer has checked in the same object after the first programmer's checkout and requires that the changes to be merged; a text-based merge tool is provided. While this does not fully solve the merge problem, other research has provided hope that semantically correct merging can be automated and invalid merging can be prevented [Horwitz 89]. In NSE, a semantically related set of objects, called a component, is checked in as a unit; this does not by itself guarantee semantic consistency among the objects in the component, but makes it possible to require some form of consistency analysis prior to checkin. Our Infuse system, which supports pessimistic checkout, requires both static semantic analysis and integration testing before checkin is permitted [Kaiser 89a]. NSE, Infuse, and most other checkout schemes include some notion of hierarchy, in which groups of programmers integrate their changes in a shared workspace at one level of the hierarchy before checking in the combined changes to its parent level.

Some of these coordination mechanisms have been extended to support synergistic cooperation among multiple programmers. One requirement of synergistic cooperation is *notification*, so that cooperating programmers can be kept informed of what each other is doing. NSE provides a general notification mechanism, but at a rather coarse grain — basically at the level of checking a component in or out. The Apollo Domain Software Engineering Environment (DSEE) [Leblang 84] provides a similar notification scheme, but at the less coarse grain of dependencies among individual files. Our Mercury system permits the programmers to select either coarse or fine grain, where in the fine grained case programmers are automatically informed when changes to interfaces of other modules induce static semantic errors in their own modules as soon as the individual editing commands have been entered [Kaiser 87].

Another requirement of synergistic cooperation is to divide programmers, or at least the transactions they control, into *groups*. The transaction groups paradigm [Skarra 89] is one example of such a scheme, where serializability may be relaxed among transactions in the same group to allow concerted effort, but the transactions of different groups are isolated in the classical sense. Transaction groups can be implemented using the locking and communication (notification) modes of the Observer II object server [Fernandez 89]. Total lack of constraints among all members of a large team of, say, one hundred programmers would not be practical; instead, programmers would be divided into small groups who work closely, for example on the same subsystem, with consistency regulation among the groups [Perry 88]. Our participant transactions model is an instance of the groups approach to synergistic cooperation, and it is different from most other such models in that it allows overlap among groups, that is, one or more transactions may be members of multiple groups. This is useful for updates to libraries and other resources shared by multiple subsystems.

3. Participant Transactions Model

In our model, cooperation is supported among member transactions, called *participants*, of a group of transactions, called a *domain*. All other transactions not in the domain are called *observers* with respect to that domain. The participants in a domain are permitted to see the intermediate results of other participants and may share access to the same objects in a non-serializable fashion. However, observers are isolated from participants in the classical sense of serializability.

Although this is not precisely correct, one way to think of this is a domain is a top-level nested transaction and the participating transactions are its subtransactions [Moss 85]. The domains are serialized, but not the participant transactions within a domain. However, the primitive operations within a participant transaction are atomic, in the sense that one participant transaction cannot read an object at the same time another participant transaction is updating the object (nor should there be multiple updates at the same time). This atomicity could be relaxed in some cases, when concurrent operations are compatible according to their semantics [Herlihy 90], but we do not address this here; we do consider tools whose invocation does not map cleanly to a read or update, in Section 4.2.

There may be an arbitrary number of domains, and every transaction is in one or more domains; in some cases, the transaction may be the only member of its singleton domain. Although the participant transactions model allows a transaction to be a member of more than one domain (violating the nesting paradigm mentioned above), that aspect of the model is not relevant to the purpose of this paper, so in the sequel we discuss transactions as if each were required to be a member of exactly one domain.

3.1. Overview of Participant Transaction Schedules and Conflicts

	Alice	Bob
(1)	begin(D)	
(2)	update(M)	
(3)	update(N)	begin(D)
(4)		access(X)
(5)		read(N)
(6)	update(M)	update(N)
(7)	read(N)	
(8)	update(N)	
(9)		commit(N,X)
(10)	update(N)	
(11)	commit(M,N)	

Figure 3-1: Example Participation Schedule

Figure 3-1 shows a legal participant transactions schedule. Note that this schedule is not serializable in the classical sense, because Alice's transaction cannot be serialized before Bob's nor vice versa because of the conflicting accesses by the transactions to object `N`. But the semantics of participant transactions allows multiple transactions in the same domain (`D` in this case) to exchange intermediate results in a non-serializable manner, so this schedule is allowed.

Figure 3-2 gives an illegal participant transactions schedule. Charlie's transaction is not a member of the domain consisting of Alice's and Bob's transactions, and is thus an observer rather than a participant. There is a circular dependency among Alice's and Charlie's transactions, involving objects `O` and `N`, and thus no serialized schedule is possible.

	Alice	Bob	Charlie
(1)	begin(D)		begin
(2)	update(M)		update(O)
(3)	update(N)	begin(D)	
(4)	read(O)	access(X)	
(5)	update(O)	read(N)	
(6)	update(M)	update(N)	
(7)	read(N)		
(8)	update(N)		
(9)		commit(N,X)	
(10)			read(N)
(11)	update(N)		update(O)
(12)			commit(N,O)
(13)	update(N)		
(14)	commit(M,N,O)		

Figure 3-2: Example Participation Conflict

3.2. User View of Participant Transactions

The examples in the previous section deal solely with legal and illegal schedules. We first describe the user commands for controlling participant transactions, and then go through the details of the previous examples.

A new participant transaction is initiated by a user giving the `begin-transaction` command, with one argument, the name of the domain. Domains must be defined and allowable participants controlled in some manner; this will be discussed in Section 4.1. The command returns a unique transaction identifier `TID`. For now, we assume that the `TID` is the unique login identifier of the user and thus is a simple name and easy to remember, but in practice this would not work because then a user could have only one transaction at a time and no subtransactions; this issue is also explored in Section 4.1.

At any time after initiating the transaction, the user may issue the `commit-transaction` command to attempt to commit the transaction and make all its updates permanent, or the `abort-transaction` command to explicitly rollback any updates made by the transaction (this might be desirable, for example, when a programmer realizes his or her approach to fixing a bug was incorrect and it is easier to throw everything out and start over rather than to attempt repair). While the transaction is in progress, the user may issue an arbitrary sequence of commands that invoke tools such as editors, compilers, debuggers, etc. Due to the user control over the transactions and other properties of the participant transaction model, the semantics of the transaction operations do not map directly to their counterparts in the classical atomic transaction model. Further, tool invocations do not correspond directly to reads and updates, but we address this in Section 4.2 and for now consider only the step by step execution of both valid and invalid participant transaction schedules assuming only read and update operations on objects.

3.3. Step-by-Step Participation Schedule

	Alice	Bob
(1)	begin(D)	
(2)	update(M)	
(3)	update(N)	begin(D)
(4)		access(X)
(5)		read(N)
(6)	update(M)	update(N)
(7)	read(N)	
(8)	update(N)	
(9)		commit(N,X)
(10)	update(N)	
(11)	commit(M,N)	

Figure 3-3: Step-by-Step Participation Schedule

Figure 3-1 repeats the correct schedule of Section 3.1 consisting of two participant transactions controlled by Alice and Bob. Alice initiates a transaction in domain `D`, and then updates object `M` and object `N` (lines 1-3). Note that this means that the value of `N` may depend in some way on the value of `M`, since an update involves reading as well as writing. At about this time, Bob initiates another transaction in the same domain (line 3), and then goes on to access (either read or update, it does not matter) object `X` (line 4). The first potential conflict arises when Bob reads `N` (line 5), which has already been updated by Alice (line 3) but not committed. Because Bob is participating in the same domain as Alice, the read is allowed; it is important to stress that Bob reads the exact same object `N` that Alice updated, not some previous or branching version.

Then Bob goes on to update `N` (line 6); read followed by an update should be treated as a single update, but we separate the operations here for ease in exposition. Again, the update is allowed because Bob and Alice are in the same domain. Bob overwrites `N` in such a way that Alice's next read of `N` (line 7) will see Bob's update while Alice's next update (line 8) will modify the product of Bob's update. In practice, it might be useful to maintain a sequence of immutable revisions, but the important point is that Alice by default sees the new revision written by Bob as opposed to her previous revision, and modifies a copy of this new revision. Of course, Alice and Bob could explicitly request to access different versions if desired, but that is not the concern here.

Now Bob commits his transaction (line 9) and goes on to do something else, perhaps initiate a new transaction to carry out another software development task. However, Bob's ''commit'' of `N` does not immediately make this object public, neither the last update by Bob (line 6) nor the more recent update by Alice (line 8). Clearly the last update by Alice (line 8) cannot be committed by Bob. Let's say that instead the last update by Bob (line 6) were published but not the last update by Alice (line 8). But this value of `N` may depend on the uncommitted update to `M` (line 2) and almost certainly depends on the uncommitted update to `N` (line 3) — unless Bob's change was to remove Alice's modifications — so making Bob's update to `N` public at this point should not be allowed by the SDE. Our rationale is explained in the next section, when an observer Charlie is added to the example.

After Bob issues the commit operation, Alice makes another update to `N` (line 10) and then commits (line 11). Assuming there are no other members of the domain `D`, now `M`, `N` and `X` are all made public and observers can access them.

Let's consider the case where Alice aborts rather than commits (on line 11); the modified schedule is not shown. Since the last remaining transaction in the domain has terminated, it seems that any committed updates from the domain should now be made public. This would mean that the last update to `N` made by Bob (line 6), not either of the later updates made by Alice (lines 8 and 10) is the one to be published. But there is a problem, because as discussed above this value of `N` may depend on Alice's rolled back updates to `M` (line 2) and `N` (line 3). This problem could be resolved in some sense by a cascaded rollback, automatically aborting Bob's transaction, but automatic abort is never acceptable for long interactive transactions.

Instead, the solution is for the SDE to automatically initiate a special new transaction, perhaps called Error, participating in the same domain D. Both Alice and Bob are notified (for example, by electronic mail) of the situation, and either (but not both) can take over control of the Error transaction. This user then decides what to do, perhaps in negotiation with the other user. He or she could immediately commit the Error transaction, if the user knows that N is not semantically dependent on the rolled back update of M and treats Alice's changes to N as if they had been made by Bob; this really terminates the last transaction in the domain, and Bob's last update to N (line 6) is now really committed in the conventional sense. Or the user could do anything else, including updating M, N or some other arbitrary object (provided this does not itself introduce a new conflict with respect to observers) in order to restore semantic consistency, and commit later on — terminating the domain. However, if the Error transaction is aborted, then the SDE must initiate a new Error transaction, repeatedly, until finally one is committed and then the domain terminates.

3.4. Step-by-Step Participation Conflict

	Alice	Bob	Charlie
(1)	begin(D)		begin
(2)	update(M)		update(O)
(3)	update(N)	begin(D)	
(4)		access(X)	
(5)	update(O)	read(N)	
(6)	update(M)	update(N)	
(7)	read(N)		
(8)	update(N)		
(9)		commit(N,X)	
(10)			read(N)
(11)	update(N)		update(O)
(12)			commit(N,O)
(13)	update(N)		
(14)	commit(M,N,O)		

Figure 3-4: Step-by-Step Participation Conflict

We now turn to the illegal schedule of Section 3.1. Alice and Bob do essentially the same things as above, except now Alice also reads (line 4) and updates (line 5) object O, which has already been updated by Charlie (line 2). Recall that Charlie is an observer of the domain shared by Alice's and Bob's transactions. If Alice reads the new version of O created by Charlie, this could lead to inconsistency if Charlie later aborts his transaction. Since the only repair — cascaded aborts — is unacceptable, the SDE would instead force Alice's transaction to access the appropriate previous version of `O`; this might or might not be transparent to the users. For example, depending on the policies embedded in the SDE, it might or might not notify Alice that `O` was being updated in another domain, and it might or might not notify Charlie that a transaction from another domain had accessed an old version of `O`. But in any case the SDE would have to remember at this point that the domain `D` consisting of Alice's and Bob's transaction's must be serialized before Charlie's (anonymous) domain. Note that this approach would not be possible if the SDE did not support multiple versions of objects, in which case the environment would have had to disallow Alice's access to `O` and she would have to do something else; since almost all SDEs do incorporate some kind of version control system, we assume the availability of multiple versions throughout the rest of this section. Versions are discussed further in Section 4.

Another conflict arises when Charlie later attempts to read object `N` (line 10). Access to to the new version created by Alice (line 8) should not be allowed, because of the potential cascaded abort problem mentioned above. Access to an old version of `N` prior to Bob's update (line 6) cannot be allowed, because Bob has already ''committed'' his work, in the sense of declaring that he is done with his task. The only possibility is access to the version created by Bob's transaction. We said above that Bob's update is not yet publicly available, since his domain `D` has not terminated, but let's ignore that for now to motivate *why* Bob's version should not be published at the time of his commit (line 9).

Now Charlie goes on to update object `O` (line 11), perhaps based on information obtained by reading `N`, and commit (line 12). The resulting schedule would be serializable only if Alice's transaction later aborts, and the consequent Error transaction permits the entire domain `D` to still be serialized before Charlie's transaction. But instead, Alice goes on to update `N` two more times (lines 11 and 13) and then commit (line 14), terminating the domain. The schedule is non-serializable, since Charlie's transaction must be serialized after Alice's because of `O`, but before Alice's because of `N`.

One solution to this dilemma would be for the SDE to prevent Alice from making her last two updates to `N`, in which case it may seem she could go ahead and commit her other work. But there would still be a problem regarding Alice's previous update (line 8), which had happened before the difficulty regarding conflict with Charlie's transaction over `O` was introduced but after the previous conflict over `N` had been resolved by allowing Charlie to see Bob's update. Perhaps even Alice's previous update (line 8) should not have been allowed; the implication would be that no transaction in a domain could update an object after one transaction in the domain that had already updated the same object had committed (at least in the case that some observer actually views the object), essentially applying a two-phase locking protocol to the domain as a whole. A better solution would be to hold back on publishing the update until all transactions in the domain have terminated, effectively applying strict two-phase locking, and thus disallow Charlie's access to Bob's update of `N` in the first place, and this is in fact the course we follow.

Thus, when Charlie attempts to read `N` (line 10), the SDE would inform Charlie that this is not valid, and he could go on to do something else but would not be allowed to read `N`. Some SDEs might also inform Alice (or even Bob), making it possible for the users to negotiate among themselves as to how to best resolve the conflict. In another paper [Pu 88], we describe two novel transaction operations, `split-transaction` and `join-transaction`, that aid such negotiation by permitting restructuring of in-progress transactions by branching one transaction into two or more as if they had always been independent or by joining two or more transactions as if they had always been combined.[2] For the purposes of this paper, however, we assume that the only options are for Charlie's access to be disallowed; for Alice and Bob to agree to both commit immediately, so that Charlie can access their new version of object `N`; or for Alice and Bob to both abort, so Charlie can access the previous version of `N`. In any case, both Bob and Alice have to do the same thing, since they have accessed each others' intermediate results, not just because they happen to be members of the same domain.

In the above example, we have described a pessimistic approach to enforcing the participant transactions model. This seems like the best approach for software development, but it is not the only possibility. Pure optimistic concurrency control [Kung 81] would require automatic rollback when transactions could not be validated, and not many programmers would use an SDE that threw out hours or perhaps weeks of effort! But the optimistic policy implemented by NSE requires merging rather than rollback when there are conflicts, and this seems acceptable if users were always notified of the future merging as soon as its potential could be detected.

---

[2]That paper does not address any grouping of transactions as discussed here; instead, the two split transactions become two top-level transactions and the two joined transactions are merged into one.

4. Interfaces to Software Development Environments

We left several open questions in the previous section, including how exactly are domains and transaction identifiers manipulated by end-user commands, and how is the transaction manager integrated with the SDE kernel. An SDE implementing participant transactions needs to provide some end-user model for how domains are defined, created and destroyed, and how it is determined whether any given transaction can be initiated in the domain. It is also necessary to define the `TID`s returned by `begin-transaction` operations, including their allocation, deallocation and storage for later reference. Such later reference comes up in the context of additional transaction operations such as `suspend-transaction` and `resume-transaction` commands, as introduced in Section 4.1. A special form of `resume-transaction`, which we might call *attachment* to a transaction, was implied in the previous section (i.e., a user taking control of the automatically initiated Error transaction). There is also the issue of how the tools invoked directly through SDE commands and indirectly through SDE automatic behaviors correspond to the reads and updates of the participant transaction model. But most significantly, the SDE kernel also has to provide mechanisms for enforcing the participant transactions model, and it must be at least compatible with primitives supplied by the object management system.

All of these issues must be resolved in any implementation of participant transactions in an SDE; analogous issues would have to be resolved for any other cooperative transaction model. We do not attempt to give answers here, but instead informally discuss the requirements placed on SDE implementations by cooperative transactions. We will try to distinguish the needs of our particular model from other potential models in the following discussion.

We have divided the interaction between a cooperative transaction model and the host SDE into the effects on three interfaces: control, SDE kernel and object management system (OMS), as shown in Figure 4-1. In some SDEs, and some cooperative transaction models, certain of these interfaces might be combined, while in others they might be divided into multiple levels. Most notably, in environment kernels that may be tailored by an administrator to present different behavior for different projects, it is necessary to separate the control interface(s) as viewed by the end-user and as viewed by the administrator (as depicted in the figure) if the behavior definition affects the transaction-oriented commands. It also seems likely that the OMS would be divided between a front end object manager and a back end object server, but we do not

Figure 4-1: Proposed Interfaces for SDEs address this here.

Any practical SDE must provide a control interface to a version control system in order to select particular versions of objects, an SDE kernel interface to manage the versions, and an OMS interface to store the versions, preferably in compressed form. This issue seems orthogonal to cooperative transactions, except that some models may assume multiple versions, while others may not; our participant transaction models would work basically the same way whether or not versions are allowed, but some schedules that are legal with multiple versions are illegal without them (as mentioned in Section 3.4.

4.1. Control Interface

The control interface defines the commands available to the SDE end-users, including the arguments and options for the commands, and the error messages returned to the user when there are conflicts. The presentation of a cooperative transaction model to the end-user of SDEs is crucial, but has not usually been addressed in papers concerned with the classical atomic transaction model, since there the end-user simply submits a transaction and it is eventually executed in a serializable manner.[3] The error messages returned usually have nothing to do with concurrency control, for example, "insufficient funds".[4] It is in general irrelevant for our purposes whether the user interface is graphical, with windows, menus and a pointing device, or consists of command lines and prompts on a dumb terminal, but some exceptions are noted below.

The control interface must provide some means for defining domains, preferably each with a name intelligible to end-users and that reflects the common task of members of the domain. In most professional settings, there must be access control to prevent modification of domain definitions by unauthorized personnel, since managers will typically require control over who may or may not share unreleased work. There are problems with end-user (manager) transactions that themselves define such domains, particularly those that dynamically add/remove participant transactions to/from domains when the domain is active (i.e., there is at

---

[3] One notable exception is Pausch's dissertation [Pausch 88].

[4] It is true that such error messages may appear to involve the serial ordering of operations in a way that may be confusing to the end-user, for example when he or she has just deposited a large check that will not be cleared for ten days, but users of ATMs have learned to accept such rules of the trade.

least one participant whose transaction has been initiated but not terminated). To support realistic software development scenarios, it must be possible to remove participants corresponding users who have been transferred among departments or have quit their jobs, even though their intermediate results have been seen by other participants. Perhaps the best way to do this is suspend the transaction as described below, change the access rights of this transaction, and allow later attachment (or resumption) by another user. It should not be possible for a user to retain rights to a transaction, or the objects it has accessed, after such removal by management; this is one of the jobs for the SDE kernel and/or OMS, as elaborated in the next sections. None of these problems are peculiar to the domains of participant transactions, and seem to apply equally to other cooperative transaction schemes based on groups; the groups must be declared and manipulated somehow.

User-controlled transactions may continue for hours, days or weeks. It is unrealistic to expect users to stay logged in and attached to the same transaction this entire time, or for their workstations, file servers, network, etc. to functioning correctly throughout the period. We do not address crash recovery in this paper, but we do need to be concerned with how users can logout and go home or switch between multiple tasks. Thus, some sort of `suspend-transaction` and `resume-transaction` commands should be provided. There must be a way to persistently store suspended transactions while they are inactive; the corresponding domains must also persist. This is discussed further in Section 4.3.

Further, there must be some way for someone other than the original user to resume a transaction, say when the original users is on vacation. It may or may not be necessary to provide a distinct `attach-transaction` command, but there must be some way to determine whether or not a given user is authorized to resume a particular transaction. When either the original user or substitute does resume a transaction, it must be easy to query the transaction itself in some way to remind or determine what is the current status and where to go from here. DSEE’s task list seems a good approach, but this might not be appropriate for all SDEs. In any case, this does not help with savepoints at a finer granularity than a task.

The notion of resuming transactions leads to the issue of naming transactions, i.e., creating `TID`s. Associating a transaction with a login userid, provided there is some way of transferring the association to another user, seems plausible if a user may have only one ongoing transaction at a time. But this does not permit either subtransactions or multiple top-level (for the user) tasks. It is probably best to name transactions, as we have assumed domains are named, but there must be some sort of catalog that users can view to list the names of their transactions, including some intelligible organization of subtransactions, as well as accessible domains. None of these issues are peculiar to participant transactions. Any cooperative transaction mechanism must have means for suspending and resuming long duration transactions, including naming the desired transaction to resume.

The control interface must support the separate roles of the SDE administrator and the SDE end-user. All the commands discussed above are in the realm of the end-user, whether that end-user is a programmer or a manager, with different access rights. There is no particular need for an SDE administrator in many environments, since the specific environment is employed ‘‘as is’’. But there has been much interest for many years on tailorable environments, ranging from systems that generate syntax-directed editors from grammars (e.g., Synthesizer Generator [Reps 89], Gandalf [Habermann 86]) to environments where the particular software process model is programmable (e.g., Arcadia [Taylor 88], Marvel [Kaiser 88]).

When such an environment is used, it may be possible for the administrator who tailors the environment to specify particular allowable and unallowable interleavings among participant transactions — in these cases, individual read and update operations would not necessarily determine the granularity of access to intermediate results and a larger granularity might be defined for a particular software process or project. There might be particular policies governing interactions between users in different departments, responsibility for responding to modification requests, or requirements for approval for all changes from an independent quality assurance organization prior to top-level commitment.

There must be commands, or more likely notation in some specification language that acts as input to the SDE kernel, along with any data model and/or process model, for defining these interleavings. It may even be possible to declare different classes of domains, or groups, with different concurrency control policies. We do not elaborate further here, but point at one particularly interesting approach to specifying a range of cooperative transaction schemes by Chrysanthis and Ramamritham [Chrysanthis 90].

4.2. SDE Kernel Interface

The SDE itself must validate or invalidate all transaction-oriented commands in the context of the software development process enforced by the environment [Tully 88, Perry 89]. The environment should understand the semantics of the tools it invokes, and how they correspond to the read and update operations it transmits to the OMS. Many environments also have sufficient knowledge to validate apparent read-update or update-update conflicts, due to understanding of the specific tools involved, for example to be aware of consistency-preserving commutativity among tools [Weihl 88].

In SDEs, there are relatively few simple reads and updates of objects. One notable case where there may be relatively simple reads is in a graphical browser, where short descriptions of objects (e.g., their names and types) and indications of their relations (e.g., component-of) are displayed all at once on a screen for a segment of the full database. In this case, it may be desirable for the SDE kernel to batch operations to the OMS, to simultaneously acquire access to a hierarchy or network of objects; alternatively, the objects might be obtained on demand as the segment is explored.

In most other cases, operations on objects are complex and do not corresponding directly to reads and updates. An editing session may involve reading many file objects and updating some of them. Editing is not usually two-phase: the user decides to read additional objects as he or she goes along, and may decide to save updates to individual objects at any point. Although debugging is usually read-only, unless the execution traces themselves are considered part of the SDE's database, but again the user decides what to look at while debugging is in progress.

One concern is the principle of least astonishment: when a user reads an object into an editing buffer or into a debugger space, and them an hour later reads the same object without making any intermediate updates, he or she expects to find the same version of the objects without any modifications by other users. This can be problematical in the context of participant domains, since serializability is relaxed such that this cannot be guaranteed.

Here is where some kind of notification scheme seems mandatory, to make sure users know about changes made by other users to objects of interest, and have the opportunity to request to continue using older versions (or perhaps this should be the default). This is a significant hole in our participant transactions model: Notification is not addressed at all, and reads and updates are treated as atomic short-term operations. Most other cooperative transaction models have the same flaw, although DSEE and Mercury do provide advance warning of updates by other users. NSE and Observer II provide notifications on initial checkout (read), and later on final checkin (update), but the damage may have been done in between and it is left to the end-user to poll objects of interest to keep track of what is happening to them. Fine grain notification must be better integrated into cooperative transaction models and supported by the SDE kernel.

Now let's consider authorization and security. Either the SDE kernel or the OMS must enforce access control with respect to domain descriptions and control over participant transactions. It would be easy to place this in the OMS, on the basis of access control lists or capabilities, but this could not take into account any restrictions or decision algorithms based on the software process model. If different phases of the lifecycle or other patterns based on the history of the software project can determine who should be able to do what, this responsibility must belong to the SDE. This means the OMS must make sure it is not possible to circumvent the SDE, for example, by direct access through the native file system. Perhaps it is best for the SDE itself, acting as a privileged superuser, to ''own'' the database and for only the SDE to have the power of controlling object access.

Finally, we should explicitly consider the issue of automatic behavior of the environment, for the purposes of bookkeeping to maintain an audit trail and/or implicit invocation of tools, for example, to automatically carry out portions of the software process model. The environment might be doing things without the knowledge of any user, but doing these things on the behalf of a particular user, on the behalf of all users collectively, or on its own behalf to maintain internal records. It is unclear how this interacts with any cooperative transaction model. Is this automatic behavior part of a user transaction or not? If it is treated as part of a user transaction, then we might have the problem of spurious error messages delivered to the user regarding object accesses he or she did not know were being made. This seems inappropriate, unless the messages can be filtered by the SDE kernel back to reflect the user commands that initiated the actions in some user-intelligible manner.

This problem is too complex to fully address here, and in any case we do not yet fully understand the interactions between cooperative transaction models and the behaviors of active environments. We have briefly considered such interaction in another paper [Kaiser 89b], specifically treating the `split-transaction` and `join-transaction` transaction operations mentioned earlier in the context of the Marvel rule-based development environment[5] [Kaiser 90b]. Marvel assists in the software development process by opportunistically forward and backward chaining on rules defining the software development activities supported by the environment.

4.3. Object Management System Interface

The OMS supports the SDE kernel in its mission. The OMS may be a relatively dumb system that is only aware of primitive accesses to objects or may be a relatively sophisticated system with soft locks of various sorts and capabilities for notifying different SDE sessions of accesses by concurrent SDE sessions. The full spectrum of OMS systems must avoid pre-empting the concurrency control policies of the SDE, whether built-in or programmed by an administrator.

An SDE may employ an OMS architecture ranging from a controlled view of the native file system to a full blown database management system. Thus the data model might be implicit (in the case of the file system), relational, object-oriented or some combination, e.g., many SDEs store conventional software objects in files and directories, with relationships between the objects stored in some kind of database. Bernstein [Bernstein 87] has explained some of the problems with applying traditional database technology to SDEs, while Garza and Kim [Garza 88] discuss some of the complications imposed by object-oriented data modeling on transaction management (for the classical transaction model). We do not address these issues further here.

The OMS must support persistent storage of suspended transactions and active domains, as discussed in the previous section. In a short position paper [Kaiser 89c], we have sketched one approach to this problem. Each ongoing and committed transaction is represented as a *transaction object*, an instance of the built-in "Transaction" class, and its readset, writeset, locks held and other relevant information are represented as the attributes of the object.[6] Ongoing transactions might be saved for fault tolerance purposes as well as suspend/resume, while saving committed transactions might be useful for keeping track of which versions of objects are consistent with each other — something that traditional version control tools such as RCS [Tichy

[5]The single-user Marvel version 2.5 is available for licensing to educational institutions and industrial sponsors; contact Mike Sokolsky, sokolsky@cs.columbia.edu, 212-854-8348 for information.

[6]Note this is different from the transaction classes of Avalon/C++ [Detlefs 88], which are essentially mixins providing synchronization primitives for use in implementing atomic objects.

85] are notoriously weak on. Nice solutions have been developed in NSE and in the Cosmos domain relative addressing model [Walpole 88] — from which we adapted our use of the term "domain". Domains might similarly be represented as instances of a "Domain" class. This approach seems rather elegant, but opens some difficult questions regarding what kinds of operations might be applied to these objects (in addition to `suspend-transaction` and `resume-transaction`) and what it might mean for a transaction to access a transaction object or domain object.

This returns us to the issue of transaction naming, previously considered in Section 4.1 regarding how names are chosen. Here we are concerned with persistent storage of the transaction name catalogs. If transaction objects are employed, the catalog might be subsumed into the normal name service or query interface used to access other persistent objects. The OMS must guarantee some level of safety, to avoid allocating the same `TID` to multiple transactions and to prevent use of a deallocated `TID`; the requirements here do not seem particularly different than for the transaction manager of a conventional database management system.

1. Check compatibility of operation with lock mode
2. If lock mode compatible, access
    - Augment lock
3. If lock mode not compatible, check compatibility of transaction with domains of transaction(s) holding lock(s)
4. If lock domain compatible, access
    - Augment lock
5. If lock domain not compatible
    - Pessimistic (single version) — no access
    - Optimistic (multiple versions) — notify and branch version

Figure 4-2: One Protocol for Implementing Participant Transactions

Figure 4-2 sketches a pessimistic protocol that might be used by a transaction manager for enforcing participant transactions, assuming a one to one correspondence between operations and objects (which is not sufficient as explained in the previous section). The protocol applies only to read, update and similar operations specific to SDEs (perhaps edit, append to change log, etc.), and does not cover operations for initiating, committing and aborting transactions; these protocols were explained informally in Section 3.

This protocol might be implemented by either the OMS or the SDE kernel, depending on the division of labor. Or perhaps a separate transaction layer would be placed between the SDE kernel and OMS. But wherever the responsibility lies, it must be possible to check compatibility between operations (easy enough for reads and updates). It must also be possible to associate multiple `TIDs`, however defined, with a lock; this is what is meant by ‘‘augmenting’’ the lock. This kind of facility is already supported in traditional database management systems, since multiple transactions may simultaneously hold read locks; all cooperative transactions add is the potential for multiple write locks (and perhaps various other kinds of soft locks).

However, the participant transaction model requires locks to be held until all transactions in a domain have terminated, so lock management must deal with domains as well as transactions. Further, there must be a way of detecting whether or not a given transaction is in the domain holding the lock. Other cooperative transaction models might or might not require this same capability for their groups, depending on the semantics. When the lock mode and domain (group) are both incompatible according to the cooperative transaction model, either a pessimistic or optimistic approach might be employed. We do not mean pessimistic versus optimistic concurrency control. Instead, the scheme is being pessimistic or optimistic about the problems of later merging if conflicts automatically lead to divergent (branching) versions. Merging requires simultaneous access by a single transaction to multiple versions of the same object, and this must also be supported by the SDE kernel and/or the OMS.

5. Conclusions

We have briefly surveyed a range of approaches to extended transaction models culminating in groups of cooperative transactions, described one such model — participant transactions — in terms of the user view supplied by the software development environment, and discussed the requirements placed by cooperative transaction models on the environment in terms of end-user and administrator control interfaces, environment kernel interface and object management system interface. The primary goal of this paper is to open discussion on the requirements of these interfaces and how these requirements can be best fulfilled.

Acknowledgments

This paper reflects insights gained during discussions with a number of people, some of whom the author has unforgivably forgotten, but she would like to mention Nasser Barghouti, Mark Dowson, Dan Duchamp, Scott Hudson, Eliot Moss, Dewayne Perry, Calton Pu, Ken Salem, Soumitra Sengupta, Mike Sokolsky and Stan Zdonik.

References

[Adams 89] Evan W. Adams, Masahiro Honda and Terrence C. Miller.
Object Management in a CASE Environment.
In *11th International Conference on Software Engineering*, pages 154-163. Pittsburgh PA, May, 1989.

[Bernstein 87] Philip A. Bernstein.
Database System Support for Software Engineering.
In *9th International Conference on Software Engineering*, pages 166-178. Monterey CA, March, 1987.

[Chrysanthis 90] P.K. Chrysanthis and K. Ramamritham.
ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior.
In *ACM SIGMOD International Conference on the Management of Data*. May, 1990.
In press.

[Detlefs 88] David Detlefs, Maurice Herlihy and Jeannette Wing.
Inheritance of Synchronization and Recovery Properties in Avalon/C++.
Computer 21(12):57-69, December, 1988.

[Eswaran 76] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-632, November, 1976.

[Fernandez 89] Mary F. Fernandez and Stanley B. Zdonik.
Transaction Groups: A Model for Controlling Cooperative Work.
In *3rd International Workshop on Persistent Object Systems: Their Design, Implementation and Use*. Queensland, Australia, January, 1989.

[Garcia-Molina 87]
Hector Garcia-Molina and Kenneth Salem.
SAGAS.
In Umeshwar Dayal and Irv Traiger (editor), *ACM SIGMOD 1987 Annual Conference*, pages 249-259. San Francisco CA, May, 1987.
Special issue of *SIGMOD Record*, 16(3), December 1987.

[Garza 88] Jorge F. Garza and Won Kim.
Transaction Management in an Object-Oriented Database System.
In *SIGMOD International Conference on Data Management*, pages 37-45. Chicago IL, June, 1988.
Special issue of *SIGMOD Record*, 17(3), September 1988.

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

[Herlihy 90] Maurice Herlihy.
A Methodology for Implementing High Concurrent Data Structures.
In *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197-206. Seattle WA, March, 1990.

[Horwitz 89] Susan Horwitz, Jan Prins and Thomas Reps.
Integrating Noninterfering Versions of Programs.
ACM Transactions on Programming Languages and Systems 11(3):345-387, July, 1989.

[Kaiser 87] Gail E. Kaiser, Simon M. Kaplan and Josephine Micallef.
Multiuser, Distributed Language-Based Environments.
IEEE Software :58-67, November, 1987.

[Kaiser 88] Gail E. Kaiser, Peter H. Feiler and Steven S. Popovich.
Intelligent Assistance for Software Development and Maintenance.
IEEE Software 5(3):40-49, May, 1988.

[Kaiser 89a] 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.

[Kaiser 89b] Gail E. Kaiser.
A Marvelous Extended Transaction Processing Model.
In Gerhard Ritter (editor), *11th World Computer Conference IFIP Congress ’89*, pages 707-712. Elsevier Science Publishers B.V., San Francisco CA, August, 1989.

[Kaiser 89c] Gail E. Kaiser.
Modeling Configurations as Transactions.
In *2nd International Workshop on Software Configuration Management*, pages 129-132. October, 1989.
Special issue of *Software Engineering Notes*, 17(7), November 1989.

[Kaiser 90a] 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.

[Kaiser 90b] Gail E. Kaiser, Naser S. Barghouti and Michael H. Sokolsky.
Experience with Process Modeling in the Marvel Software Development Environment Kernel.
In *23rd Annual Hawaii International Conference on System Sciences*, pages 131-140. Kona HI, January, 1990.

[Katz 84] Randy H. Katz and Shlomo Weiss.
Design Transaction Management.
In *ACM IEEE 21st Design Automation Conference*, pages 692-693. IEEE Computer Society, Albuquerque NM, June, 1984.

[Klahold 85] P. Klahold, G. Schlageter, R. Unland and W. Wilkes.
A Transaction Model Supporting Complex Applications in Integrated Information Systems.
In *ACM-SIGMOD 1985 International Conference on Management of Data*, pages 388-401. ACM, Austin TX, May, 1985.

[Kung 81] H. T. Kung and John Robinson.
On Optimistic Methods for Concurrency Control.
ACM Transactions on Database Systems 6(2):213-226, June, 1981.

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

[Moss 85] J. Eliot B. Moss.
Information Systems: *Nested Transactions: An Approach to Reliable Distributed Computing.*
The MIT Press, Cambridge MA, 1985.
PhD Thesis, MIT LCS TR-260, April 1981.

[Neuhold 89] Erich Neuhold and Michael Stonebraker (editors).
Future Directions in DBMS Research.
SIGMOD Record 18(1), March, 1989.

[Pausch 88] Randy Pausch.
Adding Input and Output to the Transactional Model.
PhD thesis, Carnegie Mellon University, August, 1988.
CMU-CS-88-171.

[Perry 88] Dewayne E. Perry and Gail E. Kaiser.
Models of Software Development Environments.
In *10th International Conference on Software Engineering*, pages 60-68. Raffles City, Singapore, April, 1988.

[Perry 89] Dewayne Perry (editor).
5th International Software Process Workshop.
ACM Press, Kennebunkport ME, 1989.
In press.

[Pu 88] Calton Pu, Gail E. Kaiser and Norman Hutchinson.
Split-Transactions for Open-Ended Activities.
In Francois Bancilhon and David J. Dewitt (editors), *14th International Conference on Very Large Data Bases*, pages 26-37. Los Angeles CA, August, 1988.

[Reps 89] Thomas W. Reps and Tim Teitelbaum.
Texts and Monographs in Computer Science: *The Synthesizer Generator A System for Constructing Language-Based Editors.*
Springer-Verlag, New York, 1989.

[Rowe 89] Lawrence A. Rowe and Sharon Wensel (editors).
1989 ACM SIGMOD Workshop on Software CAD Databases.
February, 1989.

[Salem 87] Kenneth Salem, Hector Garcia-Molina and Rafael Alonso.
Altruistic Locking: A Strategy for Coping with Long Lived Transactions.
In *2nd International Workshop on High Performance Transaction Systems*. Pacific Grove CA, September, 1987.

[Skarra 89] 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.

[Taylor 88] Richard N. Taylor, Richard W. Selby, Michael Young, Frank C. Belz, Lori A. Clarke, Jack C. Wileden, Leon Osterweil and Alex L. Wolf.
Foundations for the Arcadia Environment Architecture.
In Peter Henderson (editor), *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1-13. ACM Press, Boston MA, November, 1988.
Special issue of *SIGPLAN Notices*, 24(2), February 1989 and of *Software Engineering Notes*, 13(5), November 1988.

[Tichy 85] Walter F. Tichy.
RCS — A System for Version Control.
Software — Practice and Experience 15(7):637-654, July, 1985.

[Tully 88] Colin Tully (editor).
4th International Software Process Workshop: Representing and Enacting the Software Process.
ACM Press, Moretonhampstead Devon, UK, 1988.
Special issue of *Software Engineering Notes*, 14(4), June 1989.

[Walpole 88] J. Walpole, G.S. Blair, J. Malik and J.R. Nicol.
A Unifying Model for Consistent Distributed Software Development Environments.
In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 183-190. Boston MA, November, 1988.
Special issue of *SIGPLAN Notices*, 24(2), February 1989 and of *Software Engineering Notes*, 13(5), November 1988.

[Weihl 88] William E. Weihl.
Commutativity-Based Concurrency Control for Abstract Data Types (Preliminary Report).
In Bruce D. Shriver (editor), *21st Annual Hawaii International Conference on System Sciences*, pages 205-214. IEEE Computer Society, Kona, HI, January, 1988.