CASE tool support for co-operative work in information systems design

Steven Kelly
Dept. of Computer Science and Information Systems,
University of Jyväskylä, PL 35, FIN–40351 Jyväskylä, Finland
Tel: +358 14 603036, Fax: 603011, Email: kelly@cs.jyu.fi

Abstract

The need for asynchronous co-operative work in design is shown by many surveys of CASE tool use and research on design. MetaEdit+ is a metaCASE environment that allows multiple simultaneous modellers to work together on designing information systems and also information systems design methods. We describe the automatic locking strategies that enable MetaEdit+ to offer a high level of concurrency whilst guaranteeing consistency, enabling effective co-operative work. In particular we describe a new collection data structure that allows high concurrency of updates even at small sizes, fulfilling the need in CASE for largest growth of design information at the start of a project. Finally, we evaluate MetaEdit+'s collaboration support and performance as teamware, comparing it with several CASE tools.

Keywords

Co-operative work, multi-user CASE, teamware

1 INTRODUCTION

MetaCASE has long been suggested as the answer to many of the current problems of CASE tools, in particular their inflexibility. Whilst a normal CASE tool supports only a single fixed method for designing information systems, a metaCASE tool can be configured by the user to support different methods. This allows organisations to support those methods they already have in use, and modify them as necessary to address their changing needs: vital in today's world where technology and methods change so quickly. Recently commercial and academic metaCASE tools have started to appear, with a few tools serious enough to take into industrial use. However, none of these tools support metaCASE in a multi-user environment, and most are single user even for CASE work. This is particularly odd, as metaCASE is really only useful in a multi-user environment: developing and using a custom method is not normally cost effective for a single user.

Even standard CASE tools have been slow to move from single to multi-user support. Empirical research has shown the current lack of multi-user support in CASE tools is a serious problem (Sto93, Rup95). In particular, Selamat et al. (Sel94) found that lack of multi-user support was the single largest CASE-specific reason why CASE tools were not being adopted in Malaysia. In addition to these questionnaire-based surveys of organisations, an empirical laboratory examination by Vessey & Sravanapudi (Ves95) found that support for co-operative working was poor in current CASE tools.

Another commonly identified source of discontent with CASE tools is the lack of integration between methods within a tool (Sto93, Rup95). This can often be explained by the fact that many tools use a simple file-based system for storage, rather than a true repository: each model is stored in its own file, and there is no linking between files, thus nor between models of different methods.

MetaEdit+ is a multi-user repository-based metaCASE environment which was produced by the MetaPHOR research project, and has been commercialised and available as a full product since November 1996. Other papers have examined its metaCASE features (Kel96) and innovative CASE features (Kel97); in this paper we concentrate on its multi-user functionality.

In the next section, we will examine related research and the requirements for multi-user CASE, then give a brief overview of MetaEdit+ and its ArtBASE database. Following that we will describe and evaluate the co-operative support MetaEdit+ offers through its automatic locking strategies and novel data structures. The success of these solutions in making MetaEdit+ a multi-user metaCASE tool is then evaluated briefly according to the criteria of Vessey and Sravanapudi (Ves95). Finally we conclude and examine some directions for future research.

2 BACKGROUND

2.1 Related research

To our knowledge there has been no recent empirical research on how designers use CASE tools co-operatively in practice, although among others Kraut and Streeter (Kra95) examine designers' communication in a wider organisational setting.

There are a large number of single or simple multi-user CASE tools. However, attempts to make more sophisticated CASE tools and metaCASE tools multi-user seem to founder: in particular many of the few manufacturers who have announced such multi-user metaCASE tools have later withdrawn them (Paradigm+, Excelerator/Customizer, ASTI Graphical Designer / MethodBuilder). Similarly, it appears to be easy to make a simple single user metaCASE research prototype: no end of examples can be found, but these are almost never developed into full functionality multi-user metaCASE environments.

Apart from MetaPHOR, two other significant research projects have worked on extending CASE and metaCASE technology with possibilities for computer-supported collaborative work (CSCW), and produced something more than a prototype.

Lincoln (previously IPSYS) ToolBuilder (initially described in (Ald91)) has a multi-user repository solution for CASE with the possibility to configure the methods supported. Methods are configured by using three different proprietary textual languages. Whilst this can be considered as metaCASE, the time taken to specify a new method is an order of magnitude larger than in MetaEdit+, and made more difficult by the separation of the metaCASE and CASE components into different tools, making testing and correcting a lengthy process of exporting, compiling, and linking the new metamodel to the CASE tool each time. This involves all users exiting, upgrading their CASE tool, restarting it and logging in to the repository again.

The second research project, at Nokia corporation, has produced TDE (Tai97), a CSCW design environment that includes some CASE functionality and the possibility of changing the methods supported. It allows near instantaneous updates of design data between multiple users using a special notification server they have developed, which runs in addition to the underlying ObjectStore database: the normal transaction protocol in ObjectStore is far too slow to use for such fast updates of models between users. Whilst the CSCW implementation is strong, the CASE functionality is limited to that of flow-charter type tools: any kind of relationship can be drawn between any kinds of objects, and there is no code generation. Methods can be changed from outside TDE using a textual language, but this includes only the basic ERA concepts and thus represents at best significantly limited metaCASE functionality. Whilst TDE is undoubtedly useful within a single corporation with a single method, and represents an important move away from the many CSCW drawing tools towards synchronous CSCW CASE, that move cannot be said to have been completed yet.

In addition to these, and interesting to us because of their similar development environment, a simple fixed-method workflow CASE tool was developed in Smalltalk in (Bec94). Their initial implementation used an RDBMS, but this was found to be very difficult to work with, requiring duplication of many data structures: one version for actual use and another relationalised version for the database, leading to large amounts of code just for these and maintaining consistency between them. They found the move to a product similar to ArtBASE very easy, and it solved the problems stated and provided many benefits.

2.2 Requirements for multi-user (meta-)CASE

Vessey and Sravanapudi (Ves95) provide an extensive set of references and motivation on the requirements for multi-user CASE. They divide the needed functionality into taskware (basic CASE functionality, no communication necessary), teamware (CASE information sharing, access control and monitoring), and groupware (non-CASE communication, time and meeting management). They exclude taskware from their investigation of collaboration support in existing CASE tools; the absence of communication places it outside their field of interest. We agree with their opinion that most groupware functionality ‘could be provided by generalized, task-independent packages (e.g. electronic mail, bulletin boards, calendaring capabilities)’: MetaEdit+ thus does not duplicate such functionality. Thus the most prominent needs are for teamware, in particular the ability to share information, with concurrency control ‘to resolve conflict and support tightly coupled group activities’. They perceive groups as working most frequently in asynchronous mode, but also sometimes needing to access shared resources at the same time.

Newman-Wolfe et al. (New92), writing about the Ensemble concurrent graphics editor, sum up the desired behaviour for collaboration in editing thus: “sharing should be as transparent as possible to the user, yet details of that sharing should be available if desired”. This is the guiding idea we have followed in implementing the concurrency behaviour of MetaEdit+. Chen et al. (Che93) include as the first requirement for a software engineering database that there should be consistency within a transaction: “Data changes due to a transaction are not visible until the transaction has successfully committed”.

An important point to consider for a CASE repository is the behaviour that designers are used to: a prime rule of human computer interaction is to avoid unexpected behaviour. Designers are often programmers too, and programming collaboration tools in general have an atomic transaction concept, e.g. SCCS and Envy. Thus the same atomicity should be observed in a CASE repository. The workaday world is suggested as a paradigm for CSCW design (Mor90): in other words, the work practices in use before computerisation should be those followed (with improvements) in the computerised support. Before multi-user CASE tools — and in many cases even after their introduction — designers worked on their designs largely alone, and the end product of each mini-cycle of design and improvement was released to colleagues. Thus the interface that each designer worked with was a paper version of another colleague’s work, which became out of data over a period of days to weeks, before being replaced by an updated version. This suggests that there is little need for synchronous updates, but a constant need to view the most up-to-date *released* version of someone else’s design, even if they are currently changing that design. This view is endorsed by Marmolin et al. (Mar91) who conclude that in design work the need is especially for asynchronous co-working: synchronous co-working does not seem to be important. Newman-Wolfe's requirement for availability of details would motivate the low-key display of information that that design is being updated, and availability of more information, e.g. who is updating it.

2.3 MetaEdit+

MetaEdit+ is a full metaCASE environment that supports both CASE and metaCASE for multiple users within the same environment. It supports and integrates multiple methods and includes multiple editing tools for diagrams, matrices and tables. It was developed in the MetaPHOR project, which had earlier developed the single user MetaEdit metaCASE tool (Smo91). Figure 1 shows the architecture of MetaEdit+, which is client-server with the server containing a central MetaEngine and various tools.

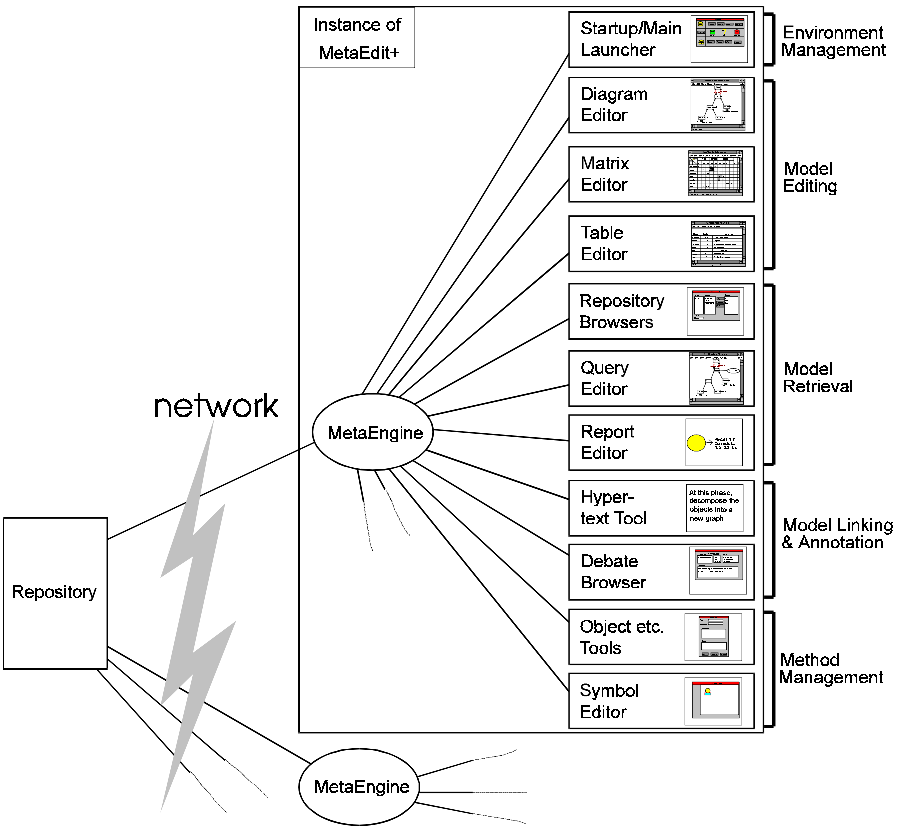

Figure 1 Architecture of MetaEdit+.

The information heart of the MetaEdit+ environment is the Object Repository. The repository is implemented as a database running at a central server: clients communicate only through shared data and state at the server. All information in MetaEdit+ is stored in the Object Repository, including methods, diagrams, matrices, objects, properties, and even font selections. Hence, modification of system designs (or methods) in one MetaEdit+ client is automatically reflected to other clients on commit, guaranteeing consistent and up to date information. The Object Repository itself is designed to be mostly invisible to users. The use of the repository is visible only when a user starts or exits MetaEdit+, opens or closes projects, and commits or abandons transactions.

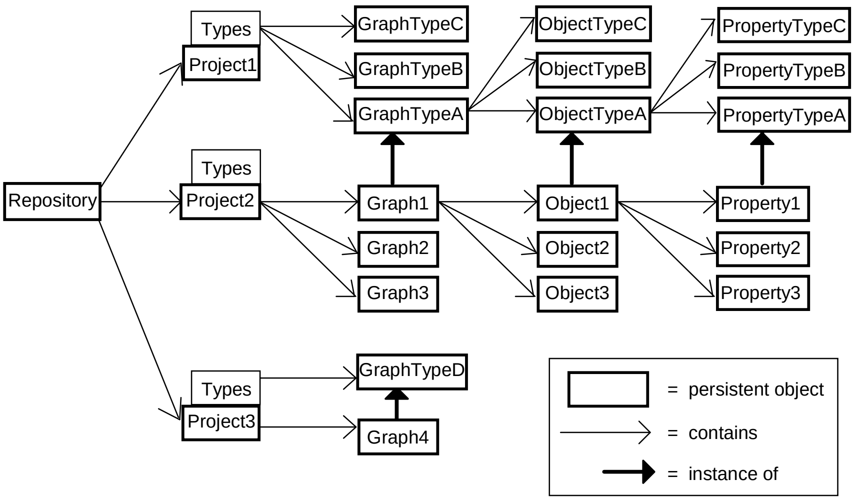

Figure 2 Structure of the repository (abridged).

A repository is composed of projects, each of which contains a set of graphs that describe a particular system, and possibly some metamodels. Figure 2 shows a partial view of the structure of a typical example repository. Project1 contains only types, which are implemented as Smalltalk classes: several graph types, each of which uses several object, relationship and role types, each of which uses several property types. Project2 contains only instances of types, i.e. graphs that contain objects that have properties. These are instances of types defined in Project1: for instance, Graph1 is an instance of GraphTypeA. Project3 contains both types and instances, with an example shown of Graph4 being an instantiation of GraphTypeD. Omitted from the figure for the sake of clarity are representations: each graph instance may have several representations, for instance as a graphical diagram or as a matrix, of its conceptual contents. Similarly, objects, relationships and roles have representations which are stored within the appropriate graph representation. Opening a project reads all the graphs in that project, so they are visible to users, e.g. in browsers. However, not all objects, properties etc. are read: these are only read as they are needed, e.g. when they are being displayed in a graph which the user opens. Objects are cached when read, and thus are only read once per session over the network: performance after that initial read is identical to non-persistent objects, and the object is only read again if another user changes it.

On a lower level, each project exists as an *area* in the repository. Each persistent object is actually persistent in one particular area. MetaEdit+ stores in each area one persistent ‘project’ object, which simply acts as a root by pointing to all graphs in that area: from the graphs one can follow on to read all other parts of model data in that area. Projects thus directly contain only graphs.

2.4 ArtBASE

ArtBASE is a library of classes which add persistence to Smalltalk objects, plus a separate Smalltalk server program (Art93). The server is the same for all ArtBASE applications. ArtBASE has been tested in various applications in both industrial and public sector settings, with hundreds of simultaneous users accessing the same repository. The work required by the client application programmer is small, as there is no separate database sublanguage. The only calls necessary are to make an object persistent; to increase performance it is also normal to mark objects as needing to be saved when they have been changed. This is a much smaller amount of work than is generally needed even for an OODBMS, and represents a tiny fraction of the code needed for interfacing with a relational database: see e.g. (Bec94).

ArtBASE automatically implements optimistic concurrency: transactions are only allowed to commit if they do not conflict with reads and writes of other overlapping transactions that have already committed. Thus the repository is guaranteed consistent automatically. To avoid users having to abandon their work in transactions, ArtBASE also allows pessimistic concurrency: objects can be read, write or exclusive locked before they are operated on, thus preventing operations that would later cause a transaction to be unable to commit.

ArtBASE transactions are fully ACID, although some of the constraints can be relaxed, for example by turning off checking of read-write conflicts. ArtBASE supports the highest degree (3) of consistency, as defined in Gray et al. (Gra76): reads are repeatable within a transaction, i.e. the value read will not change; changes are only visible to other users when committed; all writes from a transaction are committed together; and users cannot overwrite data changed but not yet committed by another user.

A similar approach to that used in ArtBASE was taken by Riegel et al. (Rie88) in the Alltalk system. There the Smalltalk object engine was changed to calculate from a transitive closure from the database which new objects needed to be stored in the database, and to recognise from assignments when an object had been changed and needed to be updated in the database. However, Alltalk was not commercially available, and the article states that there was no support for locks or other mechanism for controlling sharing of data.

There have been other commercial products similar to ArtBASE, but none available in 1993 (when MetaEdit+ development began) treated classes and metaclasses as first class objects and allowed free linking between any objects. OODBMSs such as GemStone required maintenance of the schema to be specified in both Smalltalk and their own proprietary schema language: we would thus have been forced to keep two separate descriptions of the schema and maintain their consistency each time any change was made to metamodels. Chen et al (Che93) evaluated GemStone for software engineering, finding that its concurrency support was poor: if one user made a change and committed, no other user could commit *any* change. To see the other user's changes, users had to first log out and log back in again (this has since been improved). Another possibility, Distributed Smalltalk (Ben90) was ruled out because the implementation did not allow class changes to be propagated to other users, rendering class-based metamodelling impossible. More recent versions of these environments may have overcome some of these drawbacks.

3 LOCKS IN ARTBASE AND METAEDIT+

In MetaEdit+ we have based our concurrency support on automatic locking strategies rather than user-handled versioning or configuration management. This relieves designers of the cognitive load of handling concurrency themselves (e.g. by always making explicit versions), freeing them to concentrate on design, whilst at the same time guaranteeing the consistency of the repository.

In this section we will look at how locks are used innovatively within MetaEdit+ to provide a high level of concurrency whilst maintaining consistency. First we will look at the basic concepts of locks in ArtBASE and their general use in MetaEdit+, and then at the specific application of these concepts to different kinds of data in MetaEdit+.

3.1 Concepts

There are three basic concepts which we will use in our explanations: session, transaction, and lock. We will first describe these, and also classify the different kinds of data in MetaEdit+ from the point of view of locking.

Sessions

In MetaEdit+ a session is defined as the time from when a user logs in to the repository to when he logs out. As a rough guide, a session would normally last for a work day or some part of a day, and is generally the same as the time for which the user is running MetaEdit+. Each session is composed of one or more transactions.

Transactions

In MetaEdit+, a transaction is an atomic unit of work: until a transaction is committed, other users cannot cannot see any of the work done during that transaction. Users end a transaction either explicitly by *committing* it or implicitly by logging out. Transactions also provide a measure of undo functionality by allowing to *abandon* a transaction.

At the start of each session, when a user logs in to the repository, a transaction is started for that user. The repository will remember its state at that instant, and throughout the transaction the repository will provide information *as it was at the instant the transaction was started*. Similarly, none of the changes the user makes to the information in the repository will be visible to other users until the user ends his transaction by *committing* it. At that point all his changes will be written to the repository, and will then be available to other users, but only read by them when they next start a transaction (remember that their current transactions will still be using the information available from the time they were started).

Locks

We have tried to base our user-visible multi-user behaviour on the everyday world (Mor90). In the everyday world, a person locks something if he wants to prevent others from manipulating it or removing it. A lock will prevent changes — your bike wheel disappearing, or your house being damaged — but in general will not prevent others seeing something (they can look at your bike, or peer in through your windows). The situation is in many ways similar in MetaEdit+, but here the main aim is to prevent two people making changes to the same information, rather than directly destructive acts. Thus if another user has locked something, you can still look at it, but you cannot change it.

In addition to its automatic optimistic concurrency control, which would not allow sufficient concurrency for CASE (cf. Bec94), ArtBASE supports read, write and exclusive locks. These locks can be obtained in one of two duration modes: transaction and session. A transaction lock is automatically released (the information is unlocked) at the end of the current transaction, releasing it for other users. A session lock persists over into each new transaction, until the information is unlocked by some other action. For instance, graphs are session locked while they are open in an editor, and the session lock is only released when the user closes the editor.

When a user attempts to change a piece of information in MetaEdit+, that piece of information will first be write locked, and only if that lock was successful will the change be allowed. A lock is successful if nobody else has held a lock on that piece of information in a transaction which overlaps with his.

No information is ever read locked. Because no atomic operation in MetaEdit+ reads one data value and writes other data on the basis of that value, this is perfectly safe as regards low-level consistency. On a higher, semantic, level, it is of course possible for a user to make a change based on information he can read but which is currently being changed. We do not regard this as a problem, but rather a common part of the design process: users are already used to basing their own work on the last released versions of others' work. We believe the benefits of being able to view all data at all times significantly outweigh the problems, following the research results of (Mar91, New92, Ves95). MetaEdit+ makes visible to the user if information is being changed by another user, thus allowing them to proceed, contact the user for details of the changes, or commit to read the changed version.

Types of data

For our purposes, we can divide data in MetaEdit+ into four kinds:

- conceptual objects, relationships, roles and properties
- conceptual graphs, and representational graphs and their elements
- projects, i.e. collections of related graphs
- metamodels.

A representational graph is a diagram, matrix or table. A conceptual graph is the 'real' data that underlies that representational graph: a conceptual graph may have several different representations. Information stored by a representational graph and its elements includes the positions of individual symbols, the order of items on an axis in a matrix, or the widths of the columns in a table. Correspondingly a conceptual graph stores information about which objects belong to the graph, how they are connected together via relationships, and what other graphs they explode to. Graphs in MetaEdit+ are organised into projects, which consist of a name and a set of graphs. Information may be freely linked and reused between different projects, but each graph belongs primarily to only one project.

3.2 Automatic locking strategies

MetaEdit+ automates all the functionality connected with supporting multiple users: it is perfectly possible to use it without knowing anything of the underlying principles. This is made possible by a set of automatic locking strategies: by inferring from user behaviour which operations he is about to perform on which data, we can lock that data in advance, thus guaranteeing that he will be able to perform the operations, or, if the lock fails, he can see that he can only view the data, and can also see who holds the lock.

Thus locking is performed automatically by MetaEdit+ on behalf of the user, based on the user's actions. In contrast, starting and ending sessions and transactions are actions that are explicitly carried out by the user. Why this distinction? Virtually every action in MetaEdit+ requires some kind of locking operation or check, and the burden on the user of manually setting and releasing the locks would be huge. In addition, the safety of the work done in a transaction depends on the correct locks being obtained at the correct times: any mistake, and the transaction will probably be unable to commit because of conflicting changes with another user. Transaction commit on the other hand is a question of dividing work up into semantically coherent units, the general size of which depends on many situational factors. Similarly, if a transaction were automatically committed, the user would then lose the possibility of aborting and thus undoing the actions of that transaction. Thus only the user himself can decide with any accuracy when to commit.

In MetaEdit+, locks are used differently depending on the kind of information and the current circumstances. Here we explain the types of locks and the different locking strategies in use in MetaEdit+. These different types of lock and locking strategies are designed to reflect the normal pattern of CASE usage, to give the most efficient and invisible support.

As we have seen, ArtBASE already provides the locking primitives. On top of these we have developed a LockingSystem (Luo96) which interfaces with the MetaEngine and provides the following functionality:

- It automates the request of various frequently met collections of locks, so that either all locks are obtained, or none (if some lock is not available)
- It provides handling, reporting and logging of failed locks
- It modifies caching of lock information to improve the default ArtBASE behaviour.

Much locking in MetaEdit+ is handled by the MetaEngine through the LockingSystem without tool implementers needing to worry about it. Tools and editors in MetaEdit+ are responsible for locks specific to their representation data, and call the utility functions of the LockingSystem for these.

We shall now look at how the automatic locking works with respect to different kinds of data. We shall proceed in order of increasing size, examining locking for objects, relationships, roles and their properties, then for graphs, then for projects, where we introduce a new multi-user persistent collection. Finally we shall examine the special locking solutions required for metamodels.

Objects etc.

Individual objects, relationships, and roles, and their individual properties are locked only when the user explicitly opens them in a property dialog. When a user opens a dialog on an object's properties, MetaEdit+ attempts to lock *all* that object's properties, so they can be changed. If one or more of the locks fail, then no locks are taken, and the user can only view the properties in the property dialog: the **OK** button will be greyed. If all locks were obtained, the user will be able to press the **OK** button to accept his changes. The reason behind locking all properties and not just those that are changed is two-fold. Firstly, we want to take the locks when the property dialog is opened, before the user starts to make changes, so he can see straight away whether his changes would be accepted. Secondly, the information in the various properties is normally semantically interlinked, even though there are no links in the actual data: the range of semantically correct values in one property of an object depend on the values of the other properties. If only changed properties were locked, two users could make changes to non-intersecting subsets of the properties with no lock conflicts, but resulting in a semantically inconsistent state of the objects' properties.

Graphs

When opening an editor on a representational graph, MetaEdit+ will automatically try to obtain locks both for the representational graph, and for the underlying conceptual graph. The success of these locks determines which actions the user may perform in that editor: if one or both locks fail, the editor will still open, but some of the menu items will be greyed, and other e.g. mouse operations may have no effect. The conceptual objects etc. and representational elements within the graph are not locked: the conceptual objects are thus still free to be edited by other users who access them via an editor without locks on this graph, or via any other graph or place they are reused. In contrast, representational elements are not reused, and thus cannot be reached and edited other than via an editor on this representational graph. The editor only allows modifications to representational elements if the lock on the representational graph is held, thus the representational elements are effectively 'locked', but without the overhead of explicitly locking each one of them.

For instance, if a diagram is opened and locked successfully, but the conceptual graph is not able to be locked, the user will be able to move symbols around in the diagram, but not be able to add new objects or relationships to the graph. He will however be able to add a symbol for an object that already exists in the conceptual graph, or show a relationship that already exists there. If neither lock were obtained, the user's actions will be restricted to scrolling, zooming, viewing selected types, and editing the properties of the objects etc. in the graph. Lock information in editors is visible through the menu bar, and the user may view further information about who holds any locks which he was not granted.

If a user knows he is opening a graph for viewing only, he can specify this while he opens it, and he will then not be granted any locks and will be unable to modify it. He can of course reverse this decision later and open it again normally, thus attempting to gain the locks.

Projects

One of the most difficult aspects of implementing locking in MetaEdit+ was projects. Each project stores a collection of all the graphs it contains: many users may simultaneously (i.e. in overlapping transactions) want to add a new graph, and therefore need to write lock and modify the collection itself. The traditional solution to such problems of shared collections has been to use a B-tree (or similar), and an implementation of B-trees already existed in ArtBASE. However, the B-tree structure only becomes efficient once the number of leaves becomes large, yet at the start of a project, when new graphs are being added at the greatest speed, the collection is initially empty. The largest natural number of graphs for a project is well below 100, thus a B-tree approach, where a typical node size is 50, would be inefficient in terms of storage space and performance. Similarly, the performance of a general-purpose B-tree is at its worst if keys are inserted in an ascending order: splits occur often and 50% of the storage space are wasted. Logical OIDs, however, form the only possible key for graphs in general, and these are allocated in ArtBASE in an ascending order, as is general in object-oriented databases.

More seriously, index structures like B-trees have proved to be a serious bottleneck of the system if they are updated by multiple users simultaneously. Several techniques to improve concurrency and recovery have been proposed and tested (Sri93). Nevertheless, the implementation of these algorithms is difficult and frequent modifications still reduce the performance of the system significantly. In particular, concurrency appears to be at its worst when the collection is small, whereas our need for concurrency is highest then, as many users create many graphs in a new project.

As projects are not expected to grow to contain much more than 100 graphs, and we have no need for fast key access — and indeed no useful source of keys — we do not benefit from the positive sides of B-trees, and are seriously affected by their negative sides. To solve this problem I designed a new kind of multi-user collection. It may be interesting to note that Beck and Hartley (Bec94) also found the need to extend the ArtBASE-like library of classes they used in their fixed-method CASE tool with new persistent collection classes. Their additions were however simply automatic marking of the collection as changed when elements were added or removed; my MultiUserColl includes this but its main purpose was to address a somewhat more complicated problem.

The basis of the MultiUserColl collection is a persistent array containing N elements, where each element is itself a persistent object, called an Insulator. An Insulator is a simple object, which holds one other object, or holds nil if it is currently unused. Insulators are persistent in their own right, and can thus be locked independently of each other and of the parent MultiUserColl. The MultiUserColl also has a ‘chain’ variable which is initially nil, but can hold another MultiUserColl, thus forming a chain of MultiUserColls to support more than N members in the collection. Initially a new MultiUserColl contains an empty Insulator in each place.

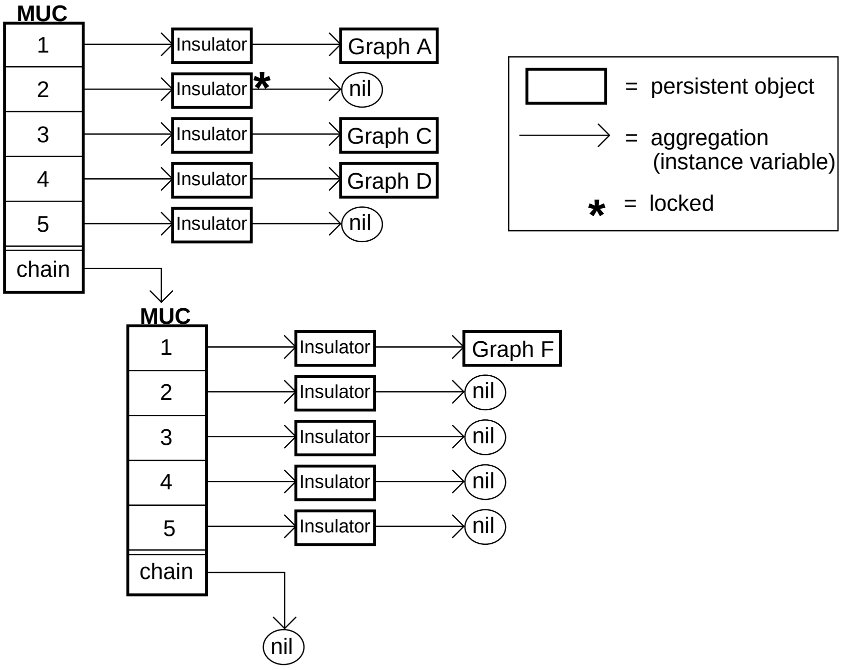

Figure 3 A MultiUserColl after several operations.

The figure shows a MultiUserColl as seen in one client after several transactions have added graphs (Graphs A to F, which caused a second MultiUserColl to be chained on to the first) and also removed some graphs (as seen by the empty Insulators at slots 2 and 5, where Graphs B and E were). The Insulator at slot 2 is locked but empty, because another user has added a graph (say Graph G) there in an overlapping transaction: this client cannot see that graph until after commit, but can see that the slot is locked, and thus cannot be used.

Iterative and collection operations on the MultiUserColl are redefined so that they operate only on the held values of non-empty Insulators, and so that they invisibly follow on to any chained MultiUserColls. Thus the standard collection API of the MultiUserColl behaves identically to other collections, hiding the implementation details from application programmers. When adding objects, the MultiUserColl scans through its Insulators to find the first empty Insulator for which an attempted lock is successful. It then places the added object into that Insulator. If there are no empty lockable Insulators in the chained MultiUserColls, it attempts to lock the last MultiUserColl to chain a new MultiUserColl to it, and add the object in there. If the chaining lock fails, an error is reported. In addition, a MultiUserColl allows pre-locking of the next free Insulator slot (including automatic chaining of a new MultiUserColl if necessary), to ensure that an approaching add operation will be able to execute. This enables better error handling, as the user is told right at the start of attempting to create a new graph that the operation cannot succeed, and is aborted. Removal (e.g. deletion of a graph) is simpler: the relevant Insulator is locked and reset to empty; if the lock fails, the user is informed that the graph cannot be removed by him; in fact, the graph must have been removed already in an overlapping transaction (the only possible situation where this user could see the Insulator as non-empty and locked).

For example, if we try to add a new graph with the situation in the figure, Insulator 1 is already used, Insulator 2 is empty but our attempt to lock it will be refused, Insulators 3 and 4 are used, but our attempt to lock the empty Insulator at slot 5 will be successful, and we can place our new graph in that Insulator.

The value of N, i.e. the size of each MultiUserColl, can best be determined by experience within a particular organisation, first roughly by examining the number of graphs created within a transaction, and then more accurately by experimentation with different values of N to set the minimum value that yields an acceptably low incidence of refused locks when trying to create a new graph. The use of too high a value of N will merely slow the system down a little, as a larger MultiUserColl and larger number of Insulators must be read than necessary. In practice, we have used a value of N=10 with 9 intermittent users of a shared repository for a year, and not once has a lock been refused, even when on several occasions most users have been logged in and modelling new data in earnest.

Because the current implementation and value of N have proved sufficient for our needs, we have not further extended the MultiUserColl to allow even more concurrency. One possible simple extension would be to make add operations attempt to grow the MultiUserColl by chaining before it is totally full — say when it is 80% full. This would allow more time for the new chained MultiUserColl to be committed and made available to other users, who in the meantime would be using the last 20% of Insulator slots to store their new graphs.

Locks for metamodelling

Modifying information on the type level differs significantly from instance level changes, in that changes made to types affect every instance of that type in the repository. Changes to types may fundamentally change the whole method and way of working which other users are following, and should therefore be treated as more dangerous than instance level changes. To the best of our knowledge, existing metaCASE tools only allow one person to make changes to types (i.e., to metamodel) at a time, and while changes are being made no other users of any kind (even modellers) may be logged in.

MetaEdit+ aims to provide better possibilities for metamodelling simultaneously with modelling, and for multiple simultaneous metamodellers. It offers three levels of concurrency: one single exclusive metamodeller; one metamodeller and several modellers; or several metamodellers and modellers. The last level allows one metamodeller for each project (in practice, for each method). The choice of which level to use is left up to the repository administrator, being dependent on local work practices and contingencies.

4 EVALUATION

Vessey & Sravanapudi (Ves95) evaluate several multi-user CASE tools on the facilities they offer for task, team and group work. They divide their analysis into control, information sharing and monitoring (teamware aspects) and co-operation (groupware aspect). Control covers security and access rights. Information sharing consists of CASE data sharing, including hypertext and queries; consistency enforcement; and concurrency control. Monitoring covers issues of timestamping, marking of creator and modifier, and logging. Co-operation includes provision of electronic mail and meeting schedulers.

Within each aspect there were several binary questions, each basically representing a desired functionality. There were different numbers of questions for each aspect, possibly reflecting the authors' view on the relative importance of each aspect. For each positive answer, i.e. piece of functionality present, a tool received one point in that aspect. The tools performed much better on information sharing than any other area, with co-operation being the weakest area: unsurprising, as the authors themselves recommend that it could mostly be handled by external tools.

We applied the criteria given to MetaEdit+, and the results are shown in Figure 4. The tools examined by Vessey & Sravanapudi were only fixed-method CASE tools, for which provision of multi-user facilities is easier than for metaCASE environments. Even so MetaEdit+, including its method engineering support, would seem to perform well on the criteria, often performing as well as or better than the best tool in a category, and even at its worst relative to the tools examined is only one point behind the best in that category. It is worth noting that the tools examined supported only structured analysis and design methodologies, and that the analysis seemed to take a largely relational database view of concurrency. An important area not addressed in the analysis is how fine is the granularity of locks, and thus how closely users can work concurrently: in this area MetaEdit+ would be significantly better than the CASE tools examined. Similarly, the analysis does not take into account the unique facilities of MetaEdit+ for several concurrent metamodellers and modellers.

The scores given for MetaEdit+, however, cannot be directly compared to those obtained for the other CASE tools. The other evaluations were performed and agreed on by several people, who were presumably unbiased. Whilst I attempt to be unbiased, I may still take a different interpretation of some criteria than the earlier evaluators. Because of these inherent problems and obvious constraints of space, I do not set down here my justification of every point given to MetaEdit+

(individual answers are not given in (Ves95) either). Perhaps Vessey and Sravanapudi, or some other researchers, would be interested in extending their criteria and tool selection to cover multi-user metaCASE tools.

Some of the criteria listed in (Ves95) were not provided by any tool they examined. It is interesting to note that generally MetaEdit+ does not provide such functionality either: perhaps this indicates that those features are not in fact desirable in CASE (e.g. their suggestion of calendar or time management facilities). One area where all tools faired badly, but MetaEdit+ provides some support, was identifying components based on timestamps or change information, e.g. MetaEdit+ marks all method components with their creation date, time and user. Overall, the performance of MetaEdit+ can be fairly summarised by saying that it generally implements those criteria which are also implemented in some other tool, i.e. practically the union of all tools' sets of implemented criteria. Its main contribution, however, is that it implements these as a metaCASE tool, rather than the simpler 'single fixed method' CASE tools studied in (Ves95).

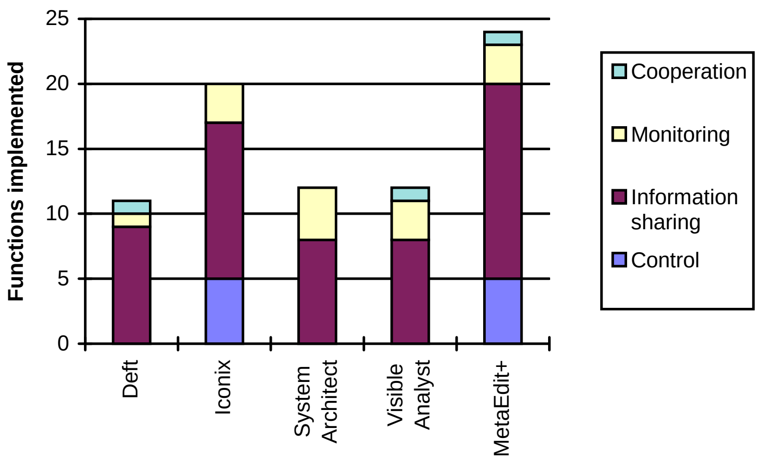

Figure 4 Collaborative support in some CASE tools and MetaEdit+.

5 CONCLUSIONS AND FURTHER WORK

MetaEdit+ is the first metaCASE environment that supports multiple simultaneous metamodellers and modellers in the repository. It is based on a persistent object store for Smalltalk using standard transaction semantics. By always allowing free reading of data and its fine locking granularity, MetaEdit+ obtains several of the benefits expected from non-standard transactions or other models entirely without transactions.

MetaEdit+ implements concurrency control with fully automatic locks, relieving the user of the conceptual burden of explicitly performing locking, check-in/check-out or versioning. Locking granularity varies for different kinds of data, and no read locks are needed, allowing a high degree of concurrency whilst maintaining consistency. A new concurrently updatable collection data structure was developed, providing high concurrency even for small collections, where B-trees perform most poorly. This solved the problem found in CASE work that the collection of graphs grows fastest, and thus has the highest density of concurrent updates, when it is smallest.

Whilst MetaEdit+ was seen to perform well in its support for collaborative work even compared to existing fixed-method CASE tools, there remain some areas which could be extended. Many organisations have a need for a large number of repository readers using light clients, thus work is in progress to provide a WWW interface to MetaEdit+, that would allow people to browse the repository without needing to run MetaEdit+. The interface works via a normal WWW server calling a small cgi-bin C program that passes the request on via a socket to a slightly extended MetaEdit+ client, which accesses the requested data, formats it into HTML and GIF graphics, and returns it via the cgi-bin program and server to the user.

Currently MetaEdit+ has not been tested with more than 10 concurrent users, with which it performed well. The underlying ArtBASE database has however been tested in other applications with hundreds of users. Overall, we have found that an object store such as ArtBASE forms a sound basis on which to build a sophisticated CASE environment. Whilst object store technology is not yet as mature as relational or object DBMSs, its value as the basis for a CASE repository is clear.

MetaEdit+ thus represents the first application to a metaCASE tool of the collaborative facilities found in the best fixed-method CASE tools. It is to be hoped that our experiences and reports of its design will encourage and help other (meta-)CASE manufacturers to add groupware support to their products, and that MetaEdit+ itself will prove useful to many in research and industry.

5.1 Acknowledgements

I gratefully acknowledge the coding and extensive testing work of Janne Luoma and Marko Somppi on the integration of the ArtBASE multi-user extensions into MetaEdit+. My thanks go also to Prof. Jari Veijalainen, for his useful comments relating the MetaEdit+ repository to other research in databases.

6 REFERENCES

Ald91 Alderson, Albert, "Meta-CASE Technology," pp. 81–91 in *Software Development Environments and CASE Technology, Proceedings of European Symposium, Königswinter, June 17–19*, A. Endres and H. Weber (Ed.) No. 509, Springer-Verlag, Berlin (1991).

Art93 ArtInApples, "*ArtBASE: Distributed Smalltalk and Object-Oriented Database Management System,*" ArtInApples Ltd., Bratislava, Slovakia (1993).

Bec94 Beck, Bob, Steve Hartley, "*Persistent Storage for a Workflow Tool Implemented in Smalltalk,*" ACM SIGPLAN Notices (Proceedings of OOPSLA '94) 29(**10**) (1994) pp.373–387.

Ben90 Bennett, J. K., "*Experience with Distributed Smalltalk,*" Software — Practice and Experience 20(**2**) (1990) pp.157–180.

Ber96 Bernstein, P. A., "*The Repository: A Modern Vision,*" Database Programming & Design 9(**12**) (1996) pp.28–35.

Che93 Chen, S., J. M. Drake and W. T. Tsai, "*Database requirements for a software engineering environment: criteria and empirical evaluation,*" Information & Software Technology 35(**3**) (1993) pp.149–161.

Gra76 Gray, J. N., R. A. Lorie, G. R. Putzolu and I. L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base," pp. 365–394 in *Modelling in Data Base Management Systems*, G. M. Nijssen (Ed.), North Holland (1976).

Kel96 Kelly, S., K. Lyytinen and M. Rossi, "MetaEdit+: A fully configurable multi-user and multi-tool CASE and CAME environment," pp. 1–21 in *Advanced Information Systems Engineering, proceedings of the 8th International Conference CAISE'96*, P. Constapoulos, J. Mylopoulos and Y. Vassiliou (Ed.), Springer-Verlag (1996).

Kel97 Kelly, S., K. Lyytinen, H. Liu, P. Marttiin, H. Oinas-Kukkonen, M. Rossi and J.-P. Tolvanen, "*MetaEdit+: CASE Functionality to Support Production, Coordination and Organizational Control And Innovation,*" Chapter 4 in S. Kelly's PhD thesis "Towards a Comprehensive MetaCASE and CAME Environment", University of Jyväskylä, Finland (1997).

Kra95 Kraut, R. E., L. A. Streeter, "*Coordination in Software Development,*" CACM 38(**3**) (1995) pp.69–81.

Luo96 Luoma, J., M. Somppi, "*Concurrency Control in Multi-User MetaEdit+ (Samanaikaisuuden hallinta monen käyttäjän MetaEdit+:ssa),*" Master's Thesis (in Finnish), TKTL, University of Jyväskylä, Finland (1996).

Mar91 Marmolin, H., Y. Sundblad and B. Pehrson, "An Analysis of Design and Collaboration in a Distributed Environment," pp. 147–162 in *Proceedings of ECSCW '91 2nd European Conference on CSCW* (1991).

Mor90 Moran, T. P., R. J. Anderson, "The Workaday World as a Paradigm for CSCW Design," pp. 318–393 in *CSCW 90 Proceedings*, ACM (1990).

New92 Newman-Wolfe, R. E., M. L. Webb and M. Montes, "Implicit Locking in the Ensemble Concurrent Object-Oriented Graphics Editor," pp. 265–272 in *Proceedings of the 1992 Conference on Computer-Supported Cooperative Work*, Jon Turner and Robert Kraut (eds.) (Ed.), ACM Press, Toronto, Canada (1992).

Rie88 Riegel, Steve, Fred Mellender and Andrew Straw, "Integration of Database Management with an Object-Oriented Programming Language," pp. 317–322 in *Advances in Object-Oriented Database Systems: 2nd International Workshop on Object-Oriented Database Systems*, K. R. Dittrich (ed.) (Ed.) Vol. Lecture Notes in Computer Science No. 334, Springer-Verlag, Berlin (1988).

Rup95 Rupnik-Miklic, E., J. Zupancic, "*Experiences and expectations with CASE technology — an example from Slovenia,*" Information & Management 28(**6**) (1995) pp.377–391.

Sel94 Selamat, M. H., C. Y. Choong, A. T. Othman and M. M. Rahim, "*Non-Use Phenomenon of CASE Tools: Malaysian experience,*" Information and Software Technology 36(**9**) (1994) pp.531–537.

Smo91 Smolander, Kari, Kalle Lyytinen, Veli-Pekka Tahvanainen and Pentti Marttiin, "MetaEdit — A Flexible Graphical Environment for Methodology Modelling," pp. 168–193 in *Advanced Information Systems Engineering, Proceedings of the Third International Conference CAiSE'91, Trondheim, Norway, May 1991*, R. Andersen, J. A. Bubenko jr. and A. Solvberg (Ed.), Springer-Verlag, Berlin (1991).

Sri93 Srinivasan, V., M. J. Carey, "*Performance of B+ Tree Concurrency Algorithms,*" VLDB Journal 2(**4**) (1993) pp.361–406.

Sto93 Stobart, S. C., A. J. van Reeken, G. C. Low, J. J. M. Trienekens, J. O. Jenkins, J. B. Thompson and D. R. Jeffery, "An Empirical Evaluation of the Use of CASE Tools," pp. 81–87 in *Proceedings of the 6th International Workshop on Computer-Aided Software Engineering, CASE93*, Hing-Yan Lee, Thomas F. Reid and Stan Jarzabek (Ed.), IEEE Computer Society (1993).

Tai97 Taivalsaari, A., S. Vaaraniemi, "TDE: Supporting Geographically Distributed Software Design with Shared, Collaborative Workspaces," pp. 389–408 in *Proceedings of CAiSE '97, Barcelona, Catalonia, Spain, June 16–20*, A. Olivé and J. A. Pastor (Ed.) Vol. 1250, Springer, Berlin (1997).

Ves95 Vessey, I., A. P. Sravanapudi, "*CASE tools as collaborative support technologies,*" CACM 38(**1**) (1995) pp.83–95.