CHAPTER 5

APPLICATION OF REPOSITORY TECHNOLOGY AND CONCEPTS TO A METACASE ENVIRONMENT

Manuscript (submitted), 1997

APPLICATION OF REPOSITORY TECHNOLOGY AND CONCEPTS TO A METACASE ENVIRONMENT

Steven Kelly
kelly@cs.jyu.fi

MetaEdit+ is a metaCASE environment which allows multiple simultaneous metamodellers and modellers. We describe how it forms a repository for (meta-)CASE data, and examine how it matches up to the requirements for a repository given by Bernstein, in particular its architecture, MetaEngine repository engine, generic tools, and tools using the repository. Important features of MetaEdit+ include its extensibility and extensive support for reuse and navigation. We describe the automatic locking strategies that enable MetaEdit+ to offer a high level of concurrency whilst guaranteeing consistency. In particular we describe a new collection data structure that allows high concurrency of updates even at small sizes. Finally, we evaluate MetaEdit+'s collaboration support, comparing it with several CASE tools.

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. 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.

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 (Kel97b); in this paper we concentrate on its use as a repository for (meta-)CASE data.

Note that MetaEdit+ is not being proposed as a generic repository in the same sense as those database / repositories currently appearing on the market, although it too can store anything from program code to business data. It is not used to implement the business processes or store the current number of widgets in stock; rather, it is used to model business processes and information systems. Thus information is one level higher than in a normal business database: our data corresponds to what is normally called metadata in a business database. Similarly, we do not concentrate on the underlying database technology that MetaEdit+ uses, but rather on how we have innovatively used and in places extended existing leading edge database technology.

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 MetaEdit+ repository according to Bernstein's framework of repository requirements (Ber96). Bernstein does not cover locking, and a major source of interest in MetaEdit+ as a repository is its automatic locking strategies. In the second half of the paper we thus leave Bernstein and look in more detail at the novel locking solutions and data structures adopted in MetaEdit+. 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. Research on communication within a design tool has focused on synchronous design mostly at a very early stage of brainstorming. This research is of course useful, and tools to support it are needed. However, the activity patterns of designers at that phase differ significantly from those met during the majority of the life of a design project. The early stage consists of largely unstructured synchronous brainstorming with generally no fixed notation, while the later stage sees more structured asynchronous design using specific method notations. These two types of design activity differ so significantly in their natures that it is unlikely that both could be served well by one system.

In support of the earlier phases, research is ahead of practice; in the latter phases, practice has tried to rush ahead of research: despite the proliferation of research prototypes for synchronous multi-user design discussion sessions, general practice is still to use physical media; in contrast, although little interest has been paid to multi-user functionality in the design development phase, at least simple multi-user CASE tools have been taken into use to a fair extent. This perhaps reflects the fact that it is possible to build a simple multi-user CASE tool with a good knowledge of a programming language and database, without investigating the fundamental concepts of CASE. However, attempts to make more sophisticated CASE tools multi-user seem to founder: in particular many of the few manufacturers who have announced such 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, three other significant research projects have worked on extending CASE and metaCASE technology with possibilities for computer-supported collaborative work (CSCW). The first is the work of Grundy and Venable (Gru96b), which includes the CoCoA language for modelling methods, the MViews class framework providing the basic building blocks to make a CASE tool, and the C-MViews extensions which allow multi-user CASE (Gru96a). Currently, these are not however integrated into a single functioning environment, although several prototypes combining two of the three components exist. Their approach to CSCW is to merge change histories: the user is informed when other users' changes are automatically merged into the state of his models. Some changes cannot be made automatically, in particular there are two important and frequently occurring types of conflict, where the user must be asked for help in how to perform the merge. Whilst MViews could be viewed as having some measure of metaCASE functionality, it is more appropriate to describe it as a library to help CASE tool builders.

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.

The third project, a prototype multi-user co-ordination and negotiation tool for software engineers, was developed in the CoNeX project (Hah91), based on the experimental DAIDA environment. Although the CoNeX tool still needed 'further refinement', it would be possible to use it to record design rationale synchronously among several users.

In addition to these, 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.

There also exist several databases which provide some existing CASE editors and support for building new CASE editors for other methods. These cannot be regarded as metaCASE environments, because of the requirement to write program code in order to implement the tool functions of the new CASE environment. Such databases include Maestro II (Mer91) and GOODSTEP (GOO95). ConceptBase is a deductive temporal repository which also has some basic CASE functionality. It implements basic graphical CASE support, but the graphical representation information is not stored in the repository, and symbols are limited to one visible label, which must be unique. Further, it requires that the whole repository fit in main memory, and multi-user support is limited. Lincoln (previously IPSYS) ToolBuilder 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 separate 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.

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)'. 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 about 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. The 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.

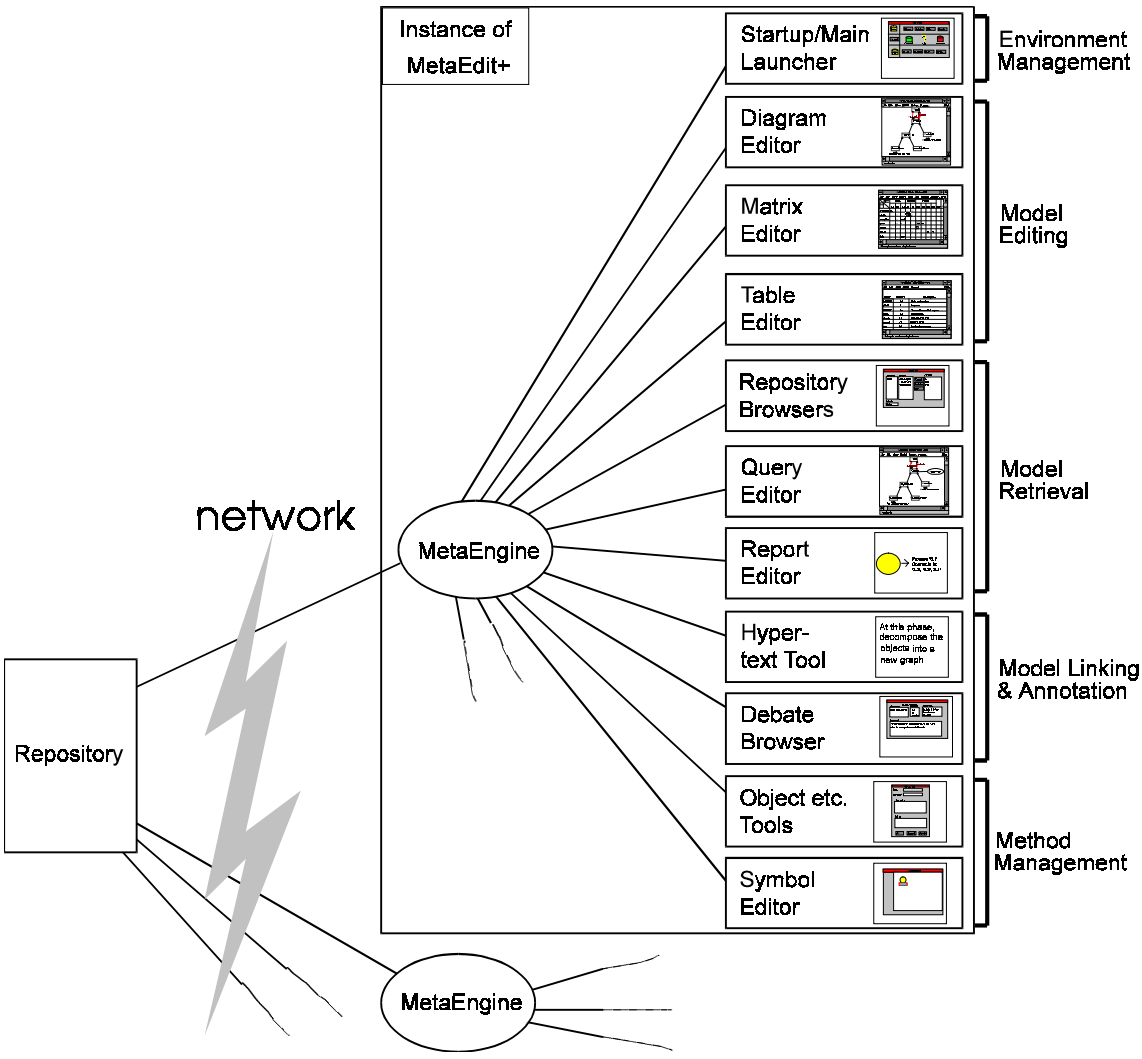

FIGURE 1 Architecture of MetaEdit+

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 the 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 transaction over the network: performance after that initial read is identical to non-persistent objects.

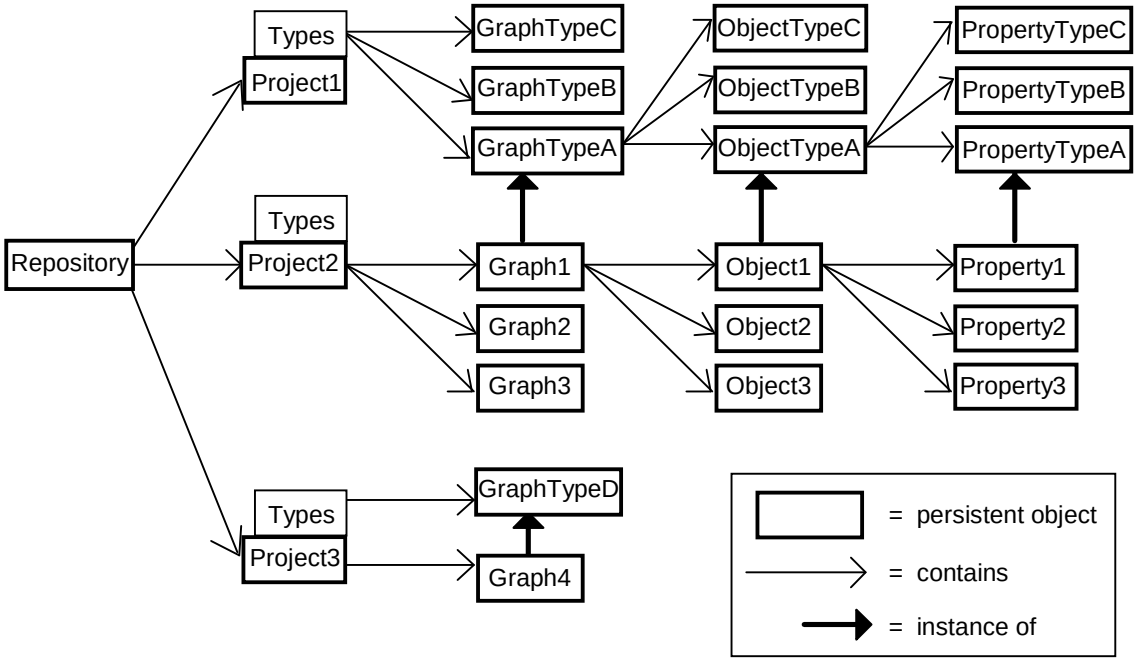

FIGURE 2 Structure of the repository (abridged)

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. Normally opening a project loads all the graphs and their identifying properties and adds them to the list of graphs visible to the user. Users may open several projects, thus loading all their graphs: the browsers allow filtering of graph display by project (Kel97b). One project is chosen as the default project where newly created graphs go: this can be changed at any time. This reflects the general pattern that users work on a single project at a time, but may be simultaneously part of several ongoing projects, and wish to consult or reuse information from previous projects.

Objects from one area may be reused in a graph in another project, even though they are in a different area from that graph. When such a graph is

opened, the project area containing that object is opened quietly and only that object is loaded: the other project is not considered as being open, i.e. the graphs of that project are not loaded and thus not visible in browsers. Similarly, if a project is opened that contains models but not their metamodels, the area of those metamodels will be opened quietly in the background, so that the metamodels from the other project are available, but the graphs from that project are not read. In this way MetaEdit+ allows free linking of data between projects without leading to the situation where accessing objects rapidly leads to opening and reading all data in all projects.

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 persistent 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 many of these environments may have overcome some of these drawbacks.

Whilst ArtBase represents leading-edge technology in terms of object-oriented data storage, it does not in itself form a repository in the sense of (Ber96), but may serve well as the underlying database for such a repository, as we use it in MetaEdit+. In the next section, we will examine to what degree MetaEdit+ measures up to Bernstein's description of a repository.

3 The Repository: A modern vision

Bernstein (Ber96) describes requirements for a repository, based partially on the current state of the art and partially on desires for further developments. We will use his article as the basis for description, discussion and evaluation of the MetaEdit+ repository. Whilst the purpose of this article is not to describe a generic repository as such, a metaCASE environment forms a kind of repository itself, specialised for dealing with information systems models and methods. MetaEdit+ extends this idea beyond normal CASE data to include methods and models for business process engineering and systems architecture, thus further approaching the idea of a generic repository.

Bernstein lists several different kinds of situations where a repository is needed, and MetaEdit+ falls into his second category: data integration support for high-end CASE tools. Thus the MetaEdit+ repository provides this support to the rest of the MetaEdit+ environment, and through that to its end users.

This separation into 'repository' and 'rest of the environment' is based on Bernstein's understanding that a repository involves more than just data storage, but does not itself form a total application. Thus some higher level functionality and even tools are considered as belonging to the repository, some tools are considered as being closely related to the repository, and other tools are considered as forming the application that uses the repository. In MetaEdit+ the repository, in Bernstein's usage, consists of the data stored at the server, the server itself, the MetaEngine at the client and some generic tools at the client.

These generic tools are those that operate only on conceptual data, not representational data. Thus the diagram, matrix and table editors are considered as being part of the MetaEdit+ application — and indeed it is in these that a user spends most of his time in CASE work — but most other tools are either part of or related to the repository.

Bernstein describes the repository architecture, the repository engine, generic repository tools, and tools using the repository. We will look at those four areas and the requirements Bernstein has for each, and see how MetaEdit+'s repository answers those requirements.

3.1 Repository system architecture

The architecture consists of an information model, used by the application to interpret the data in the repository database and its behaviour; a repository engine in the client which abstracts the functionality of reading and writing to the repository away from the main client application, and a repository database, which is the database system on the server, including the data stored and the server program. We will examine each of these as it exists in MetaEdit+.

3.1.1 Information model

MetaEdit+ uses the GOPRR model for its models and metamodels. GOPRR deals only with conceptual data, so each editing tool adds its own set of classes to specify how representational data is stored and behaves. The MetaEngine in MetaEdit+ includes the GOPRR classes, and also the abstract superclasses for representational data, thus significantly reducing the work needed for each editing tool, and also making the editing tools consistent.

3.1.2 Repository engine

ArtBase in the client consists of a number of classes that implement the repository engine, making reading and writing data transparent to the application, and reducing other database operations such as locking or committing to simple single API calls. In addition, MetaEdit+'s MetaEngine adds some high-level operations which provide consistent behaviour for both data and tools in MetaEdit+. A good example is the opening and locking of graphs in editors, which is implemented in one place in a few lines of code in the MetaEngine, and yet handles the locking and opening of any graph in any editor, and even the disabling of those menu items in each editor which could not be used because the user does not hold a lock for editing that data in that graph.

3.1.3 Repository database

The ArtBase server program is a normal Smalltalk program that runs on a server machine: platforms supported include most Unixes and Windows 3.11, 95 and NT. The clients and server communicate through sockets, although if sockets are not available it is also possible to use a shared file for communication. The data is stored as a set of files divided into a directory for

each project area in the database. Project areas can be distributed, so that they are stored closest to the clients that use them most often: the only requirement is of course that the server and clients must be able to access the files, e.g. through mounting the directory via NFS or Samba.

3.2 Repository engine

Under this heading we cover the issues of object management, dynamic extensibility, relationship management, version management and configuration management within the MetaEngine of MetaEdit+.

3.2.1 Object management

While most CASE tools until recently have used relational or network databases, these lead to problems with newer methods. Older structured methods in general required the user to perform extra work in assigning a unique key to each object (e.g. a Process number in a Data Flow Diagram), but newer methods, e.g. for object-oriented design, are better supported by an object-oriented database. The problem is similar at the level of the underlying database: it is important for a repository to read and write objects, not records, rows or entities, to avoid the burden of constantly translating between the different representations. As Bernstein puts it: "A repository that manages objects in a manner conforming to the underlying object model is easier to use and integrate with tools designed for that object model, compared with repositories that are integrated through an adapter layer". The ArtBase database is a totally object-oriented extension of Smalltalk, keeping everything as objects and simply allowing them to be persistent. It thus goes one step further than an OODBMS in removing the adapter or interface layer between the program and its database.

3.2.2 Dynamic extensibility

This means it must be easy and efficient to add new type definitions and extend existing ones. It can be viewed on two levels: it should be easy to 1) extend the repository to meet the requirements of a new tool, and 2) create new object types and instances immediately in an existing repository-enabled tool. In (Kel94b) we showed how simple it was to extend MetaEdit+ with a new tool, a matrix editor: the information model, repository engine and database did not need to be changed at all. The only additions were the editor itself and classes for its representations, which were both able to inherit or use much code from the MetaEngine's generic behaviour for tools. The creation of new object types and their testing by creating new instances is simple in MetaEdit+: (Kel96) describes it in detail. Briefly, MetaEdit+ uses a simple form-based GUI for creating types, in contrast to the textual metamodel programming languages used in most other tools. Further, to our knowledge MetaEdit+ is the only metaCASE environment where new types can be instantly instantiated and thus tested: all other tools require some kind of export-compile-link-run cycle to

upgrade the separate CASE tool with the new type defined in the metaCASE tool.

3.2.3 Relationship management

Bernstein correctly observes that relationships form a large portion of data in a repository, and are much more complicated than in traditional databases. MetaEdit+ is no exception. There are two basic kinds of relationships, those within a graph between objects, and those between graphs, such as explosion, decomposition, object reuse and property sharing. Both kinds have been a significant area of research in designing the information model of the MetaEdit+ repository. The separation of two distinct natures of relationships within graphs was shown in (Kel95) to improve both the conceptual model and the implementation performance. The various relationships between graphs in CASE, previously used haphazardly in different tools, are analysed and dealt with in a coherent way as described in (Kel94a). In particular the paper emphasises the new kinds of relationships that occur when objects and properties are reused in different graphs: this is discussed further below.

3.2.4 Version management

As Bernstein recognises, version management is significantly more difficult in an object-oriented repository than in old file-based relational systems. When creating a new version of an object, we must decide whether to create new versions of the other objects linked to this. For example, when we create a new version of a DFD graph, should we also create new versions for each of the objects in that graph, and even for each of the properties of those objects. Bernstein is forced to leave the question open: versioning complex objects is still an active research topic of its own, and no clear, easily applicable solutions have yet been found. ArtBase automatically stores all versions of all objects, but does not address the issue of complex object versioning, in other words it remembers which objects were contained in version 3 of a graph, but does not remember which versions of those objects were current then. A Master's thesis project (Pyy94) has produced a possible design for versioning of complex objects in MetaEdit+, but this has not yet been implemented.

3.2.5 Configuration management

Configuration management according to Bernstein is the grouping of versions into coherent units. This is somewhat redundant in a CASE scenario, where the coherent units are already defined in the form of projects and graphs: other more generic repositories do not have such a helpful and easily identifiable aggregation structure. Versions and configurations can lead to additional complexity when we also consider types: (Che93) and (Ska86) discuss problems of versioning of types and its effect on their instances. ArtBase supports basic versioning of all types (classes), remembering which version of a class an object instantiated, and automatically performing the basic necessary changes to upgrade that instance to a newer version of the class when it is read later (lazy updates), e.g. removing attributes or adding new empty ones. MetaEdit+

supplies the additional functionality needed for updating instances to fully conform to the type definition, for instance by filling in new empty attributes with properties of the correct type holding their default initial value. Consistently with ArtBase, this is performed lazily, only when that attribute is read. This removes the need to update all instances of a type when that type changes which renders many other databases impractical for metaCASE tools.

However, as with version management, this is clearly an area where MetaEdit+ is currently lacking, and there is a need for a comprehensive solution covering both to provide proper support for large, long-term projects.

3.3 Generic repository tools

Bernstein divides tool support into those functions that operate at a fairly low level, generically, with semantics left free for the user to decide, and those that operate on a higher level, with more defined semantics. This subsection describes the first set, including browsing, scripting, import/export and extensibility, and the second set is covered in the next subsection.

3.3.1 Browsing

In addition to the editors etc. provided in an application, users need the ability to browse the repository contents in a generic browser. MetaEdit+ provides two such browsers, one viewing information on types and by type, and the other viewing information on graphs and by graphs, i.e. using instantiation and aggregation as their main principle. Both browsers allow restriction of the viewed objects by project, by metatype and by wildcard strings on properties and types. They are described in more detail in (Kel97b).

3.3.2 Scripting

MetaEdit+ provides a high-level reporting language and report writing tool which operates on the GOPRR model data in the repository. Reports can be written by typing or by using the context-sensitive menu-driven editor: they can then be stored in the repository and run on graphs. Bernstein limits this to simple code generation (MetaEdit+ comes with reports for Java, Delphi, C++, Smalltalk, SQL etc.) but its application can be much wider: textual reports, documentation, checkings and statistics. MetaEdit+ supports documentation generation with basic text formatting (fonts, styles, sizes), and more complicated layout can be achieved by using textual formatting languages such as HTML, RTF or TeX. Reports can easily be written to check for illegal combinations of property values, classes having several superclasses etc. The Reporter also provides support for the build tool described later in Bernstein, because reports can follow links to other graphs and continue running there. Thus with a single command the user can generate code for a whole system consisting of many graphs and sub-graphs. The Reporter is described in more detail in (Kel96).

3.3.3 Import/export of metadata

In MetaEdit+ the basic data is CASE models, i.e. one level higher than in the business repositories Bernstein has in mind. The equivalent of metadata in our situation is thus actually metamodels. MetaEdit+ supports the export and import of metamodels via a Type Manager tool. This provides intelligent assistance in choosing what to export - the user in general would not be able to take the correct necessary and sufficient set of metamodel components, because of the complicated relationships between them: if we export a type, we must also guarantee that all types it needs are exported or already present in the target repository. Users are however capable of specifying which graph types they need, and from these MetaEdit+ can calculate which other object, relationship, role and property types are required. This set of types is displayed to the user, and for each type the user can view the reasoning that led to its selection.

When importing types, if any components to be imported already exist in the target repository, MetaEdit+ updates the existing components rather than creating new ones (class names are uniquely timestamped with creator and date to give a basis for object ID over different repositories, thus distinguishing between a type that has been imported and a type that just happens to have the same name). This enables an organisation to have several separate repositories using the same method, and to update that method to all organisations periodically. One large multinational corporation is already making extensive use of this functionality in MetaEdit+ over several sites in Europe and the USA.

3.3.4 Extensibility

Research shows that designers rarely use an existing method as it is defined, but rather extend it, modify it, or even create their own methods on the fly: Russo et al. (Rus95) found that 85% of method users reported modifying the method. As Twidale et al. found (Twi93) "Design is viewed by designers as a creative and personal activity. It involves the development and normalisation of concepts relating to the artefact being constructed. Designers tend to adopt flexible and personal notations to express these concepts." Ter Hofstede and Verhoef (Hof96) studied experienced designers using Structure Systems Analysis in the Netherlands, and found that they almost invariably made significant changes to the method they were using. Fixed-method CASE tools prevent such changes, and other metaCASE tools make the process sufficiently long that the intuitive creative element is lost. MetaEdit+ allows fast intuitive type changes and additions, automatically updating the existing models to reflect changes where appropriate. The new and changed types are of course immediately available for testing and use.

Easy extensibility is vital in Bernstein's opinion: "Making it easier to add type definitions makes it easier to add more tools, which in turn increases the repository's value." In another article, we have described how MetaEdit+ supports easy addition of new tools (Kel97b) and covered in detail the addition of a matrix editor (Kel94b). Grønbæk (Grø94) describes a similar approach with addition of new types of hypertext data and new tools to handle those types.

His additions of new types however always requires programming, contrary to Bernstein's requirement in the next paragraph.

Extensibility should be supported at a high level: "It is usually cumbersome to add a new type definition by calling repository engine functions directly. This may be simplified by offering a type definition language that compiles into the appropriate calls on the repository engine". Such an approach was followed in MetaEdit (Ros95), and also in several other metaCASE tools. Bernstein would like further sophistication: "Developers can simplify work even further using an object design tool, which translates a graphical definition of the type into calls on the engine". MetaEdit allowed this from a graphical metamodel diagram via a series of steps using an external tool (Ros95). MetaEdit+ allows this directly via a form-based graphical user interface (Kel96); previous metaCASE tools have used textual or sometimes graphical languages, but these have then needed to be transformed and compiled before changes have any effect. We have performed an empirical study (Kel97a) on performance of metamodellers with different representations of metamodels, specifically diagram and matrix formats, which tends to suggest that several different representational paradigms would be useful in a metaCASE environment. Thus we plan to extend MetaEdit+ with alternative metamodelling representations, whilst keeping the form-based GUI for direct access to the metamodels.

MetaEdit+ also supports Bernstein's requirement for modelling of external dependencies: methods can be extended with concepts for external objects, modelled as the user wishes. Currently under development is a property type of external object that would allow live links to Word documents, sound files etc.

3.4 Tools using the repository

In addition to the above generic functionality, Bernstein requires that a repository offers tools that use the repository. These tools are distinct from the actual applications that may be built on top of the repository, although they will of course often be used by the application. They support the higher level functions of the user organisation, including reuse, linking, business rules, data warehousing and team development. In this section, we examine the tools in MetaEdit+ that offer this kind of support.

3.4.1 Reuse library

MetaEdit+ is designed from its information model up to provide strong support for reuse. All GOPRR components can be reused, on both type and instance levels. In particular, graphs display a type-free interface to reusers, allowing them to be reused across different methods, but still supporting the linking of interface relationships of an object in a higher level graph to the objects within the lower level decomposition graph. This allows graphs to be reused in a similar way to components in CAD, including both black-box and white-box reuse.

Tools support for reuse is built into the MetaEngine, and is thus available in all editors, browsers etc. This includes the ability to select graphs, objects, relationships, roles and properties for reuse, selecting them based on their type or via another component that already uses them, e.g. a DFD Process object could be reused by viewing a list of all Processes or by first selecting a DFD from a list of all graphs and then selecting the Process from that graph's set of objects. In addition, browsers offer wildcard string-based queries against type and identifying property. Each GOPRR element can be viewed in an Info Tool, that shows details of where it is currently used, what links it has to other components, and which of its own components are reused where.

A graphical Query Editor has been developed for MetaEdit+ (Liu95), and this can be used to search for reusable components by their structure and relationships with other components. Queries are defined 'by example', by drawing the desired structure as in a normal graphical diagram, and specifying types and extra constraints such as Booleans and wildcards on property values.

Reuse should also be informed by information about the design rationale and history behind the component being reused. MetaEdit+ allows design rationale to be attached to both conceptual (GOPRR) and representational components. Design rationale and history take the form of Question-Answer-Argument graphs, and can be viewed as diagrams or through a list-based browser (Kai97).

3.4.2 Intertool navigation

Bernstein requires that a user should be "able to move quickly and easily from one tool to another, following the links of an object." This idea has been made more explicit by Brinkkemper (Bri93), who calls it 'modelling transparency' and provides a scale of transparency from 0 (all tools separate, must exit to view another diagram) to 3 (all tools integrated, free linking between diagrams). MetaEdit+ supports this in many different ways, and indeed goes beyond Brinkkemper's level 3 by allowing metamodelling and modelling in the same environment. We do not endorse the view that all linking should be free, as in flow charters and TDE (Tai97); rather, there should be typed links for relationship, explosions, and decompositions, following the rules of the method, and these should be supplemented by the ability to freely link any components with hypertext links, and to navigate amongst components by their various links (as in the Info Tool).

MetaEdit+ allows hypertext linking and annotation of both conceptual (GOPRR) and representational components. Several types of links are supported, including annotations, association jumps, and a traceability link that can be used to document the flow of information from a requirement through the various components that answer it in different phases to the eventual low-level design component that implements it. Hypertext functionality is described more fully in (Kai97).

From an Info Tool a user can open another Info Tool on one of the components listed, or can open a normal editor to show e.g. where this object is used in another graph.

3.4.3 Impact analysis

The Info Tool also answers Bernstein's later requirement for impact analysis: when an object is changed, the Info Tool can be used to list and open the other components that use the object, so they can be modified if necessary.

Automation of impact analysis is desirable: "Sometimes a tool can automate the process of making fixes. For example, when changing the name of an SQL stored procedure, a tool can automatically replace all instances of the old name with the new one." MetaEdit+ takes a more object-oriented approach: the name is stored only once, and all relevant objects refer to that one name, so the change affects all those objects without any need for an automatic search/replace. This reuse works similarly in the case of other GOPRR components on both type and instance levels.

3.4.4 Business rule management

In addition to normal systems analysis and design methods, MetaEdit+ supports methods for business process engineering, allowing the modelling of business objects and rules specified in these methods. Looked at from another point of view, business rules are simply the integrity constraints of a repository, and thus in our repository map to one level higher, i.e. the metamodels. The metamodels specify the kinds of models the user can create and the rules the components in those models must obey. Looked at this way, the requirement is answered by MetaEdit+'s metamodelling capabilities and extensibility, discussed above.

3.4.5 Data warehouse management

We are currently developing 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. Because of the atomicity of transactions, the MetaEdit+ client will return data consistently from a snapshot of the repository, until it is explicitly told to commit and thus refresh the state of its cached data. This has the double benefit that the MetaEdit+ client's cache will grow to contain the majority of data requested, improving response times to WWW browsers, and that cached data will not be requested from the MetaEdit+ server, reducing the load on it.

3.4.6 Team development

"When multiple developers work together on a complex project, they must coordinate their changes": Bernstein writes that version and configuration management, in addition to there standard functions described earlier, are also the key repository technologies that address co-ordination of work. The solution he outlines is, he admits, complex, and we feel it to be impractical. Most CASE users are not experts on the whole project and its interdependencies, but rather just on their small area, thus requiring all users to

be able to specify the interrelations of their work with others and manage their own co-operation is expecting too much.

For these reasons, in MetaEdit+ we have based our concurrency support on automatic locking strategies rather than user-handled versioning and configuration management. The version and configuration management functionalities are then needed to complement this functionality, rather than provide it. This relieves designers of the cognitive load of handling concurrency themselves by always making explicit versions, freeing them to concentrate on design, whilst at the same time guaranteeing the consistency of the repository. In the next section, we describe our automatic locking strategies and their implementation.

4 Locks in ArtBase and MetaEdit+

Bernstein's description of the repository is a useful framework for analysing most repository functionality, but there is a rather prominent omission: he makes no mention of locks, and provides no details of how any other concurrency mechanism (e.g. versioning) would work. Vessey and Sravanapudi (Ves95) however cover these concurrency control and consistency maintenance aspects of teamware in addition to the basic information sharing that forms the basis of communication on the product between developers. 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+.

4.1 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.

4.1.1 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+. If a user wants, he can however exit MetaEdit+ without ending his session: the state of his transaction is saved in an image file, which he can start later to resume work. In the mean time, as far as the server and other users are concerned, it is as if he were logged in all the time, e.g. he still holds all locks held at the point the long transaction image was saved. Aside from such long transactions, each session is composed of one or more transactions.

4.1.2 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).

Thus for one user's changes to be visible to another user, the first user must commit his transaction, and the second user must start a new transaction after that, either by ending his transaction (and thus starting a new transaction) or by logging in (if he is not already).

The normal length for a transaction will depend largely on the way of working in an organisation. If there is a need for very fast updates of information between different designers, transactions may last from half an hour to an hour. Where such rapid updating is not necessary, users can reduce the overhead of ending and starting transactions by using transactions of a day or even longer.

4.1.3 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.

The interval between a lock being taken and a change being made varies with different kinds of information: most often it is negligible, but with some information, particularly graphs, locks are by default taken when a user first opens the graph, even though he may not change it immediately. The lock guarantees that he will be able to make changes when he wants to.

4.1.4 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.

4.2 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.

4.2.1 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.

4.2.2 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.

4.2.3 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.

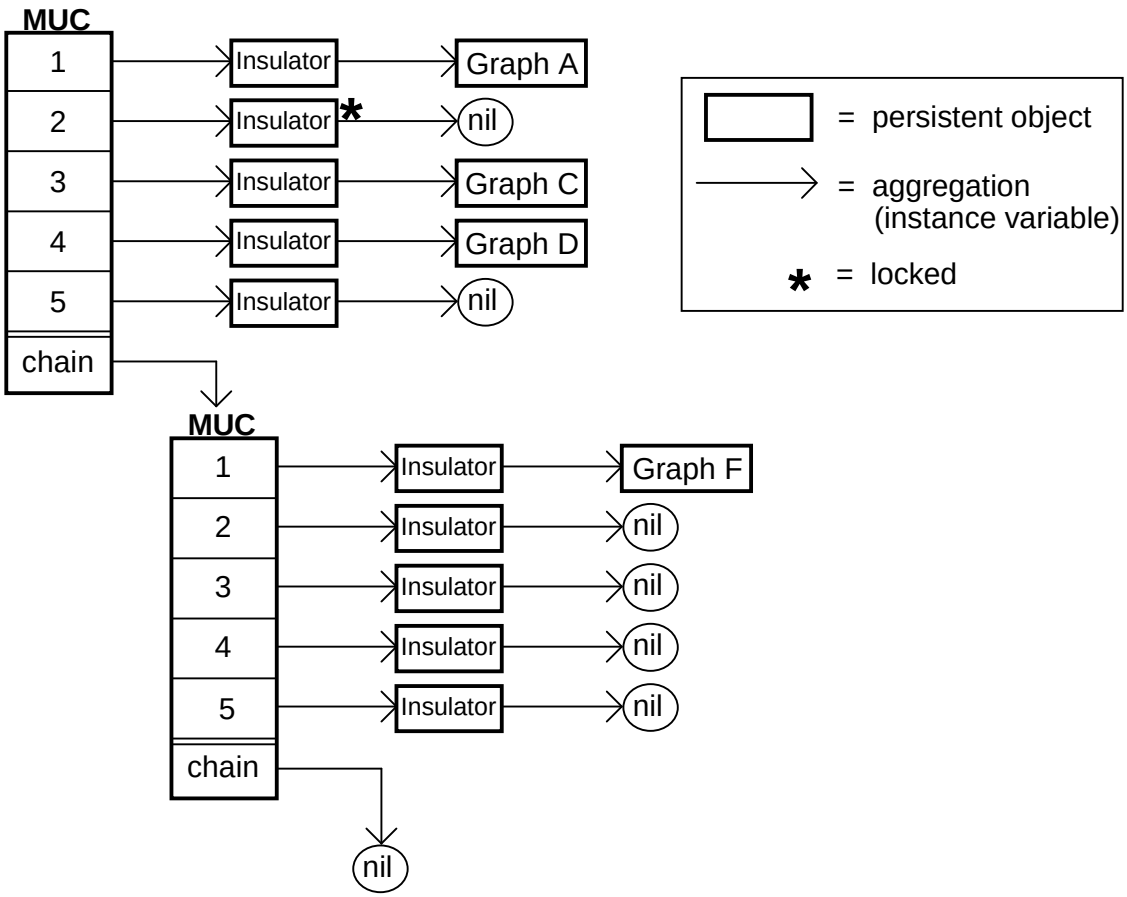

FIGURE 3 A MultiUserColl after several operations.

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.

4.2.4 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. In a large multi-user repository, it is a practical impossibility to lock every instance of a type, and the situation becomes even more complicated when we take into account links between types — for instance, a change to a property type will potentially affect its subtypes, and also any other types that use that property type, and of course all their instances. In contrast, changes on the instance level affect only immediately related instances, which form a sharply defined and easily calculated and reachable set. This is as true from a user's perspective as it is on a technical level: it feels somewhat odd to have the types you are instantiating change under your very nose, whereas it is somehow more logical that an instance object made by your colleague may change.

In addition, 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. Different companies follow different practices with regard to type level or comparable schema changes in any kind of database: some forbid all but the system administrator from making such changes, others have a single user or group of users who specialise in metamodelling and are authorised to make changes, still others (especially small companies) may allow anyone to make changes.

There may also be different policies or customs related to database changes: in old relational databases, all other users were excluded from the database while the schema was updated. Whilst we can now offer better possibilities than this, some companies may decide to perpetuate this way of thinking: it does after all protect the user from the unsettling feeling of seeing the very method he is using change. There is also the question of how much it is possible to have several simultaneous metamodellers: can they co-ordinate their work effectively, or will their changes — even if technically correct — result in a garbled mess for themselves and their users. Setting the bounds within which a metamodeller may work can be problematic: his types may all be defined in one bounded area, e.g. a single project, but may then be used in several areas, overlapping with other projects' types.

Further, the requirements for simultaneous type level and instance level users will often change throughout the life of a project: at the start of an ISD project (following Tol93), most work will be metamodelling with a few example instance models to test the types; as the method is taken into use there will be much activity on the instance level, and a few changes to the types; later it may be decided to freeze the method — even though it may still not be perfect, the detrimental effect of changes to existing models, and the burden of unlearning and relearning, outweigh the benefits of the remaining minor corrections that could be made. Thus the requirements for metamodelling access rights and locks vary over time within a single project.

Clearly, with all these factors to be considered, many of them strongly dependent on contingency, usage environment and culture, a single strategy for type level locking would not be appropriate for the majority of organisations. We are thus forced to deviate from the simpler 'single strategy' solution we applied on the instance level, and create multiple strategies, allowing the system administrator to choose which one to apply at any given time. Obviously, the number of possible strategies should not be too high, as these settings will be made rarely: we should not overburden the system administrator with choices of which he has little knowledge or experience.

Starting from end user requirements, there is a clear requirement that metamodelling access rights should be restricted to a certain set of users, specified by the system administrator. At no time can any other user carry out metamodel changes. Our interest however lies more with locking than access rights: what are the locks that users require?

1. **Repository Exclusive**
   Reflecting the habits and thought patterns developed in relational databases, we can require that changes to types can happen only when nobody else is

logged in to the repository. This also protects users from the unsettling effect of types changing before their eyes.

2. **Repository**
   Considering the danger and wide-reaching influence of type changes, we can have a single metamodelling lock for the whole repository: other users may be logged in, but only one user may make changes to types at a time. This helps solve the problems associated with the web-like nature of metamodels, which makes coherent simultaneous multi-user modification difficult.

3. **Project**
   There are several benefits to be gained in general by modelling different methodologies in different projects, perhaps all using some shared types from a core project. Thus it is useful to be able to lock projects individually for metamodelling. This allows many users to metamodel simultaneously, but their work is semantically separated by the project division, thereby avoiding problems of contention, changes in semantics etc.

The first option corresponds to the situation in the older relational databases, and also to that in other existing metaCASE tools, to the best of our knowledge. The second allows the work pattern proposed by Tolvanen (Tol93) and observed by Russo et al. (Rus95): methods are changed as they are taken into use. Whilst in a single project with a small number of users it would be possible to make such changes even if the repository were locked exclusively whilst metamodelling, with a larger number of users or projects there will be work continuing within the repository whilst a method is taken into use. Similarly, when a method is modified between projects, another method in the repository may simultaneously be in full use by another project.

If there are several projects, it is also possible that more than one method will need to be modified at a time. This would require the third level of locking, so that different metamodellers can make changes within different metamodel projects. A similar situation arises at the start of a project where a large method is being developed from scratch.

A possible fourth level of locking would be to lock types individually. This has been implemented in MetaEdit+, and shown to work. However, we feel that in practice this freedom is both potentially dangerous and also rarely needed, at least in the foreseeable future. The danger arises when one metamodeller modifies a type that another metamodeller is starting to use: whilst we can lock all types that currently use a type, even if we ignore the large number of locks that will often result, we cannot reasonably lock all types that might be about to use a type. Whilst MetaEdit+ handles resulting conflicts without actual errors, resulting metamodels are semantically 'wrong' far more frequently than if there is only one metamodeller per project, and the need for such high concurrency in metamodelling is dubious.

5 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+. Perhaps Vessey and Sravanapudi, or some other researchers, would be interested in extending their criteria and tool selection to cover multi-user metaCASE tools.

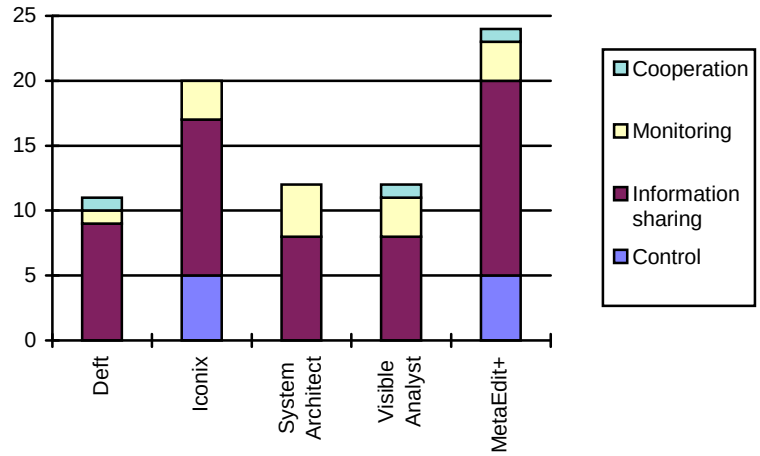

FIGURE 4 Collaborative support in some CASE tools and MetaEdit+

6 Conclusions

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.

MetaCASE environments can usefully be considered as repositories themselves. We described how MetaEdit+ considered thus met well the requirements for a repository given by Bernstein. In particular we showed how the generic functionality of the MetaEngine provided extensive support for reuse, navigation and change impact analysis throughout the tools, editors and browsers of MetaEdit+. The high extensibility of MetaEdit+ for both types and tools was described, including the intelligently-guided metadata import and export facilities, and flexible scripting language for report, code and documentation generation.

Bernstein expected concurrency control to be implemented by user-guided version management, but MetaEdit+ was able to implement it with locks. Locking was fully automatic, relieving the user of the conceptual burden of explicitly performing locking, check-in/check-out or versioning. Locking granularity varied for different kinds of data, 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.

Several locking strategies for metamodel data were provided for the system administrator to choose between. With the most restrictive strategy,

behaviour mirrored that of other repositories and metaCASE systems, with metamodel (schema) updates only possible when there were no other users logged in the repository. The strategy with the highest concurrency allowed multiple normal users and metamodellers to work simultaneously in the repository whilst still maintaining consistency.

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. Although MetaEdit+ does not need version management for concurrency, versioning and configuration management per se is important for CASE users, and should thus be extended beyond the basic repository snapshots provided currently. Similarly many organisations have a need for a large number of repository readers, thus work is in progress to provide a WWW interface to the MetaEdit+ repository.

Currently the repository has not been tested with more than 10 concurrent users, with which it performed well. The bulk of the load appears to be at the client, and whilst this would probably be beneficial to scalability, the work done by the client during commit when large amounts of data have been read is significant, requiring over a minute for 1MB of data on a Pentium 90MHz. This effectively rules out short transactions, which might in some situations be effective; our current experience however suggests an average transaction length of half an hour is sufficient in practice for teamwork. The reasons for this slowness at commit lie in the way persistence is implemented in Smalltalk by ArtBase, and possibly also in its somewhat inefficient data storage. More recent versions of ArtBase promise to address these problems, and should thus be tested.

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+.

References

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.

Bri93 Brinkkemper, S., "*Integrating diagrams in CASE tools through modelling transparency*," Information & Software Technology 35(**2**) (1993) pp.100–105.

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.

GOO95 GOODSTEP_Project, , "*The GOODSTEP Project Final Report*," ESPRIT Project 6115, http://www.dbis.informatik.uni-frankfurt.de/REPORTS/GOODSTEP/goodstep.html, University of Frankfurt, Germany (1995).

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).

Grø94 Grønbæk, Kaj, Jawahar Malhotra, "*Building Tailorable Hypermedia Systems: the embedded-interpreter approach*," ACM SIGPLAN Notices (Proceedings of OOPSLA '94) 29(**10**) (1994) pp.85–101.

Gru96a Grundy, J. C., J. R. Venable, J. G. Hosking and W. B. Mugridge, "Supporting Collaborative Work in Integrated Information Systems Engineering Environments," in *Proceedings of the 7th Workshop on the Next Generation of CASE Tools (NGCT'96), Crete, May 20-24* (1996).

Gru96b Grundy, J. C., J. R. Venable, "Towards an Integrated Environment for Method Engineering," pp. 45–62 in *Method Engineering '96: IFIP WG 8.1/8.2 Working Conference on Principles of Method Construction and Tool Support, Atlanta, August 26-28*, S. Brinkkemper, K. Lyytinen and R. Welke (Ed.), Chapman-Hall, London (1996).

Hah91 Hahn, U., M. Jarke and T. Rose, "*Teamwork Support in a Knowledge-Based Information Systems Environment*," IEEE Transactions on Software Engineering 17(**5**) (1991) pp.467–481.

Hof96 Hofstede, A. H. M. ter, T. F. Verhoef, "*Meta-CASE: Is the game worth the candle?*," Information Systems Journal 6(**1**) (1996) pp.41–68.

Kai97 Kaipala, J., "Augmenting CASE Tools with Hypertext: Desired Functionality and Implementation Issues," pp. 217–230 in *Proceedings of CAiSE '97, Barcelona, Catalonia, Spain, June 16–20*, A. Olivé and J. A. Pastor (Ed.) Vol. 1250, Springer, Berlin (1997).

Kel94a Kelly, S., V.-P. Tahvanainen, "Support for Incremental Method Engineering and MetaCASE," in *Proceedings of the 5th Workshop on the Next Generation of CASE Tools*, B. Theodoulidis (Ed.) No. Memoranda Informatica 94-25, Universiteit Twente, Enschede, the Netherlands (1994).

Kel94b Kelly, S., "*A Matrix Editor for a MetaCASE Environment*," Information and Software Technology 36(**6**) (1994) pp.361–371.

Kel95 Kelly, S., "What's in a Relationship: on distinguishing property holding and object binding," in *Proceedings of 3rd International Conference on Information Systems Concepts, ISCO 3*, W. Hesse and E. Falkenberg (Ed.), University of Marburg, Lahn, Germany (1995).

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).

Kel97a Kelly, S., M. Rossi, "Differences in Method Engineering Performance with Graphical and Matrix Tools: A Preliminary Empirical Study," in *Proceedings of 2nd CAiSE/IFIP8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design, EMMSAD'97, June 16–17, Barcelona, Spain*, K. Siau, Y. Wand and J. Parsons (Ed.) (1997).

Kel97b 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*," ACM Transactions on Software Engineering and Methodology (submitted for publication) (1997).

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

Liu95 Liu, H., "A Visual Interface for Querying a CASE Repository," in *Proc. of the Eleventh IEEE Symposium on Visual Languages (VL'95)*, Darmstadt (1995).

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).

Mer91 Merbeth, G., "Maestro II — the integrated CASE system from Softlab (in German: Maestro II — das integrierte CASE-System von Softlab)," in *CASE Systeme und Werkzeuge, 3e Auflage*, H. Balzert (Ed.), BI Wissenschaftsverlag (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).

Pyy94 Pyykkö, A., "*Version Control in a Meta-CASE Environment (Versiohallinta CASE-kuoressa)*," Master's thesis (in Finnish), University of Jyväskylä (1994).

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).

Ros95 Rossi, M., "*The MetaEdit CAME environment*," Proceedings of MetaCase 95, University of Sunderland press, Sunderland (1995).

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

Rus95 Russo, Nancy L., Judy L. Wynekoop and Diane B. Walz, "The Use and Adaptation of System Development Methodologies," in *Proceedings of the 1995 International Resources Management Association Conference*, Atlanta (1995).

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.

Ska86 Skarra, Andrea H., Stanley B. Zdonik, "The Management of Changing Types in an Object-Oriented Database," pp. 483-495 in *Proceedings of the OOPSLA'86*, Norman Meyrowitz (Ed.) Vol. 21 No. 11, ACM Press, N.Y. (1986).

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).

Tol93 Tolvanen, J.-P., K. Lyytinen, "*Flexible method adaptation in CASE environments — The metamodeling approach*," Scandinavian Journal of Information Systems 5(**1**) (1993) pp.51-77.

Twi93 Twidale, M., T. Rodden and I. Sommerville, "The Designers' Notepad: Supporting and understanding cooperative design," in *Proceedings of the Third European Conference on Computer-Supported Cooperative Work*, G. De Michelis, C. Simone and K. Schmidt (Ed.) (1993).

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