Support for Collaborative Object-Oriented Development

Boris Magnusson
Dept of Computer Science, Lund University
Box 118, S-221 00 Lund, Sweden
e-mail: Boris.Magnusson@dna.lth.se
and
Rachid Guerraoui
Dept of Computer Science, EPFL
1015 Lausanne, Switzerland
e-mail: guerraoui@lse.epfl.ch

Abstract. This paper presents an environment supporting collaborative object-oriented development in a distributed, large scale, situation. The environment provides a fine-grained version control mechanism which enables explicit representation of development histories. The *wait-freedom* principle underlying our environment ensures that a developer can always work locally without waiting for any remote communication and despite network and node failures.

1 Introduction

Although object-orientation encourages the design and implementation of applications by composing various components developed by different users, very few environments support collaborative object-oriented development. The availability of world-wide networks adds a dimension of geographical distribution to the picture and makes the problem even more urgent.

Existing techniques for version control building on *locking* of checked out *files* do not scale up to many people and geographic distribution. The problems created with locking during long check-outs grows with the number of people involved, the geographic distance and with the size and complexity of the software system maintained. This is true in general but it becomes even more problematic for OO software due to the many dependencies between modules. Here we have not only the *use* relation, but also relations through *inheritance* which may bring in more use and inheritance dependencies. The resulting network of dependencies often become very involved and surprisingly hard to overview. Changes to one file thus often result in check-out and modifications to other, dependent files. In combination with pessimistic, lock on check-out strategies, there is a problematic situation with development of OO software in distributed settings. [Thomas88] describes in detail why current revision control systems being module oriented are inadequate for OO development demanding finer granularity. This mismatch lead to large scale copying and recompilation of source code.

In the following we describe a set of requirements we have identified for collaborative object-oriented development and we give an overview of our environment.

1.1 Requirements for collaborative OO development support

Supporting collaborative OO development puts additional emphasis on a number of requirements which often are useful also in single user environments. The requirements we have formulated are as follows.

- A flexible revision control system which makes it easy to view the development history of a program (or set of software components), reconstruct any version of the program and to easily identify the differences between versions.
- The revision control system must support aggregation of program modules into systems, including version control of links between modules (representing superclass and use relations).
- The structure of a program should be determined by the logical demands on the system and not hampered by restrictions of the collaborative environment. In particular hierarchical structures must be supported (like operations nested into a class).
- The interaction with a client must be wait-free with respect to activities of other users. This means that a user can always work locally, albeit with temporarily reduced functionallity like not having access to the latest version in case of network failure.
- Support for controlled creation of new versions in particular situations such as during the release of a software system. In such situations there is a need for managed control of what modifications are included in the released version.
- Support for merge of alternative versions, created by purpose, or as a result of parallel activities of users in a partitioned network. The merge functionallity also detects conflicts between the merged versions.

1.2 Overview of our environment

In this paper we present a multi-servers environment which supports *wait-free* collaboration between developers located at different nodes of a (possibly large scale) network. This environment is based on previous work on the Mjølner project [Knudsen93, Magnusson93, Minör93].

Our environment uses an explicit representation of the development history of a program, called the *evolution graph* of the program. This graph is the main mechanism for user awareness in the environment. The model of sharing is that programs exist in a number of versions. After having been defined, a version never changes. There is therefore no problem of consistency in replication of these programs. The fundamental representation of programs supports a hierarchical organization. This suits OO development and allows the system to utilize sharing of common sub-parts between different versions. Supporting hierarchy also makes it unnecessary to artificially chop up programs into smaller parts to increase concurrency among users. Differences between versions of a program are presented as *active diffs.* These highlight edit changes between versions including structural changes rather than just different source lines. Together with the hierarchical representation of programs they serve as effective means to present the differences to the user.

Wait-freedom is one of the major design principles underlying our environment. A developer can always work locally without waiting for any remote communication, and despite network and node failures. To support this facility, we have designed a multi-servers architecture where program versions are replicated (in a lazy way) for fast access and redundancy in case of network failure. The basic communication protocol between servers completely avoids locking as an implementation restriction.

The rest of the paper is organized as follows. Section 2 presents the mechanisms for hierarchical structuring and fine-grained revision control. Section 3 describes the collaboration facilities. Section 4 presents the architecture of the environment and discusses its scalability features. Section 5 contrasts related work. Section 6 summarizes the paper and presents the status of the environment and some future work.

2 Development Support

First in this section we show the hierarchical representation of programs and the hierarchical browsing. We then give a short introduction to the fine-grained revision control functionality, which automatically keeps track of program modifications and supports merging and simultaneous editing by different users.

2.1 Hierarchical Representation and Browsing

A program [1] is organized as a hierarchical structure of fragments. The fragments typically correspond to abstractions in a programming language, such as blocks, classes, procedures, and functions. This hierarchy is fundamental in our architecture. A program is displayed using this hierarchy, it is edited in terms of the hierarchy, all elements in the hierarchy are revision controlled, and programs are stored in the servers' databases in terms of the hierarchy.

Figure 1 shows an example of the user's view of a program. In the figure the program hierarchy is at the top and an evolution graph below. The program contains several class and procedure fragments, which are shown and manipulated by a hierarchical browser allowing parts at one level to overlap but not to be moved outside its parent window. Subparts are added and removed in the browser by selecting entries from a menu, e.g. "new class". The parts in the figure contain text which is edited by a text editor. The approach for hierarchical browsing was initially developed in the Mjølner Orm environment [Hedin 88] where it turned out to be a useful and "intuitive" way of viewing and editing

1. We use the term 'program' here in a generic fashion. It may be a program fragment, typically a class or a module. Most of the discussion is also relevant for text documents.

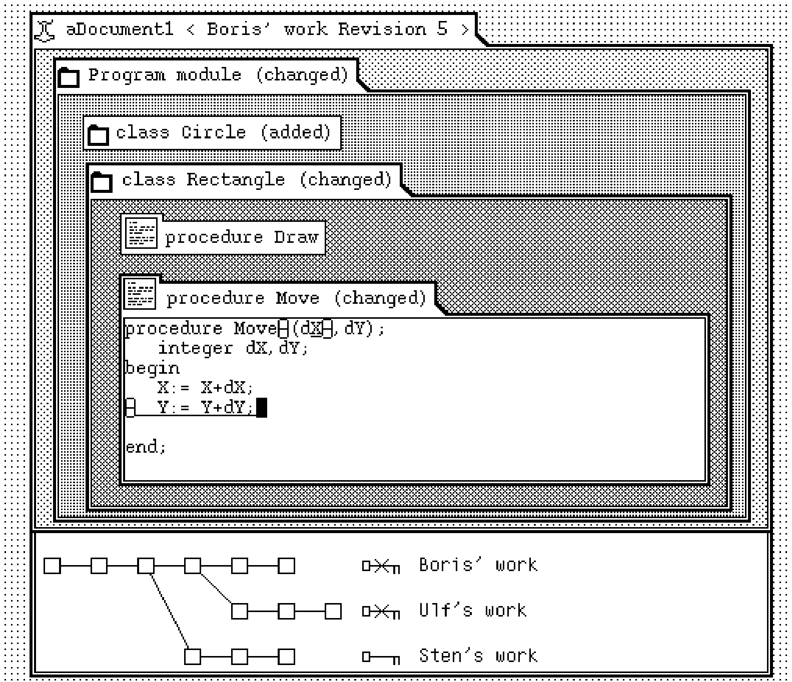

Figure 1 *Hierarchical program presentation*

program structures. Furthermore, in a collaborative environment it may be used as a means to organize the work.

2.2 Fine-Grained Revision Control

All fragments of the hierarchy are revision controlled. The lower part of figure 1 shows the evolution graph of the program. An evolution graph contains results from three different actions: creating revisions, creating alternatives, and merging alternatives.

A revision is one step in the evolution of a program. It may contain arbitrary changes compared with its predecessor . Typically it may contain new classes and procedures, parts may have been removed, or it may contain minor modifications such as error corrections. A revision is never changed once it has been frozen. The only way of modifying a program is to create a new revision with the desired modifications.

This hierarchical fine-grained revision control functionality , where all fragments (in figure 1, for instance) are revision controlled, is implemented as basic functionality of servers' databases for hierarchical documents. It is also supported by the other parts of the system (next section).

3 Collaboration Support

In this section we present the support for users awareness, and we describe our mechanisms for controlling concurrency and access rights.

3.1 User Awareness

In addition to the advantages revision control offers to teams of programmers, our system supports collaboration through users awareness. It is based on the hierarchical organization and the revision control functionality . Users awareness is available in two ways in the system: by the *evolution graph* which is shared among all users and by presentation of *active diffs* .

Evolution Graph

When a user enters the system, the evolution graph indicates the status of the program. It depicts the programs' history by presenting the versions that have been developed so far . The user can see who is editing the program at the moment and what has happened since his last session. In figure 2, the evolution graph presented to Boris indicates that the program has been developed by three users: Boris, Ulf and Sten. Boris is currently editing the program (the editing key of its editing versions is crossed over) whereas Ulf is not. Furthermore, the graph shows that the last version developed by Boris results of a merge of a previous Boris' version and the latest Ulf 's version. Finally , the graph indicates (with the question mark) that Sten has also developed versions of the program but his site is currently unreachable. This can be the case if Sten' s site has failed or if there is a communication failure between Sten' s site and Boris' site.

Due to communication delays and communication failures, evolution graphs of the same program could be different at remote sites. This is not surprising if we consider a large scale network: the WYSIWIS *(what you see is what I see)* paradigm is not sensible in such a system. For instance, figure 3 depicts the evolution graph as it can be viewed by Sten if a communication failure has occurred between Boris' site and Sten' s site. Both graphs (figures 2 and 3) indicate that Sten is not aware of the latest version developed by Boris and vice versa. The graphs will be updated after the communication link is repaired.

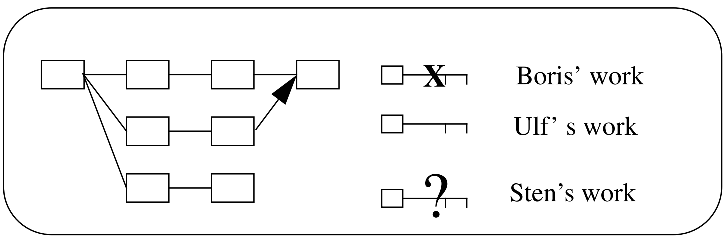

Figure 2 *The evolution graph as viewed by Boris*

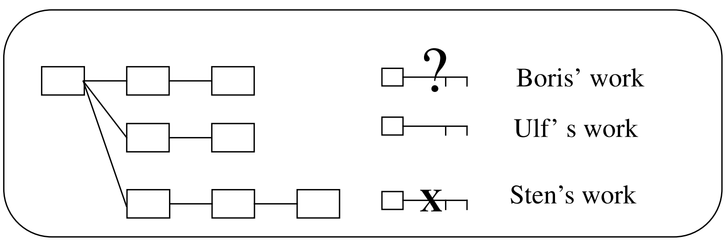

Figure 3 *The evolution graph as viewed by Sten*

Active Diffs
An active diffs shows the differences between revisions of the program. The diffs are based on the actual edit operations performed, which are stored by the revision/database handler as deltas between revisions. In this way fine-grained and accurate differences can be presented which reflect what modifications actually have been performed by the user. An example is given in figure 1. The differences between the current and the previous revisions are shown in the text window used for editing. All insertions appear as underlined text and all deletions as '-' markers. The '-' markers may be expanded interactively in order to see the actual deletions. In the same way the underlined insertions may be collapsed to '+' markers. Differences at the structure level are shown in a similar way with the icons in the window titles marked as additions, deletions, and changes within a part (or one of its subparts). The name active diffs comes from their role during development when they mark all changes from the origin version for the session and for their role in support for collaboration. Active diffs may be used for giving a 'synchronous editing' view of the program, presenting a 'what-if-I-merged' situation. The resolving of conflicts is, however, delayed until merging. The awareness provided by the active diffs during editing contribute to avoiding unintentional conflicting changes and thus unnecessary resolving of merge conflicts.

3.2 Wait-free development

Wait-free development is one of the major design principle of our framework. A user can always edit a new version of a program without waiting for a remote communication and whatever the status of the system is. For example, Boris can start editing a new version even if Sten is concurrently developing another version of the same program. Furthermore, a communication failure between Boris' site and Sten's site does not delay the work of any of them. Later on, any of them can integrate the development of the other by merging two independently developed versions. The evolution graph showed in figure 2 for example indicates that Boris has merged one of his previous versions with a version developed by Ulf. By default, a merging operation integrates all changes into the newly created version. It is then up to the developer to exclude unwanted changes and solve conflicts.

The merging operation enables to adopt an optimistic approach to concurrent development. We believe that this is typically what is required for the main phase of a program development when different users work on different parts of the same program. Each user can independently develop his part and later on merge his development with parts developed by other users. In this phase, wait-freedom is an important principle since communication delays can take arbitrarily long. As we will see in section 5, this principle excludes locking strategies for this development phase.

Wait-free development is important to achieve for most development situations. We do, however, recognize that there is a need for more rigid control over updates during particular phases of a program development, and that an environment must support also these situations. For example, an organization may decide that in the final development of a new software release only specific users may be authorized to modify the release version of the program. Maybe the integration of changes is done only after a validated testing routine and the integration work is serialized. T o support also such needs we provide facilities to manage protected version lines.

3.3 Protected version lines

A group of developers can decide that a version line (i.e a set of versions on the same line in the evolution graph) is protected. A new version on this line can only be developed by a member of that group. Figure 4 shows a development line which could represent, for example, the new releases of the program. After working independently on that program, Boris, Ulf, and Sten can create a protected development line. As showed in figure 3, the first version of this line (with black rectangles) is developed by merging the latest versions of Boris and Sten. The second version on the line could have been developed as a result of testing for example.

New versions on a protected line can only be created by specific authorized users. In our example (figure 4), these users are Boris, Ulf, and Sten. A new user cannot add a version to this line, unless the set of authorized users is modified. This modification can be done by an authorized user . In addition, version development on a protected line is performed sequentially . Ulf for example cannot develop a new version on that line when Boris is doing so. This is ensured by a (majority) locking based protocol described in section 4.

The sequencing required in the development of protected versions is not in contradiction with our wait-freedom principle. One can always develop a new version of the program on other lines. For example, Sten can start developing a new version on his own line if Boris is working on the protected line. Furthermore, a version on the protected line can be used in a merging operation on any other line.

Protected development lines are thus a feature offered by our environment, designed to support particular needs. It should not be confused with locking mechanisms inflicted by other environments as an implementation restriction.

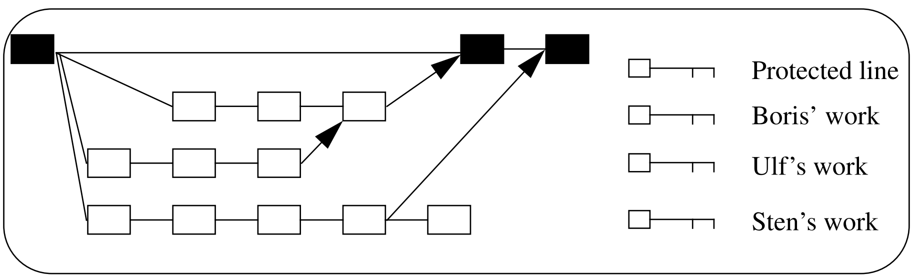

Figure 4 *An evolution graph with a protected version line*

4 Architecture

4.1 A multi-client multi-server architecture

The system is organized as a multi-clients multi-servers architecture. A client is a front-end process dedicated to one developer, for one development session. A server is a back-end process managing a database. The server stores and retrieves, in and from its database, program versions and evolution graphs. This information is represented in the database as sequences of bytes. There is one server per site connected to a group of clients. in figure 4, for example, client processes interacting with Ulf and Boris are located at the same site (at Lund) and are connected to the same server.

A client process is in charge of:

- providing an interface to the developer;
- translating the changes made by a developer into sequences of bytes and transferring them to the server, e.g. every newly created version is stored in the server's database;
- interpreting and translating the information given by the server into program versions and evolution graphs displayed to the developer.

A server process is in charge of:

- providing its clients with the information from its database;
- storing in its database the information given by the clients, and transmitting the information to other servers;
- managing access rights and locks for protected versions.

In the following, we discuss information storage, information propagation, and protected line managements.

4.2 Data storage and interpretation

Programs with their evolution graph information are stored as sequences of bytes by the servers. The hierarchical structure of a program is recognized also in the storage form and nodes that are identical in several versions are shared and thus only stored once. Each node in the hierarchy (typically representing a class or a method) is stored in full for the last versions and backwards deltas are stored in order to reconstruct older versions of the node. The calculation of deltas and the reconstruction of older versions is handled by the application (e.g. an editor) in the client. The storage scheme is therefore very general and can support different data representation forms such as text or abstract syntax trees. The storage form is also very compact since it utilizes deltas constructed by the editor and a high degree of sharing between versions. In our implementation editors use change-oriented deltas as described in [Magnusson93] and also in [Lie89].

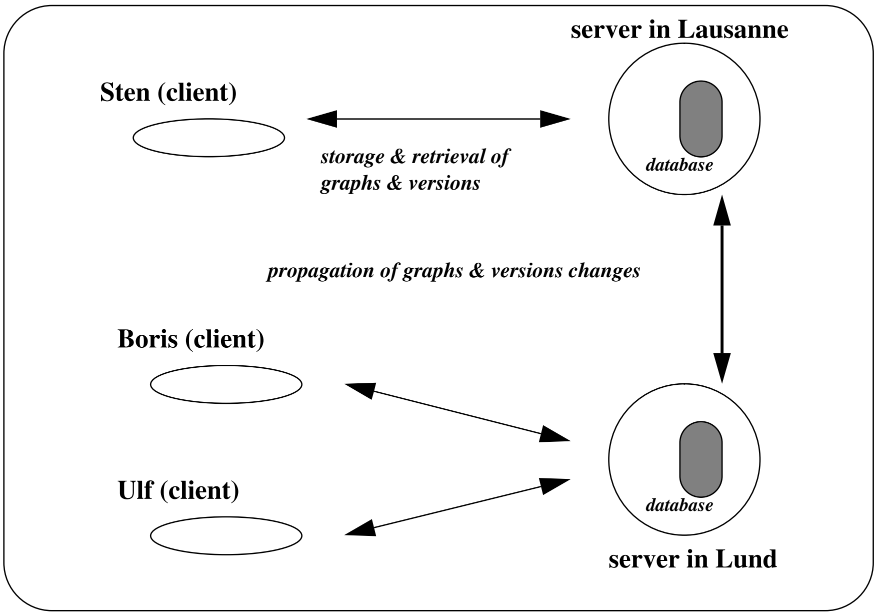

Figure 5 *A multi-clients multi-servers architecture*

All information (evolution graph, full node information, deltas and structure) is stored by the server in one Unix file using a special format (T reefile). The server reads this information from the file and communicates it to the requesting client. The client interprets this information and presets it to the user . After user changes the client sends the full node information back to the server together with the delta information it has constructed.

4.3 Accesses to program versions and evolution graphs

Starting a session

When a developer starts a development session of a program *P*, the dedicated client process requires its server *S* to retrieve an evolution graph and a version of *P*. *S* returns the version and the evolution graph of *P* from its database. If *S* has no information about *P* in its database (i.e. *P* has never been edited on *S*), then *S* creates a new empty evolution graph. Hence, following our wait-freedom principle, the developer can always start working on *P,* without waiting for any inter -servers communications (which can take an arbitrarily long time).

In parallel with the developer 's work, *S* updates the evolution graph of *P* and exchange new versions of *P* by contacting other servers. If any other server S' is editing *P* (at least one client of *S'* is editing *P*), then *S* updates its graph and retrieves new versions according to the information given by S'. If no such server exists or is unreachable, then *S* integrates the information given by different servers in which *P* has already been edited.

Changes propagation

Each server S maintains, for every program P edited on S, a set, noted *servers-editing(S,P)*, of identifiers of other servers's editing P. Whenever a client of S edits or creates a new version of P, S records the changes in its database, then propagates these changes to the members of the set *servers-editing(S,P).* The composition of this set is modified according to the communication between servers. Initially, *servers-editing(S,P)* contains only S. Whenever S communicates with another server S' where P is edited, S records S' in *servers-editing(P)*. If S learns later that P is not edited in S' anymore, S deletes S' from *servers-editing(S,P)*.

Inter-servers communication is performed using an underlying *best-effort* inter-process communication protocol. The *best-effort* semantics is provided by retransmitting messages until an acknowledgment (from the receiver) is received by the sender. After a time-out period without acknowledgment, the sender considers the receiver as *unreachable* and reports it on its evolution graphs (see Sten's site on figure 2). It is obvious that due to communication failures and unbounded communication delays, each server can have its own view of where a program is being edited.

4.4 Accesses to protected development lines

Development of versions on a protected development line is restricted by checking access rights and the locking mechanism. The former ensures that only authorized users can develop versions on a protected line whereas the latter ensures that this development is performed in a sequential way.

Checking access rights

Each server S maintains for each protected development line L_P of a program P, the set of identifiers of authorized developers for L_P. If a developer asks for editing a version on L_P, S checks if the developer belongs to that set. If not, S returns a negative answer to its client, otherwise S requires a lock on L_P by contacting other servers. The composition of the set of authorized developers' identifiers for L_P can be changed dynamically by any member of this set.

Locking

Our locking protocol is an adaptation of the *dynamic voting* protocol [Jajodia 90]. This protocol has been proved to be well suited to large scale distributed systems since it enables availability despite network partitions.

At any time there is a set of servers, noted *up-to-date(L_P)*, holding the most up to date replicas of L_P. To edit a version on L_P, a server S requires locks on all L_P replicas that are located on the servers in *up-to-date(L_P).* If a majority of locks are acquired (i.e at least |*up-to-date(L_P)*|/2 +1 servers are reachable, and their L_P replicas are not already locked), S edits the version on L_P. The composition of the set *up-to-date(L_P)* changes dynamically according to network configurations changes, new connections and disconnections:

- *Network configuration changes:* when a server S terminates editing L_P, all the servers on which a lock could not be acquired (e.g. because of a partition) are removed from *up-to-date(L_P)*. The replicas of L_P are updated on all the servers in the new *up-to-date(L_P)* set.

- *Disconnections:* when a server *S'* stops editing the program *P* (i.e no client of *S'* is displaying the evolution graph of L_P anymore), *S'* removes itself from the set *up-to-date(L_P)*. Consequently , no lock for L_P is no longer required on *S'*.
- *Connections:* If a server *S"*, which is not a member of *up-to-date(L_P),* tries to edit a version on L_P, it must also lock a majority of replicas on the server *up-to-date(L_P). At* the end of the edition, *S"* adds itself to this set.

Every server editing *P* holds a copy of the set *up-to-date(L_P),* as well as a counter representing the number of times this set has been changed. This counter enables the server to recognize the most up to date copy of the set *up-to-date(L_P).*

4.5 Evaluation

In this section we discuss the scalability of our framework with respect to memory size, response time, and fault tolerance.

Memory size

Storage overhead comes from storing deltas from old revisions and from the overhead for structure information of the tree. The structure information is fairly small, one pointer per node, which should be negligible if the content of the node has a reasonable size, like a class operation or a text paragraph, which is the intended use. Also keeping track of deltas costs one pointer to be compared with a separate file in the traditional approach costing file directory overhead and much longer access times. The deltas themselves could (in the worst case) be stored in the same format as standard diffs (as generated by the Unix utility *diff*, used by other revision control systems), so there is no reason to believe that our storage form is inherently worse than current techniques. There are in fact several aspects that indicate that the revision trees might be more memory efficient. First, traditional approaches, such as CVS, makes a full copy of all files in the checked out 'module' while our storage format share common parts. This will pay off very quickly as the number of alternatives grow . Secondly , the change-oriented deltas can turn out to be more compact than traditional line-based text-oriented diffs, especially if changes typically are small compared to the length of the source line.

In addition, versions are not replicated in all databases. A server does only store a version of a program if it has edited the program at least once. Furthermore, if a server stops editing a program, it will not need to store further versions of that program.

Response time

As we have discussed before, one of the design principle of our architecture is wait-freedom. A developer can always edit a new version of a program without requiring any remote communication, i.e the developer is never obliged to wait for any inter -server communication which could take an arbitrarily long time. It is obvious that the developer may have to wait in order to have an up to date evolution graphs and new versions, but this is done in an explicit way.

Regarding the time overhead to calculate an old revision we can note that this is done on demand on fairly small pieces of information assuming the hierarchical storage is used as intended. The time for the calculations needed should be compared to communication times between the client and the server (where the full node information and necessary deltas are sent as one message) and file access times (where our model gives essentially one read per delta while traditional diffs requires also a file to be opened and closed). We are thus confident that performance for applying deltas will not be a problem in practice even for large applications.

Access to a version on a protected line can also take an arbitrarily long time, but this is caused by agreed policies. Furthermore, when compared to a static protocol that requires to lock all replicas, our dynamic and majority-based locking protocol decreases the time required to acquire a lock and thus waiting time inflicted by technical grounds in this situation.

Fault-tolerance

The wait-freedom principle applies also to fault-tolerance. One can edit a version also when other servers are unreachable. Thanks to our multi-servers architecture, replicating evolution graphs and program versions enables to tolerate site failures.

Editing a protected version can, however, lead to blocking if a majority of servers is not reachable. Nevertheless, as we have discussed before, the dynamic changes of the set of servers to be locked adapts to disconnections and network configurations and increases fault tolerance with respect to a static lock-all protocol.

5 Related Work

5.1 Unix based environments

The traditional Unix version control systems, SCCS [AT&T] and RCS [Tichy85], both work on single files, supporting storing and retrieval of versions of files. They are using a pessimistic lock on check out approach and offer no support for hierarchical organization of data, or distribution.

CVS [Watson] is a system built on top of RCS that does address some of the problems we have discussed here. CVS enables collaborative work on *modules*, groups of files of for instance program sources. CVS provides a central *repository* which contains the most recent version of the files. A user can at any time create a personal copy of these files by checking out the files from the repository and later *commit* them into the central repository. Conflicts, i.e changes to the same source lines of a single file must be resolved manually.

The approach described in this paper shares some common strategies with CVS. The idea of *module* is matched with our hierarchical representation and both use an optimistic check out strategy with no locking, but support for merging and detection of conflicts at check in. The differences are, however, substantial. First of all, CVS does not support distribution (in the sense it is covered in this paper). The hierarchical representation used in our approach supports a finer granularity of version control since it is intended for smaller pieces of information than what is stored in a single file. Our integrated architecture gives a much better user interface with graphical presentation of the version graph, and visibility of change oriented diffs (rather than whole source lines). This means that the user has better control in a merge situation where he can easily identify and exclude non-conflicting changes he does not want to include.

In summary , the approaches share some ground. The differences seems rooted in CVS being a system built in traditional tool based Unix manner , using text files and Unix directories, while our approach builds on an integrated architecture using a structured representation.

MESSIE [Sasse93] is also built on top of RCS exploring the distribution aspect. MESSIE maintains a RCS system on a single computer , supporting remote users to perform operations via standardized email messages. MESSIE is thus a program that reads email, interprets them to perform RCS operations and emails the result, including checked out files, to the user . However , as we have discussed in the introduction, the use of locking causes problems with many users.

Our approach thus covers the attempts made in both the CVS and MESSIE systems at the same time and as described above holds potential for more sophisticated support.

5.2 Environments supporting fine-grained revision control

The Orwell system

Thomas and Johnson [Thomas 88] presented a configuration management tool for multi-person Smalltalk projects. It allows groups of Smalltalk programmers to develop code from a common class library . Orwell introduced configuration management at a fine-grained level (i.e the method level) in contrasts to previous collaborative systems which used to provide management at a large grained level (i.e the module level). In Orwell a class is stored as a number of records in a database, one for each using a aspect of a class, including one for each method.

Orwell thus acts as a common repository for a group of developers sharing a common file system. It is assumed that policies like personal responsibilities for classes, will make sure no conflicting changes are made. Orwell address the problem of multi-site distributed development by manually replicating the database, trusting the policies to guarantee conflicting changes are not made, and manual merge of the changes to the database at specific times, such as release-points. Although a valuable pragmatic tool, Orwell has a much lower ambition than discussed in this paper .

The Mjølner project

The framework we present in this paper is based on previous work on the Mjølner project [Knudsen93] and is an extension of the framework presented in [Magnusson93]. The former was also designed to control the evolution of software systems, but the grain size were coarser , complete source modules. It was designed in a centralized architecture, intended to reside on one machine or file system, without support for large scale distribution issues. These issues have lead us to reconsider the semantics of collaboration by introducing a fine-grained version control system, rebuilding the architecture so now version control is fundamental to the storage representation and thus supported by all processors in the architecture, including editors. Compared to [Magnusson93] we have here added support for a distributed multi-servers environment with features for fault-tolerance, awareness of unreachable nodes, and a dynamic locking protocol to support protected versions development.

5.3 Large scale distributed systems

Several distributed systems that deal with large scale issues have recently been described in the literature [Kistler91, Heidermann92, Pacull94]. Except for the DUPLEX system [Pacull94] which supports *dynamic* and *fined-grained* decomposition, all these systems support collaborative development at the *file* level. These systems enable disconnected operations through multi-servers architectures that tolerate node and communication failures. However, their lack of multi-versions facilities at the user level has two major drawbacks for collaborative software development. First, management of software histories must be explicitly *simulated* by the users through different files (or objects in DUPLEX). Second, avoiding locking implies an optimistic way of handling conflicts and reconciliations are performed afterwards. These reconciliations generally require external communication between users. In our system, no external communication is required since each user would have its own version (thanks to multi-versions management). Any user can decide latter to integrate the modifications done by the other users.

Feeley and Levy [Feeley 92] have proposed a distributed shared memory model named *VDOM (V ersioned Distributed Object Memory)*, which considers immutable objects and provides version management at the application level. Although the *VDOM* model was not designed with collaborative work in mind, it is very similar to our sharing model.

6 Conclusions

This paper presents an environment for collaborative OO development. It is intended for groups of developers located at different nodes of a (possibly large scale) network. It maintains programs' histories in the form of evolution graphs. Together with active diffs presenting differences between versions, evolution graphs provide adequate support for users awareness.

Our environment provides a hierarchical (OO) representation of programs. It supports aggregation of operations into classes as well as relations between classes representing subclassing and use-relations. All these dependencies are subject to revision control.

In order to enable wait-free development we have designed a multi-servers architecture. The basic communication underlying this architecture completely avoids locking as an implicit restriction. However, and in order to meet the requirement of particular development phases, we present a mechanism for sequential and private collaboration.

The current prototype of the system offers hierarchical programs representation and fine-grained revision. It is based on a client-server architecture and does not support replication on multiple servers. The environment has been tested on Simula development.

In this paper we have mainly focused on asynchronous collaboration (users are not aware of being active at exactly the same time). The need to support synchronous collaboration is recognized but not discussed in this paper (although we expect the design and protocol to be extendable to support also this situation). The kind of synchronous editing mode adequate for large scale distributed systems is however still to be defined.

Acknowledgments

References

[AT&T] SCCS - Source Code Control System. UNIX System V programmer's Guide, Prentice-Hall Inc. pp 659-700.

[Feeley 92] M. Feeley and H. Levy - Distributed Shared Memory with Versioned Objects - ACM Proceedings of the Conference on Object Oriented Programming Languages and Applications - pp. 247-262 - 1992.

[Hedin88] G. Hedin, and B. Magnusson - Direct Interactions with Abstractions- Proceedings of the European Conference on Object-Oriented Programming - Springer Verlag (LNCS 322) - Oslo, Norway, - 1988.

[Heidemann92] J. Heidemann, T. Page, R. Guy and G. Popek - Primarily Disconnected Operation: Experiences with Ficus - IEEE Proceedings of the Second Workshop on the Management of Replicated Data - Monterey California, November 1992.

[Jajodia90] S. Jajodia and D. Mutchler - Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database - ACM Transactions on Database Systems - Vol. 15, no.2, pp. 230-280, June 1990.

[Ksitler 91] J. Ksitler and M. Satyanarayanan- Disconnected operating in the Coda file system - ACM Transactions on Computer Systems, 10(1):3-25 1991.

[Knudsen93] Knudsen, J.L., Löfgren, M., Madsen, O.L., Magnusson, B., Object-Oriented Environments - The Mjølner Approach, Prentice-Hall, 1993.

[Lie89] Lie, A., Conradi, R., Didriksen T., M., Karlsson, E.-A.: Change Oriented Versioning in Software Engineering Database, ACM SIGSOFT Software Engineering Notes, Vol 17, No 7, Nov 1989.

[Magnusson93] B. Magnusson, U. Asklund and S. Minör - Fine-Grained Revision Control for Collaborative Software Development - ACM Proceedings of the Symposium on the Foundations of Software Engineering, Los Angeles, California, December 1993.

[Minör93] S. Minör and B. Magnusson - A Model for Semi-(a)Synchronous Collaborative Editing - Proceedings of the European Conference on Computer Supported Cooperative Work, Milano, Italy, September 1993.

[Pacull94] F. Pacull, A. Sandoz, and A. Schiper - Duplex: A Distributed Collaborative Editing Environment in Large Scale - ACM Proceedings of the Conference on Computer Supported Cooperative Work - Chapel Hill, North Carolina, October 1994.

[Sasse93] M. A. Sasse, M. J. Handley, S. C. Chuang - Support for Collaborative Authoring via Electronic Mail: The MESSIE Environment - in

Proceedings of the 3rd European Conference on Computer Supported Cooperative Work, Milano, Italy, September 1993.

[Thomas88] D. Thomas and K. Johnson - Orwell: A Configuration Management System For Team Programming - ACM Proceedings of the Conference on Object Oriented Programming Languages and Applications - 1988.

[Tichy85] W. Tichy - RCS - A system for Version Control. Software Practice and Experience, Vol 15(7), July 1985, pp 637-564.

[Watson] G.Watson, - CVS Tutorial, available from gay.watson@antaire.com