A Distributed Environment to support Cooperative Software Development

Adérito F. Marcos a *

aFraunhofer Institute for Computer Graphics
Wilhelminenstr. 7, 64283 Darmstadt, Germany
{marcos}@igd.fhg.de

Cooperative environments to support Software Development (CSD) have as main goal to enable several users connected over network to work together in order to develop software. They have to solve problems such as: coherence maintenance of software project through the distributed system by managing possible conflicts between local versions of each group member and promote the necessary mechanisms for the inter-group awareness and integrity.

We introduce in this paper a distributed solution to support CSD by describing our own prototype: a Computer-Supported Cooperative Work (CSCW) architecture for software development. It enables a group of developers (2 to 4), possibly located at remote places and connected over network, to develop software together. A cooperative multimedia editing environment is available during the development cycle, enclosing mechanisms of computer-conferencing (text, audio and video communications).

Keyword Codes: C.2.4, D.2.6, H.5.1
Keywords: Distributed Systems; Programming Environments; User Interfaces

1. INTRODUCTION

In the last years, the increased evolution on both network and workstation technology had enabled system designers to present more and best concrete solutions on distributed environments. This comprehends the accessibility to more cheaper and powerful resources on video-conferencing, data-transfer and multimedia capabilities which are becoming common place and used as bottom platforms to cooperative solutions.

In fact, important results in groupware and CSCW have been achieved by improving existing solutions for distributed and networked systems. The basis of these solutions had been laid in distributed operating systems theory since the 80’s (see [8,14]). However, it has been shown that the requirements in these two fields are not exactly equal and so the solutions are not completely acceptable for groupware systems. The differences occur in requirements concerning user response time, time and space distribution, data organisation, and user communication. Another problem specific to cooperative systems is the breadth of the fields and problems that are included. A third contingency is that when

*This work is partially funded by a CIENCIA (J.N.I.C.T) grant (BD/2663/93 - IA).
analysing the transition from individual work to group work, technological communication difficulties arise (if computer supported) associated with productivity falls. These problems above have lead to different developments of the techniques used before in distributed and networked systems. It is easy to come across solutions that are advantageous in a situation but not in others [16].

We deal here with the problems posed by Cooperative Software Development - a specific case of CSCW, which encompasses the questions of supporting the activities of a group of people that cooperates in order to produce a piece of software. We mean as Software Development those tasks directly related with code programming and software research implementations, which do not (expressively) use CASE (Computer-Aided Software Engineering) technology.

The work in development and maintenance of software is typically alternating between tasks involving many persons and individual assignments [9]. This development work is performed following a specific development cycle which includes activities like: conceptualisation, design and specification, editing, integration of software components, debug, test, review, and others.

Usually, during a first step, the group has to conceptualise and decide guidelines and strategies to be used during the development process, and also divide implementation tasks and responsibilities among themselves. After this phase each developer will concentrate in his own tasks, and starts properly the development process.

It has been verified that almost all the implementation work tends to be done asynchronously and more or less independently, where each developer only carries out the task(s) assigned to him. However, points exist during the development when two or more developers want to collaborate by completing together a specific goal (sharing or not software object(s)), or simply exchange opinions about details on the software project [10].

Global meetings occur when the group needs to (re)consider together the software project state (e.g. test up results or redefine strategies). Then, all the changes achieved in the system by each developer appear to fit in an unique version, turning the system to the same global state. At this point and due to the distributed characteristics of the development process, changes performed by one developer can conflict (or collide) with the changes made by the others users in the system. This represents the main difficulty of CSD processes, i.e., support of consistency. A solution must be taken from negotiations and organisational protocol strategies [11].

1.1. Consistency Support Models

Cooperative systems are related to parallel work and consequently with the management of simultaneous effects being produced. Conflicts exist when actions from two or more developers are competing for a common desired effect.

CSD encloses two kind of conflicts:

- **merging conflicts**: are those considered when changes from different developers (referring or not to the same software object) can potentially turn the system to an inconsistent state.

- **accessing conflicts**: are those related to attempts to access simultaneously the same software object.
These two types of conflicts are strictly co-related in such a way that reducing one of them increases more or less proportionally the other. When a system increases parallel work by reducing accessing conflicts, promotes the generation of different versions of software objects and consequently increases merging conflicts.

Consistency in CSD is essentially supported by the management of merging conflicts. Namely, the whole CSD process is commonly based on the management strategy adopted.

Following the above arguments we can outline two global strategies to support consistency: the first one consists on avoiding completely merging conflicts by disabling any parallel development, and the second one will do the opposite, i.e., allowing development to proceed in parallel and support more or less complex merging mechanisms.

Disabling parallel development is conceptually a simple solution with the advantage that system inconsistencies can never arise. It is based in object locking schemes (such as classical two-phase locking in traditional database systems), which guarantee that access to objects is sequentialized. However, its great disadvantage are the accessing conflicts, in way that a developer would need to wait indefinitely for locks to be released or would risk losing major investment due to rollback [13].

CSD process can also be based on a more optimistic assumptions about the frequency of merging conflicts in practice. Then, the second above strategy can be used. It allows parallel development on different local work stations and supports mechanisms to handle merging conflicts, when they occur. This is generally performed using a 3-way merge [1,3,11]. This strategy comes to be more user-friendly in such way that disables completely accessing conflicts and then each user can (theoretically) access and change objects without ”any” restriction.

However, it has been proved that a complete automatic detection of constraint violation is completely unrealistic. Hence, human users will need to be part of the process of constraint violation detection and repair. Consequently, unsolved or difficult merging conflicts can be generated (not contemplated in the current mechanisms) whose solution would imply losing investment from one of the parts. This must be avoided under condition of decrease irretrievably the group work performance.

Hybrid strategies combining both of the above schemes are also possible and desirable in the sense they can benefit from the advantages of both (see [11,13]).

1.2. Our CSD Strategy

We think CSD consistency is essentially a subset of the global cooperative strategy for coherency support through the distributed system. In effect, consistency and merging conflicts are only strictly related to tasks involving cooperative code editing, i.e., those allowing the change of the software project to several versions. Debug or test are processes that do not generate software inconsistencies, in way that they hardly will change the software project.

On the other hand, merging conflicts commonly tend to difficult or unsolved situations implying human intervention, waste of time, and above all, losing of work investment from one or all developers involved. This decreases group work integrity and performance and must be avoided. Furthermore, we have observed that rarely two developers want to edit exactly in the same point of the software component, i.e., code editing is almost all the time a private work.
Accordingly, we have adopted a strategy following two guidelines: first, prevent *merging conflicts* from arising by avoiding absolute forms of parallel work; and second, support a strong awareness within the group by allowing mechanisms for easy inter-group communications. In fact, when several developers try to edit in the same software object (but not in the same point), the object is divided among the candidates in strict reserved areas (not overlapping), for their own exclusive use. This permits a broad way of parallel work, in the sense that no different object versions are created, but the same software component is being simultaneously changed. When contentious situations appear, referring to areas or a whole object access, they can then be solved by direct human (user-user) negotiations or asking the intervention of the group's moderator.

In fact, the aim of this paper is to propose a global frame-work for CSD, instead of simply describe consistency strategy. We deal with the decisions and solutions adopted in our approach - a CSCW architecture for software development. It includes a complete software development cycle (C, C++ dependent) and is implemented over a cooperative multimedia editor (see [16]), for Unix/X-Windows Sun and SGI workstation environments. It is designed to support small groups of 2 to 4 developers working in possibly different locations (connected over a LAN or WAN (Internet and ISDN)) who want to collaborate (not compete) in order to produce a final piece of software.

The system supports awareness within the group by allowing the traditional paradigms of cooperative editing: personalised multiple cursors, WYSIWIS (What You See Is What I See), social roles, developers' identification, tele-pointing, multi-user interface, multi-user communication. The media available for communication are text, audio and video. The media available for editing in the several phases of the development cycle are text and graphics.

In this paper we firstly expose the global system architecture and algorithm adopted. Next, we explain in detail the development cycle and finally we draw our future work directions and conclusions.

2. THE SYSTEM DISTRIBUTION STRATEGY

We mean as architecture the way the cooperative system is organised in order to enable the cooperative work. This organisation concerns the distribution of the physical processes and files over the different machines where the users are located, as well as the way the communication is enabled. In context of CSD, the architecture concept also includes the organisation strategies of the software project.

On the other hand, the algorithm to support the CSD process encircles the strategies to control the information flow through the distributed system, concerning about the issues to support global coherency.

2.1. System Architecture

CSD systems require special attention due to software consistency support. Their architectures should take in consideration where and how the software project data can be physically organised according to its parts already stated as consistent and stable (commonly used as source for the current and subsequent development), and also those components being changed (potentially inconsistent).

Usually, stable software components are saved as the current global project version, and
must be independently preserved from inconsistent changes. Accordingly, the centralised architecture seems to be the adequate solution.

On the other hand, most of the times changes being generated by each developer are temporal (or volatile), in the sense they are not yet integrated in the global version. Therefore, for efficiency reasons, developers’ workspaces should maintain, more or less independently, these temporal changes in local copies.

In conclusion and following the above guidelines, we have adopted a hybrid architecture able to support a centralised control of global versions and also local structures of developers’ workspaces.

An external process (see the Global Server in Figure 1) synchronises all the developers’ actions and is also responsible for the global software project management.

All the connections between the Global Server and the developers’ processes and
between the developers’ processes themselves are made using Ethernet-LAN functions (TCP/IP) or ISDN-WAN [16].

Each user starts a process on his own workplace which establishes a communication link with the Global Server (which is started automatically if it does not exist). After this connection, the Global Server sends all the information concerning about the current software project (if it does exist) necessary for the new developer process to register itself has one more developer in the editing session.

Each developer process, supports several structures needed to sustain locally a version of the software project. Also updated copies of the editing state of the other developers are kept, which facilitates the support of awareness within the group.

The Local Server functionalities avoid the overload of the network by handling the traffic between developer process and Global Server.

The architecture also allows each developer process to connect to another workplace via a text, audio and video channels. This allows user-user communication without a explicit Global Server control and promotes the integrity and effectiveness of the group task.

The video communication uses the JPEG codec standard [7] for the video frames, and VideoPix hardware or IndigoVideo on Sun or SGI workstations respectively. For audio, we provide crossplatform usability, by using a common intermediate exchange format and realising a number of on-line bi-directional converters supporting different code formats (see [18]). Finally, the text communication follows an improved version similar to Unix/Talk feature.

2.1.1. Project Organisation

We mean as project organisation the way its involved entities are conceptually located and the kind of relation they have.

We define software project as a central head and a set of components. The head includes the makefile and all the data referring to the project design and conceptualisation. The components are pieces of text or graphics, with or without logical or hierarchical relations, consisting of parts of a whole, i.e., the project structure. These components can be:

- **units**: the traditional C program elements (modules, headers and libraries). They are in practice the software components of the project environment.

- **reports**: text or graphics files manually or automatically generated, including considerations about the development task.

- **documents**: refers to any documentation file about the software project.

- **history**: text file automatically generated, which holds a logical narrative of the developers’ actions along the development process.

The software project environment comprehends also entities such as compilers, debuggers, or any tool assisting the development process.

In fact, our considerations about consistency are strictly related with units and their management. They represent the "physical" software project being produced.

Each unit has one creator and one or more developers. At the development process, successive versions for each unit are stated stable (or consistent) and integrated in the global software project version located at the Global Server.
2.1.2. The Physical Workspace

The system physical workspace, concerning about manipulation of files, is based on a fixed tree of disc directories. It integrates several software versions, specific tables for consistency policy, compiling or debugging and files for general purposes. At start time, this tree is automatically generated (if does not exist) on the related machine or simply updated with the last project version (if the last server machine was different). Any disc access is performed using a path related to the "well-known" fixed tree, and no dependent references to the machine Network File System (NFS) are considered. This strategy allows the independence of the CSD process through different NFS.

The tree structure contemplates the principle of global and local versions, by maintaining subtrees respectively for global data (server directory), and one for each machine active in the system (local directories) (see Figure 2).

Each one of the local directories keeps the development state about the related machine. It includes for each developer, a subdirectory holding his software version. This scheme allows the Global Server to independently store the global version and also each Developer process to have locally the software state of the other users. If a developer decides to compile his or another version, then uses the contents of the respective developer entry under his own correspondent machine directory.

When the Global Server starts on a different NFS, considering the last session, then it is automatically updated by receiving the contents of the related server directory located on the previous NFS/machine. In the case of the same NFS, the updating is not needed, i.e.,

Figure 2. The physical system workspace when two Developers are active in two different machines and working in different NFS. The filled directories represent structures not used. There are two server workspaces, but only one is active, located on the machine/NFS where the first Developer started the system. Notice, that each one of the workspaces represents the structure used when two users are working in the same NFS (but in different machines).
the tree of directories is single and visible from all machines. Only one system workspace exists per NFS.

At the Developer process login time, the Global Server sends it the current global project version, even if is starting on a different NFS, in order to update its local correspondent workspace. This guarantees that late users can enter the work at the current development state.

2.2. The Cooperative Algorithm

It is easy to devise that different architectures adapt better to different algorithms. For example, centralised architectures adapt better to Client-Server algorithms and replicated architectures to Order algorithms.

We have adopted a hybrid solution, incorporating concepts from Client-Server (see [2]) and also Order algorithms (see [8]) and adapted to the system architecture (section 2.1). On one hand, there is a Global Server process to perform the management and on the other hand, by means of the Local Server, the user process gets more "intelligent" helping in the synchronisation problems.

As the development actions are performed by each of the developers, they are transmitted to the Global Server which re-transmits them to all the developers (including the one who originated it). There is no direct communication for development actions between users in the system. The Global Server receives and dispatches users' requests using
a FIFO rule. Therefore, this guarantees mutual exclusion and serialisation of the users actions. There exists no parallelism in the answer time [16].

The Global Server has locally a set of consistency tables containing the current development state of the whole software project. Access conflicts, updating requirements, or general coherency violations, are checked using the consistency tables.

By receiving the actions from the Global Server each user process executes them either by changing the development state or by outputting a result (see Figure 3). The user actions are only definitively executed after receiving the Global Server answer (agreement). During the development process, local versions are being successively updated by receiving, through the Global Server, the changes (stated consistent) from other developers. Actions such as local debug or test, even passing through the Global Server, are always executed using the contents of the local workspace. Global test or debug demands, for purposes of coherence, the integration of all the local changes in a same global version.

The Global Server process is the only one with real access to the global version being developed and each one of the user processes has only a local copy (local version). Precisely, the maintenance of the consistency of these copies and also of the global version, is the main goal of the algorithm.

3. THE DEVELOPMENT CYCLE

The development of software is necessarily a set of cyclical tasks, following a common goal, i.e., the production of a "satisfactory" package of software. These set of tasks, commonly named as the development cycle, represent in practice the system interface to the users. The Multimedia mechanisms/metaphors play here a decisive role by enhancing the expressiveness and a best manipulation of the involved information, and consequently improve the editing environment of the development cycle. The editing media used in our system are text, graphics (raster and 2D) and audio. For the inter-group communication there are three media channels: text, audio an video. The software development performed by individual or small groups of developers usually follows more or less the traditional "four-steps" cycle strategy (conceptualisation, code editing (programming), debugging and testing). Moreover, we have verified this strategy adapts quite well to cooperative environments, (see Figure 4) where certainly, we need to take into account the specific requirements of the transition from individual to group work.

Accordingly, we have defined a Cooperative Software Development Cycle as having four inter-connected phases:

- **Conceptualisation and Strategy** - the cycle's first step, where the group decides together the strategies for the project development. It represents a convergence of the whole group.

- **Editing** - during this phase each developer performs his task as decided in the conceptualisation step, which can be done alone or together. It encloses the environment of creating and changing project components.

- **Debugging** - it is the common compiling and syntax error fixing task, and can be done together with other developers' help.
Figure 4. The *traditional* software development cycle performed by an individual developer (left), and the cooperation role in the cycle (right).

- **Testing** - after producing an executable program the developer should test the results, with or without the group collaboration.

Tasks as review or inspections, are not properly independent development steps, and consequently are included in the editing environment.

**Social Roles**

As the CSD work is a task performed by a group of people, it must include some kind of inter-coordination in order to promote the work performance. One response to the problem is the definition of social roles. The social roles we have adopted are *Moderator* - chairs the session, *Developer* - one of the participants in the team that actively contributes on the development process, *Reviewer/Inspector* - does not participate directly on the development work, but contributes to the CSD process by commenting on the software project. This can be done informally or executing a well defined review/inspection task.

The social roles of each of the participants in the meeting is defined at the login time or during the *Conceptualisation and Strategy* phase.

Developers can also assume another role in the system, i.e., as *expert*. The *expert* is any active developer in the system and who has some understanding on specific areas of the project. The developers can ask him opinions about conceptual things or send him an error report while looking for help. An *expert* affects a special importance in the *Debug* phase, where difficult syntax errors can be more easily solved with colleagues’ help.

**3.1. Conceptualisation and Strategy**

We define *Conceptualisation and Strategy* as the first project step where the group members decide together things as: conceptual guidelines, global resolution strategies and tasks distribution. From here, the development responsibilities are divided among
the team members and each one should promote the work in his specific part.

We provide Conceptualisation and Strategy in our system by a set of brainstorming tools that includes:

- **brainstorming zone** - is a shared drawing area where all the users can sketch simultaneously. Each one sees instantaneously the inputs of the others;

- communication channels (video, audio and text);

The brainstorming zone plays here the role as a traditional shared "white board" often used in face-to-face meetings, which holds informal schemes generated during the discussion. Therefore, each group member can add (draw) new details while looking for attention to a specific idea. This information can be stored as part of a report and recovered subsequently.

In the brainstorming zone all the editing actions of each user occur with his personalised cursor, which allows the individual recognition.

The conferencing process of this phase is coordinated by using the established group members social roles.

3.2. Editing

As we referred before, the editing phase encloses the environment to manipulate project components. A component is a piece of text or graphics that can be edited by one or more developers. Two specific cooperative editors (text and graphics) support the components editing. They include the WYSIWIS and multiple cursors paradigms. The system also presents a global environment where the user can edit at same time several components and organise them (see Figure 6 (top-left window)).

The manipulation of units as software objects represents the main goal of the editing phase, and therefore must be considered under the global consistency strategy adopted. Therefore, only two editing modes are permitted:

- **turn taking** - only one user can edit at a given time, i.e., the unit is locked.

- **split & combine** - allows simultaneous user editing. Even so, each user has a reserved area for his own strict use and cannot interfere in the other users' parts. Anyway he is completely aware of the others actions (see Figure 5).

The turn taking mode concerns about the exclusive use of units. In fact, this mode encircles the possibility that other users have to wait indefinitely for the lock to be released. In such case, direct user-user communications can be used to find out an agreement, or as a last resource, ask for the Moderator intervention who has authority to break the lock and free the unit.

Split & combine mode demands a merge mechanism in order to integrate the various changes. As no absolute parallel development exists and also no physical "collisions" are generated, consequently the merge function is reduced simply to a copy of the changed areas to the original unit (see Figure 5). Human intervention can always be used to repair difficult merge cases, usually coming from logical dependencies between changed parts from different users.
When the developer leaves a whole unit or simply a reserved area, the related changes are sent to the global server, integrated in the global software version and finally distributed to all the local versions of the other developers in the system.

**Code Review/Inspection**

Code Review is the inspection and analysis of source code units by developers who are knowledgeable in the application domain and programming environment. Code reviewers analyse individually the unit to be inspected, looking for coding errors, portability problems, violations of coding standards, etc. Thus, review is mainly based on the commenting on the software source code. These comments are fragments of information referring to a certain piece of text in the code, and express considerations about this related area.

Our system supports public comments and private annotations. They can contain textual, graphical and voice information. Their function is based on the traditional hypertext paradigm - they can be accessed by following a link when clicking on the area. The comments are immediately distributed after their creation to all users. They are then common knowledge to the group and can be further edited by anyone [15]. Annotations are appropriate to privately generated ideas, which may subsequently be communicated to the group.
3.3. Debugging

We define debug as compiling and syntax bugs fixing. Thus, in most of the cases, it is a closed individual task. The cooperation can appear in all the situations where a developer needs to ask for someone else's opinion about details such as errors or software characteristics. Our environment takes that into account by permitting developers to send errors or the whole debugging environment such as windows or units to an expert. The expert can be any of the other users. Therefore, a sub-group or all the group can follow out and help in the debugging process of a member.

The cooperation occurs also in the manipulation of a specific debugger tool (the current prototype version supports only the Unix debugger dbx). Several developers can perform inputs and receive outputs in a shared interface, following the paradigm of WYSIWIS. This aims to maintain a "on line" discussion over the related local or global version being debugged. The input/output control strategy of the running debugger is similar to the one approved in the testing phase. The only difference resides on the related executable file.

3.4. Testing

Testing is essentially the assessment of the current results achieved on the software project. This process can be made locally or globally. In the first case, represents a private test where the developer uses his local project version in way to evaluate his own changes. A global test comes when the group decides to integrate all the local versions and observe the whole aspect of the project. Then, one of the machines is adopted to hold the running process, usually belonging to the developer who requested the global test.

A global test involves a shared interface and the control of the several input/outputs coming from the group. The principle of WYSIWIS must be followed, permitting as much as possible the awareness within the group. The input flow can be controlled by two modes:

- **token-ring**: only one developer (who has the token) can perform input.
- **free-for-all**: all the group members can enter inputs.

The token-ring mode, refers to the classical token strategy, i.e., only one developer has the turn to perform input. The turn can be given or lost to another user.

When testing with free-for-all mode, all the developers can perform inputs, and the Global Server takes the responsibility to solve possible conflicts. In fact, a strategy of global stamps is used, i.e., each developer process has locally a total sequence order, referring to the inputs accepted by the Global Server. This sequence is the same in the whole system, and reflects the inputs serialisation already performed.

An input is accepted only if it follows the global time stamp, otherwise it is discarded. When the Global Server process accepts an input, it informs all the developers (including the owner) which was the accepted input, who is the user owner, and the current value of the global time stamp.

This scheme permits to sustain coherently the sequence of input/outputs through the system.
Figure 6. An overview of the CSD system. The units organisation environment (top left), with two open unit editors (bottom right and left), "expert" window (bottom center), the debug window (top right), and also the video channel window (top right).

4. FUTUR WORK

Even though the prototype offers a complete Software Development Cycle, we still have a number of research directions underway.

One of these directions that we want to follow is the so-called version management, which permits simultaneous editing of different (time/space) versions of units and consequently a best evaluation of the whole software project. This represents one of the most important points to be completed as a future work.

However, we are already doing the first steps in what we call our priority goal - a generic multimedia cooperative environment able to integrate non-cooperative applications. It comprehends a complete computer-conferencing top level structure which allows people to share their own tools and environments and enforces a best continuity of the cooperative
framework.

Another important point concerns about the support of remote software packages demonstrations and consulting. It has to combine: forms of logical representation of editing actions, multimedia, hyper-organisation of objects and transfer mechanisms over network, to enhance an interactive generation of presentation sessions. Also we have under consideration the use of ATM and "mobile" technology for communication proposes.

5. CONCLUSION

In the last years, more and powerful cooperative systems have been developed following the evolution on technology and performance of both network and workstations platforms.

One of group' activities being supported cooperatively is Software Development. Software development is usually not pursued by only one person. Most of the times it involves the work of software designers, programmers, engineers, end-users, reviewers, etc., not located in the same geographic place, i.e., expertise is not located in just one place. Therefore, the development of software products can be dependent on the cooperation of many people.

In this paper we have introduced a global distributed environment to support CSD. An architecture and algorithm to support the CSD task, were explained, and also the development cycle. These have enclosed issues such as: the way the software project is organised through the distributed system, referring the cases of working in the same or in different NFS, or which kind of strategy was used to perform the software development process in its several steps.

Acknowledgements

We thank Prof. J. Encarnação for the opportunities given, Dr. Ch. Hornung and A. Santos for the useful suggestions and discussions, and finally the student J. Rossa for the help in the implementation.

REFERENCES

5. Ishii H., Kobayashi M. "Integration of Inter-personal Space and Shared Workspace: ClearBoard Design and Experiments", in Proc. of CSCW'92,Toronto (1992).
13. Pu C., Kaiser G., and Hutchinson N., "Split transactions for open-ended activities", in Proc. of 14th VLDB (1988), ACM.