SUMMARY
This paper presents a prototype computer supported cooperative work (CSCW) environment that supports the engineering design process. In particular, this environment provides collaborative tools that support computer-aided design (CAD) and computer-aided software engineering (CASE). This paper takes an high level look at the system as a whole and then focuses on the method used to implement the CASE support. It addresses how group members can simultaneously work on coding a program and then integrate their work to form a single program inside this environment. © 1998 John Wiley & Sons, Ltd.

key words: Computer Supported Cooperative Work (CSCW); distributed CASE; collaborative programming; groupware

INTRODUCTION
The computer has become an invaluable tool throughout the world and has greatly enhanced productivity. However, computer systems, especially software systems, have long been designed as stand-alone entities that target making individuals more productive. Yet it is common for groups of individuals to collaborate on a project. These computer systems support the individual work done by group members, but provide only token support for group interaction. The details of this interaction are left for the users to organize. In the foreseeable future, a new generation of distributed computer software geared for supporting collaborative work is expected to flourish. The success of this new generation of software is now almost certain, because allowing multiple users to contribute to the same piece of work anywhere and at any time is a concept welcomed by users. Moreover, the fast advances in computer networks have laid solid technological foundations for making this new generation of distributed software systems possible.

Contract Grant Sponsor: U.S. Department of Commerce, National Institute of Standards and Technology; contract grant number: 70NANB5H0117.

CCC 0038–0644/98/060657–23$17.50
© 1998 John Wiley & Sons, Ltd.

Received 2 May 1997
Revised 18 November 1997
Accepted 3 December 1997
Systems of this nature are known as Computer Supported Cooperative Work (CSCW) systems. Numerous commercial and experimental systems have been designed, including a system called DCWA for collaborative writing applications. Work in the area of CSCW research and development has led to the realization of the possibility and the potential significance of CSCW systems in the engineering design processes. An engineering design process normally involves a great deal of discussion, cooperation, as well as individual effort.

The system presented here is a prototype CSCW environment to support engineering design, specifically, a distributed environment that combines Computer Aided Design (CAD) and Computer Aided Software Engineering (CASE). In addition to supporting the engineering design processes, this system also combines design and simulation in a distributed environment. Computer simulation is an important tool in almost all engineering areas for evaluating the quality of designs. Appropriate use of computer simulation requires training on the designer’s part, because a gap exists between a design and its simulation – mathematical modeling and computer programming are, therefore, needed to bridge the gap. For a large design involving a group of engineers, the modeling process requires frequent discussions, clarifications and redesigns. The simulation programming phase will generate additional such needs. To enhance productivity in such an environment, a CSCW tool which can provide real-time distributed conferencing, distributed CAD and CASE capabilities, and automatic document and simulation code integration is much needed.

CSCW SYSTEMS

The collection of hardware and software supporting CSCW and the like is also known as groupware in the literature. Centralized groupware (i.e. either large mainframes or a PC-based system) was explored in the past as a means of achieving office automation. Only recently has the focus been shifted to the development of networked distributed groupware. Many problems remain open in a distributed environment, the most salient ones being group management, multicasting, distributed database management and the user interface. These problems are mutually dependent and cannot be dealt with in isolation. In this section, we discuss some well-known CSCW systems.

Message-based systems

Message-based systems such as electronic mail and bulletin board systems are the most primitive CSCW tools, yet they are still a powerful way of achieving collaborative work. Sharples showed in his research how email can be utilized among groups of researchers and students to jointly author an academic paper. However, he stated that email alone cannot provide social/cognitive cues like the status of senders and receivers, the social context, and the writer’s intentions. It also lacks control over the workflow. It is unable to handle things like group negotiation and workspace partition. Recently, several tools have tried to compensate for this deficiency by adding some of these features. For example, some systems have added group awareness tools, navigation tools, access control tools and version control tools.

COSMOS attempts to enhance the basic email system by allowing greater user customization capabilities. Each user is able to configure his message structure,
a computer supported prototype cooperative work environment

communication structure and role. A particular piece of information can then be viewed based on its configuration. Although the network problems kept the system from being successfully implemented, the idea showed that more structure in email systems could be very beneficial to a collaborative work environment.

Active mail allows users to interact through the use of persistent and interactive connections. Instead of sending messages directly to remote users, the message is routed to the remote user’s Active Folder where persistent connections are maintained. Message contents will first be updated in this folder and then delivered to the receivers. Since this folder’s contents are not destroyed until the user exits, messages can be viewed at any time. The system uses a text conversation tool to facilitate collaborative writing, revision control management, and meeting scheduling.

In MESSE a set of embedded commands in the email message’s body is automatically interpreted to generate the appropriate output for the receivers. Shared files and version control utilities help users submit, read, edit, delete and list files or documents in the collaborative authoring environment.

Coordination Systems

With coordination systems users can not only perform their work, but also observe and coordinate the activities of other users. This capability allows a user to organize, plan and manage projects for an entire team. However, this type of coordination is a more difficult problem in distributed environments.

The COOPerator supports asynchronous distributed collaborative writing. It also provides a group agenda for the organization and planning of group meetings. Co-authoring is accomplished through the use of annotations and email. Documents are organized in a tree-like structure, and access to a particular section of a document is coordinated by predefined social roles.

MUCH is a hypertext-based collaborative work system. The organization of the documents follows the semantic net principle. By utilizing the semantic net, different arrangements of social/cognitive information may be obtained. For example, users may choose a people’s view, a group view, an area view, a journal view, and a status view of the document. The system also helps users to plan and organize group work by reflecting the group’s visions and objectives.

Collaborative Authoring Systems

In the field of collaborative editing, the collaboration among co-authors can be classified into two categories: shared mind and division of labor. In the first category, multiple authors edit the same file at their own locations and one coordinator integrates the contributions. In the latter category, a task is divided into mutually independent sub-tasks for co-authors. These classifications are in parallel with the notions of synchronous versus asynchronous collaborations. In the synchronous approach, users work at the same time on the same file, while in the asynchronous approach, users may not work at the same time.

Synchronous systems

GROVE is a real-time text editor that allows shared items to be viewed and edited through the outline window. The outline window not only provides the shared
contents in an outline, but also groups information such as session members and item read/write permissions. Users can edit outline items and change permission rights. However, GROVE does not have an integrated voice tool, which means that any audio conversation must be achieved separately. It also lacks concurrency control. Modifications made simultaneously by multiple users cannot be integrated.

ShrEdit\textsuperscript{13} allows users to independently customize and view any part of a shared workspace. It provides exclusive locking mechanisms to control synchronous access on the shared object. The lock can be applied to various granularity levels such as sentence, word, character, etc. It is useable with other word processing applications, but it too does not support an integrated audio tool.

SASSE\textsuperscript{14} provides awareness information through the use of color, a shared scroll bar, audio cues, a document overview and a tracking mode. It also provides a selective locking mechanism so that the user can lock sections of text in the document. It supports both synchronous and asynchronous collaborative work modes to fit different collaboration styles.

The well-known rBIS\textsuperscript{23} and SEPIA\textsuperscript{24} are synchronous hypertext systems providing various modes, or levels, of collaboration. In the independent mode, users may work on their own tasks without interfering with each other. In the loosely-coupled mode, users may share certain public information while working on their own tasks. In the tightly-coupled mode, users will share the same view, but resources (e.g. mouse and file) are strictly controlled to avoid conflicts. An interesting observation\textsuperscript{15} is that the success of collaborative editing depends heavily upon the mutual confidence and trust among team members. Mutual trust among authors should increase when communication facilities are conveniently available.

**Asynchronous systems**

MILO\textsuperscript{16} supports distributed asynchronous collaborative writing by creating structured documents in a graphical tree-like representation. A user can define any form of hierarchical structure. The nodes in the tree are referred to as *notes*. Each note stores social information such as creator, creation time and creator along with its document section. Notes also have permissions set to control user access. To generate a completed document, the notes of the tree are merged.

Alliance\textsuperscript{17} and Duplex\textsuperscript{18} provide for a similar scheme of structuring documents. However, in Alliance and Duplex, the structure is more formalized. Instead of allowing any hierarchical structure, a structure is a set of predefined logical units called *fragments* and a set of logical relations between these units. The size of a fragment is variable, and is automatically divided depending upon the user’s choice and role. To maintain document consistency and provide group awareness, Alliance utilizes a version control tool and a delayed awareness tool with a user validation mechanism. In Duplex, each fragment is stored in a replicated kernel, and the concurrency in the kernel is achieved through a serialization. A user negotiation mechanism is also provided to handle conflicts.

PREP\textsuperscript{19} is a text differencing system. PREP’s *flexible diff* compares and tracks differences in two versions edited by different users. Difference reports are annotated into an adjacent editor column, and can be performed at various levels of granularity. PREP also supports an enhanced mechanism for supporting social roles and cognitive activities by specifying responsibilities and patterns of interaction. The system also
supports changes in the collaboration mode in each cognitive phase, and specific cognitive activities are defined by including jotting, drawing, writing and gestures.

Besides the synchronous/asynchronous classification, a collaborative writing system can be characterized by the degree of structure imposed on the document. The degree of document structure ranges from non-structured applications like simple email to very structured applications like CASE/CAD tools. Tammaro and Mosier investigated the benefits of well-defined and formatted document structures in their research. They stated that currently existing collaborative writing tools lack the ability to express complex social, cognitive, intellectual and procedural work behavior. They further indicate that a predefined format can eliminate much of these complexities.

**Group conferencing systems**

Synchronous group conferencing (i.e. all participants are active and instantly aware of events that happen in the system) has been accomplished through multimedia communication channels. The MERMAID and the SPIN systems are prototypes that provide real-time conferencing environments for geographically distributed participants by using synchronous textual, audio and video communications. In addition to providing conferencing service, the COGNOTER system also collects and organizes ideas from its participants for discussion.

**Collaborative graphics drawing systems**

One effort of allowing multiple users to perform concurrent graphical operations in a distributed, networked environment is the TeamWorkStation (TWS). Its design is based on a real-time, openly shared workspace which every member can see, point to, and draw on simultaneously by using a variety of drawing tools. TWS provides a translucent overlay of individual workspace images. This overlay is accomplished by superimposing images created at different sites. Ensemble is an X-W indow based, object-oriented graphics editor. Concurrency is controlled by the so-called implicit locking. When a user loads a file, he may create an object or select an object to edit. A lock is then placed on that object. Although a user is able to see other users’ pointers on screen, other users’ editing sessions cannot be seen.

**Collaborative programming**

Although version control can be applied to any type of file that undergoes revisions, it is commonly associated with application development due to the numerous iterations involved in that process. Typically, it is associated with a group collaborating on a programming project. The more advanced systems are sometimes classified as Software Configuration Management (SCM) systems, but their primary purpose is still version control. Although the CSCW tool presented in this paper does not implement version control, it does share some of the basic ideas.

The main purpose of a version control system is to track the history of revisions made to files. Files are normally stored in a central location or a repository. From this location, users can check out a file, make revisions to it, and then put it back.
While the file is out, the file is locked so other users who try to check it out will only get a read only version. When the file is returned, it is assigned a new identifier. In this way, the revision history is maintained and any version of the file can be retrieved. Most systems also have some method to compare versions to see what changes have been made.26

Most modern systems take advantage of new technologies to increase their functionality. For example, MKS Source Integrity and StarTeam can both use the World Wide Web to distribute files.26 The Revision Control Engine (RCE),27 the successor to the popular Unix based RCS, is a powerful version control system by itself, but it also provides a programmable interface. This API provides over 100 functions that allow a user to build version control into an application. It is available for all the popular operating systems, and can be integrated with many development environments such as Microsoft Visual C++.27

ClearCase from Atria is also classified as an SCM system. Although its primary function is version control, it also helps developers manage the various builds of their applications. Like RCE, it is available on most major operating systems, and integrates with popular development tools. Add-on products from Atria provide additional functionality. ClearCase Multisite allows geographically separated teams to share files by controlled replication of the repository. ClearGuide adds project management to the SCM system, and allows a manager to assign tasks to individuals and track the progress.28

Though these tools are valuable, they mainly keep team members from destroying each others’ work, and do not facilitate cooperation between members. With the ClearGuide add-on, the ClearCase tool takes a step toward the type of support needed for the environment described in the introduction. However, it still lacks real-time collaboration.

THE DISTRIBUTED COLLABORATIVE WRITING AID

The environment is primarily based on an earlier CSCW prototype, called DCWA (Distributed Collaborative Writing Aid).4 The original purpose of DCWA is to support a group of users in the process of composing a document, e.g. a paper, a manual, a program, etc. It provides collaboration through the system’s groups services facility. A distributed multimedia database organizes both textual and graphical information according to their organizational (structural) relationships, as well as their semantic relationship. Users can maneuver in the document logically using a semantic network that is overlaid on the organization structure of the document. The user interface helps the user view his work and other group members’ working areas with either WYSIWIS (What You See Is What I See) or WYSINWIS (What You See Is Not What I See) style. The WYSIWIS style is typical for synchronous cooperation, where members must see changes instantly, while the WYSINWIS style is important in case information must be tailored before being presented to a remote user. More details of the design and implementation of DCWA can be found in Chang and Murphy.4 Although DCWA has laid the groundwork, the system has been significantly modified to provide the necessary CAD and CASE functionality. It is implemented in many standard programming facilities, including Solaris 2.5, the TCP/UDP/IP protocols, the OSF’s Motif toolkit set, the X-library drawing
primitives, X imaging Library, and C++. Major features of the new environment include:

(a) **Collaborative CAD tool with an online coordinator**: in a distributed collaborative CAD application, multiple facets of a design must be integrated. Designers must conduct discussions on how the integration should be accomplished. Although many prototypes have been developed for collaborative drawing, they lack this online coordination handling capability. For example, most systems provide a superimposing capability for multiple drawings; however, essential CAD capabilities such as binding, attachment and proportional scaling of objects are not provided. In DCWA, graphical objects are represented and handled in terms of object primitives, e.g. circle, rectangle, etc. This makes specifying and maintaining object relationships possible.

(b) **Integrated CAD and CASE functionality**: traditional collaborative writing systems can be used as a group programming environment. Group members are able to view other members’ program segments. However, integrating program segments into a complete program and compiling the program are left for the members. This situation can be overcome if appropriate CASE support is provided. In addition, developing software simulation can be viewed as a redesign of the same engineering product using computer languages. Therefore, there is a need to avoid repetition of effort in these two phases—product design and simulation design. They should be done side-by-side using the same environment. The design and simulation capabilities provide a test bed for this unique combination.

(c) **Document integration from contributions**: the design result of a collaboration team is represented in a tree-like structure, where the leaf nodes include the actual contents. A leaf node may contain text description, graphic design, image or software code. When a document is desired (e.g. report or manual), a tool is provided to generate an integrated, customized document. It allows the user to select and organize the desired contents to meet the desired needs. The document is represented in the HTML format, which allows text, graphics and images. The user can use any web browser to view the HTML document.

The central idea behind the system is that a collaborative project can be represented as a tree. When using the system, group members view this tree from their own workstation. The tree, also known as the logical view (Figure 1), is located in DCWA’s main window. From the logical view, the group can manipulate the tree structure to organize the project. The leaf nodes of the tree contain the data of the project, which can be text, code, graphics or images. Leaf node contents can be edited by any group member, who clicks on the appropriate node. Color changes in the node indicate to all members when a node is being edited and is locked. Node creation and tree structuring can be done by simply clicking on the desired node, and a node editing manual will pop out. At the start of the collaboration, the root is the only node that exists.

Through this system, group members can work on different parts of the project simultaneously. DCWA does allow multiple group members to view a locked node’s contents, which means that the group members can discuss that part of project. Communication between members is facilitated by DCWA’s video conferencing system.
DISTRIBUTED SOFTWARE DEVELOPMENT

One of DCWA’s objectives is to allow a group to collaborate on software development. To facilitate this functionality, some capabilities similar to an SCM system need to be built into DCWA. However, functionality and flexibility is limited to the capabilities of DCWA itself. The intention is not to provide a sophisticated version control mechanism, but rather to give users a single place in which to perform all the steps of the typical software engineering development process. Text and graphic nodes provide the means to generate documents, graphs, charts and other deliverables for the analysis and design phases. A new type of node, the simulation or code node, is needed to provide the means for software implementation and/or prototyping. Once these code nodes have been created, DCWA integrates them together and generates an executable (Figure 2).

Three basic problems must be addressed by any mechanism that attempts to implement this feature. First, there must be a method to unite the code segments in a proper sequence. DCWA allows much freedom when creating a tree structure for a collaboration. There is therefore no guarantee that the tree structure presented in
the logical view will have the code segments in the necessary order. Furthermore, simply ‘gluing’ segments together would make it difficult to implement flow control in the program. A second problem that must be investigated is naming conventions. Any time several individuals work on a programming project, there is a chance that there will be conflicts with variable names. The same name could easily end up being used to identify different variables. Likewise, it is possible that a single variable may be identified by several different names. The solution must be able to handle these types of situation. The third problem is that of global data structures. It is likely that there will be some data structure unique to the program, and that this structure will be used in several of the code nodes. It would be undesirable to force users to define the structure in every node that uses it. So there should be a way that the structure can be defined once and that all of the code nodes can use it.

THE APPROACH

The basic approach used by DCWA is to force users to adhere to some common practices when developing software in C++. Normally when developing a large program, code is broken down into several files. The code files are then combined through the use of `#include` directives in the actual code files and of some `make` utility. This technique is used for several reasons: it is simply easier to edit and debug smaller files; it would be difficult for several people to edit a single file; splitting a program into several files saves on compiler time. If the program was contained in a single file, the whole program would have to be recompiled whenever a change was made. By splitting the program, only the altered file must be compiled.34

The first part of the approach is to define precisely what is allowed in code nodes. Instead of simply allowing any fragment of code, DCWA specifies that the content of code nodes should be black box functions, i.e. the code segments in the nodes should be complete, self-contained functions with a single, distinct point of entry for the data it needs and a single, distinct point where it places its output. Complete means that the node contains the whole function and not a piece of it. Self-contained
indicates the node needs only the appropriate data to operate correctly, and will not make calls to other code nodes. However, a code node can call functions/procedures that are referenced in the `#include` statements specified in the code node. Requiring data to enter and leave the function at a well defined point essentially means that users should attempt to avoid global variables if possible. The only information a user of the function needs to know is its name, its parameters, and its return type, if any. Anyone using the function should not need an intimate knowledge of how the function was implemented.

By enforcing this constraint on the content in code nodes, the nodes essentially become building blocks. After all the building blocks have been completed, a single user can organize them and create the desired program. This integration code is stored in the collaboration’s ‘master’ file. DCWA automatically creates the master file template and places the necessary code (e.g. include statements) to allow the code node functions to be used. It is normally a single user’s responsibility to write the main program with any necessary flow control and output. In this manner, the problem of integration and ordering is solved. The problem of name conventions is also solved since the variables in each code node will have scope only in that node.

To address the problem of shared data structures, DCWA uses a ‘global’ file for each collaboration. In the global file, users can define any data structures to which multiple code nodes and the master file need access. DCWA then performs the necessary processing to make these definitions available. This file can also be used to place `#INCLUDE` and/or `#DEFINE` directives that may be needed by several nodes. It is important to note that this file cannot be used to define global variables, and that this approach encourages users to avoid their use. It is still possible to use global variables, but they must first be defined in the master file. The user must then redefine the global variables in the code nodes with the `EXTERN` command.

To produce the executable, DCWA produces a ‘make’ file. The make file contains all the required information needed to compile the program. The user can edit the make file to make any customizations needed. For example, the user may need to define the appropriate directories for any libraries that are used. At the user’s request, DCWA runs the make file through the UNIX make utility to create the executable code.

![Figure 3. Creating a code node](image)
In the actual implementation, the first step in creating a program in the DCWA environment is to create code nodes. They are created the same as other nodes. The user right-clicks on a place-holder node and selects the node type. When text is selected, the node becomes a text node and an empty text file is created. Likewise, when simulation is selected, the node becomes a code node. However, when a code node is created, DCWA creates three files: \texttt{NODE.cc}, \texttt{NODE.test.cc}, and \texttt{NODE.h}, where \texttt{NODE} is what the user named the node.

The white boxes in Figure 3 indicate that the files can be edited by the user while the gray box cannot. The editable code files are accessed like the other types of node files. The user simply left-clicks on the desired node, and the file is loaded into the appropriate editor. One slight difference is that, since two editable files are associated with a code node, two text editors are loaded so the user can work on both files. DCWA will make the appropriate locks so other users can view the file but not edit it.

Neither of the editable files are empty upon creation, but are actually worksheets, or templates, that the user must fill out. Various markers in the template are then sought out and replaced to customize the template for the node and collaboration. For example, one such marker is \texttt{<GLOBAL\_HEADER\_FILE>}. This marker is replaced by the collaboration's global header file (e.g. \texttt{COLLAB\_global.h}). This altered template is then written to the newly created node file.

\texttt{NODE.cc} is the main node file, and is where the user will define the black box function for the node. The default template for a \texttt{NODE.cc} file follows.

```cpp
#include "<GLOBAL\_HEADER\_FILE>"
/** REPLACE THIS LINE WITH THE PROTOTYPE FOR YOUR FUNCTION!!! **/
/**
 * PLACE DESCRIPTIVE COMMENTS ABOUT YOUR FUNCTION
 * AND HOW TO CALL IT PROPERLY HERE!!!
 */
/** PLEASE DO NOT REMOVE OR ALTER THIS LINE!!! **/
******
* PLACE CODE FOR YOUR FUNCTION HERE
*****

/*******************************************************************************************/
* NOTE TO THE USER:
* For the program to function properly it is
* important to place the prototype for your
* function above the 'DO NOT REMOVE' line and
* to NOT remove or alter the 'DO NOT REMOVE'
* line.
* 
* Note that the comments you place in the area
* marked for comments will be copied to the
* master file to help the author of that file
* connect the various functions together.
*******************************************************************************************/
The template generated for this file has three purposes. First, the template inserts a special line that divides the file into two sections. The marker states ‘PLEASE DO NOT REMOVE…’. This file marker is used by DCWA when processing the node file and, as the line suggests, should not be altered by the user. Second, it has several comments explaining how code should be placed in the file. The comments inform the user that he should place a prototype for the node’s function and some descriptive comments of that function in the section above the file marker. They also instruct the user to place the code for the function below the file marker. Third, it includes the global header file so that global definitions may be used in the function. When the user saves the node file, DCWA reads the file starting with the line after the global include statement and ending with the line before the file marker. In this manner the function prototype and comments will be read. DCWA places this information in the node header file (NODE.h). The node header file is used when updating the master file and will be explained later.

The purpose of NODE.test.cc is to allow a code node’s author to test and debug the function before it is incorporated into the master file. The template for this file contains necessary statements to make the global definitions and the node function available for use in NODE.test.cc. The default template for a NODE.test.cc file follows.

```c
#include "GLOBAL_HEADER_FILE >"
#include "NODE_HEADER_FILE >"

main()
{
}
```

It also contains an empty main() procedure. The user can write a simple program in this file that tests the function.

Once the code nodes have been completed, the next step is to combine them. This process begins when the collaboration’s first code node is created. This first code node indicates the users will be using this collaboration for software development. Therefore DCWA creates four files: COLLAB.global.cc, COLLAB.global.h, COLLAB.master.cc and COLLAB.Makefile, where COLLAB is the name of the collaboration.

In Figure 4, the white boxes indicate files that the user can edit, while the gray
a computer supported prototype cooperative work environment

box is a file maintained and used solely by DCWA. As in the case of the node files, these files are not empty. They also contain default templates that DCWA customizes for the current collaboration. To allow the user access to the editable files shown in Figure 4, nodes representing them are displayed along with the logical view. These nodes are independent of the tree and are located under the root node. To access one of these files, left-click on one of the independent nodes. DCWA will then load the appropriate file into the text editor and make the necessary locks on the file.

Like the NODE.cc file, the global file is divided by a file marker and contains several comments instructing the user on how the code should be placed in the file. The comments above the file marker instruct the user to define the various data structures that are needed by multiple nodes. The lower half of the global file was designed for implementing C++ classes. When defining a C++ class, it is possible to include member functions for that class. In this file, users can define the member function prototypes in the upper section and implement the functions in the lower. By doing this, the member functions will not have to be compiled every time the system is compiled. During compilation, object code is created from the global.cc file and linked with the other files. The default template for COLLAB.global.cc follows.

/*
  DEFINE GLOBAL CLASSES, STRUCTURES, ETC. IN
  THIS AREA BEFORE THE 'DO NOT REMOVE' LINE. THIS IS
  ALSO A GOOD PLACE TO PUT #include STATEMENTS
  NEEDED BY MULTIPLE NODES
 */

/** PLEASE DO NOT REMOVE OR ALTER THIS LINE!!! **/

/****************************
 * DEFINE MEMBER FUNCTIONS FOR YOUR CLASSES HERE...BELOW
 * THE 'DO NOT REMOVE' LINE.
 */
/****************************

The master.cc file is also divided by a file marker. Unlike the files discussed previously, the top section of this file is maintained by DCWA. In this section, DCWA will place the necessary statements to allow the functions defined in code nodes to be used. This updating process will be discussed later in this section. The master.cc default template follows.

#include "<GLOBAL_HEADER_FILE>"
/** Do NOT change this line or the lines above!!! */
int main(int argc, char *argv[])
{
    return 0;
}

Below the divider is where the user inserts the code necessary to link the nodes together in an appropriate order. DCWA automatically creates an empty main procedure when the master file is initialized, but the user must institute the flow control that is needed and handle the output that is deemed necessary. DCWA's maintenance of the top section does not affect the code below the file marker.

DCWA uses a make file to generate the executable. Again, DCWA creates a default make file customized from its templates. When the user issues the appropriate command to generate the executable, this make file is run through the UNIX make utility. The commands in the make file cause the necessary compile commands to be issued to the system. There is no file marker in the make file, but comments instructing the user on the use of the file divide it logically into three sections. The COLLAB.Makefile default template follows.

```
<COLLABORATION = >

NODES =
NODES_C =
NODES_O =

######################################## BEGIN USER CUSTOMIZATION AREA
########################################
#
# Set CC equal to your compiler name--Sparc Compiler C++ (default)
# CC = CC
#
# Set CFLAGS to any special options you wish to pass to your compiler.
# By default, information needed by the debugger is included by the -g option.
# See the documentation for your compiler for information about options that
# you can pass to it.
# CFLAGS = -g
#
# Set CPPFLAGS to any options needed by the preprocessor. This will most
# often be the include paths (of any library header files) not searched
# by default by your compiler. For example, the line:
#       CPPFLAGS = -I/opt/openwin/include
# tells the preprocessor where to find header files for the X Window system.
# See the documentation for your compiler to determine what paths are
```
# automatically searched.
CPPFLAGS =

# Set LDLIBS to any options needed by the linker. This will most often # include directories and library names not linked by default by the # linker. For example, CC does not normally link in the X libraries, but
# LDLIBS = -L/opt/openwin/lib -lxT -lx11 # tells the linker to look in the specified directory and get the Xt and X11 # libraries. See your compiler's documentation for information on # options that can be passed to the linker.
LDLIBS =

END USER CUSTOMIZATION AREA

The rest of this file should not, in general, be modified by the user.

EXECUTABLE = $(COLLABORATION) .exe
ALL_H = $(COLLABORATION) .global.h
ALL_C = $(COLLABORATION) .master.cc \ $ (COLLABORATION) .global.cc \ $ (NODES_C)

ALL_O = $(COLLABORATION) .master.o \ $ (COLLABORATION) .global.o \ $ (NODES_O)

# Generate an executable from simulation nodes, master file, and globals
$(EXECUTABLE): $(ALL_O)
  $(CC) -o $@ $(ALL_O) $(LDLIBS)
  chmod 770 $@

# Generate an object file from a C/C++ file
%.o: %.cc
  $(CC) $(CFLAGS) $(CPPFLAGS) -c <$
  chmod 660 $@

# Compile and link the test driver for a simulation node. The resulting # file will be [nodename].test and will be located in the same directory # as the simulation node
%.test: %.test.cc
  $(CC) $(CFLAGS) $(CPPFLAGS) -o $@ $@.cc
The top section is the most important part of the file and is maintained by DCWA. The three node lines ("NODES =", "NODES_C =", and "NODES_O =") in this section define what code nodes are included in the program so that they can be compiled.

The middle section is where the user can customize the compilation process. For example, the program may make use of a particular library. The user would then need to inform the compiler where the appropriate header files and libraries are located. This section is heavily commented to guide the user in making the customizations. The bottom section contains the actual compilation commands. These commands use the information defined in the upper sections to generate an executable.

One of the choices on the main menu of the logical view window is CASE, and the user is proffered two choices from this menu: Update and Make. When the user selects Update, a dialog box containing a list of all the code nodes is displayed. This list is generated by searching the logical tree and finding all the code nodes. Once this is done, DCWA reads the top section of the collaboration’s make file. Code nodes that appear in this section are indicated by highlighting them in the dialog box. The user can then select and/or deselect code nodes in the list. When finished, the user clicks on the OK button in the dialog box. This action causes global.h, master.cc and makefile to be updated.

In the update process, DCWA first updates global.h by reading the top section of global.cc and writing it to global.h. Then to update the master file, DCWA creates a new file. First, the #INCLUDE directive for global.h is written to this new

Figure 5. Creating a collaboration
file. Next, the header file (\texttt{NODE.h}) of each of the nodes selected is copied into the file. Finally, DCWA copies the bottom half of the master file into the new file. DCWA then makes a backup of the current master file and turns the new file into the master file. In this manner, the prototypes and descriptive comments for each of the selected code nodes are included in \texttt{master.cc}. This process provides \texttt{master.cc} with the appropriate statements needed for compiling. By writing out the prototypes and comments, it is hoped that the composer of \texttt{master.cc} will be able to use the functions without having to refer back to the individual nodes.

To update the makefile, DCWA first constructs the new node lines from the code nodes selected by the user. It then reads the current makefile into a string and searches this string for the node lines. The current node lines are replaced with new ones. Finally, this string is written back to the file replacing the old contents.

Once the code is ready for compilation, the user can choose Make to compile the entire program. This function simply invokes the UNIX make utility with the following command:

\begin{verbatim}
make -f COLLAB.Makefile
\end{verbatim}

The resulting output is piped to a dialog box for the user to view. Assuming the compilation is successful, DCWA creates an executable file named \texttt{COLLAB.exe}. This file can be run at the user’s leisure.

\textbf{AN EXAMPLE}

To further explain this process, a sample run using the system follows. In the sample run, a collaboration will be created and used to implement a simple program. The collaboration will only contain code nodes, and will demonstrate how these nodes are created and then integrated together to build an executable program.

The first step is to create a new collaboration. In Figure 5, DCWA has been started and the option to create a new collaboration has been selected. The name of
this collaboration will be *demo* and it will have three group members, the creator and two others. Once the initial parameters have been set, the Start button is selected. At this time the system will create appropriate files for the collaboration and start the session. When the collaboration is started, the logical view is initialized with a root node.

The next step is to create the structure for this collaboration. Collaboration *demo* will be a simple project. It will only contain four code nodes. Figure 6 shows the placement of the nodes.

In Figure 7 the user has right-clicked on a node and brought up the node attributes dialog box for node *demo_1*. In this box, the user sets the node type and renames the node. In this test run, assume simulation is selected. When the first simulation node is created, DCWA recognizes the user’s intent for software development. It therefore creates the files needed for the CASE system and the nodes needed to access them (Figure 8). Notice in the figure that the name of the first leaf node has been changed to *init*. The names of the other three nodes are changed to *add_two*, *multiply_ten*, and *subtract_17* subsequently.

Figure 9 shows the result of the editing work by the user to the default *NODE.cc* template for the node named *init*.

Once all of the codes are complete, the user issues the *update* command to the DCWA. This command loads the names of the code nodes into a list box. From this box, the user can select and deselect which nodes should be included in the main program (Figure 10). When the nodes have been selected, DCWA places appropriate information in the master and make files (Figures 11 and 12) and does the necessary behind the scenes processing on the global file.

The next step is to write the code for the main procedure in the master file. This code will call the functions, implement any necessary flow control, and perform any needed output (Figure 13).

For this example there is no need to do any customization in the make file, so the user can issue the make command when the main procedure is complete. Output from this command will be displayed in a dialog box (Figure 14).
If any error messages are displayed in the box, the user must correct them and then attempt the make again. Otherwise, the executable file demo.exe is created. This file can be run from the command prompt (Figure 15).

CONCLUSION

This paper discusses a method in which computer-aided software engineering can be incorporated in a computer supported collaborative work environment. Research in this area indicates that DCWA is a pioneer system to provide this capability. While the technique used could not be called revolutionary, it does use existing
technology and utilities in some unique ways. The most important result of this prototype is that it has greatly helped isolate the problems and requirements that a CASE system in a CSCW environment must consider. From these considerations it is possible to define the criteria for which similar systems in the future should strive. For example, the paper refers to the individual chunks of code as building blocks. Future systems will still need to maintain this idea, but perhaps they will be more visually oriented. Future systems may allow users to take these building blocks and draw a flowchart to connect them instead of using code. Also, DCWA limits the code to C++, but future systems should allow users to code the building blocks in different languages and still be able to integrate them. There are many ways that DCWA can be expanded, but regardless the direction it evolves, DCWA should be a solid platform on which future research can be based.
REFERENCES


Figure 13. **demo.master.cc** after main procedure has been coded

Figure 14. **make utility's output**


17. D. Decouchant, V. Quint and S. Kiesler, ‘Structured and distributed cooperative editing in a large scale 
18. F. Pacull, A. Sandoz and A. Schiper, ‘Duplex: A distributed collaborative editing environment in large 
20. S. Tammaro and J. Mosier, ‘Collaborative writing is hard to support: a field study of collaborative 
writing’, Computer Supported Cooperative Work: The Journal of Collaborative Computing, 6, 19– 
CA, 1990.
23. M. Stefik, G. Foster, D. Babrow, K. Kahn, S. Lanning and L. Suchman, ‘Beyond the chalkboard: 
24. H. Ishii, and N. Miyake, ‘Toward an open shared workspace: computer and video fusion approach of 
Thesis, Auburn University, Department of Computer Science & Engineering, August 1996.
30. T. D. Dollar, ‘A method for maintaining spatial relationships among graphical objects in a distributed 
collaborative writing environment’, MS Thesis, Auburn University, Department of Computer Science & 
32. J. M. Haake, and B. Wilson, ‘Supporting collaborative writing of hyperdocuments in SEPIA’, Proc. of 
33. T. W. Dollar, ‘Ensuring user coordination and document security in a distributed collaborative writing 
34. J. Liberty and J. Mark Hord, Teach Yourself ANSI C++ in 21 Days, SAMS Publishing. Indianapolis, 
IN, 1996, pp. 609–621.