Collaborative, Distributed Software Engineering

JON A PRESTON

Clayton College and State University
Department of Information Technology

Georgia State University
Department of Computer Science

This survey paper investigates the current state of the art with respect to collaborative computing. Specifically, the paper addresses the field of collaborative software engineering and focuses on the background and issues related to distributed software development. The paper begins by exploring collaborative computing in general, discusses synchronous and asynchronous collaboration and communication mechanisms to ensure updates are handled properly, and then focuses on elements that have significant impact on distributed software engineering: mutual exclusion, achieving "undo" and "redo," organizational theory, merging code, and distributed version control. The paper then examines some of the human-computer interface (HCI) issues of such collaborative systems and presents various classification schemes that are helpful in comparing various collaborative domains and applications. The paper concludes by discussing recent and future work in the field.

Categories and Subject Descriptors: H.5.3 [**Information Systems**]: Group and Organizational Interfaces – *Collaborative Computing*; D.2 [**Software**]: Software Engineering - *Management*; I.7 [**Information Computing Methodologies**]: Document and Text Processing – *Document and Text Editing*

General Terms: Management, Documentation, Human Factors

Additional Key Words and Phrases: Software engineering, collaborative computing, synchronous, asynchronous, distributed, configuration management, merging

---

This research was supported by Dr. Prasad (CSC8530).

Authors' addresses: Jon A Preston, Department of Information Technology, Clayton College and State University, Georgia 30260. Email: jonpreston@mail.clayton.edu

TABLE OF CONTENTS

1. INTRODUCTION AND OVERVIEW OF COLLABORATIVE COMPUTING

Computer-supported cooperative (or collaborative) work (CSCW) systems bring people together and utilize computing to facilitate a work goal. The field is rich in sub-categories and has decades of research to support many of the current theories about how people can use computer technology to interact and achieve a common goal. The most pervasive example of collaborative computing (at a fundamental level) is the World Wide Web wherein users cooperatively share documents. But many other task-specific CSCW systems exist; in this paper we examine the field of computer-supported cooperative work systems in general and then discuss some fundamental problems and areas of CSCW as they related to software engineering. The intent of the paper is to provide a survey of literature pertinent to collaborative, distributed software engineering and enumerate the issues involved in a collaborative system that facilitates synchronous and asynchronous access to files within a distributed development environment.

Online systems have the potential to improve the work between people. If we view meetings as enabling coordination among activities and provide interactions among participants on a project, then we can construct a view of collaboration defined by two variables: connective richness and collaborative empowerment. Those interactions which necessitate high connectedness are those that require real-time interactions with short, pressing deadlines. In such situations, synchronous communication technologies are needed, and teleconferencing can really pay off. Those interactions which necessitate high collaborative empowerment are those that involve sequential, interdependent tasks. In this case, project scheduling, file sharing, and task coordination dominate the interactions. Interactions which involve both a high level of connectedness and collaborative empowerment are those that require cross-functional collaboration and are typically highly complex in nature; such environments

require application-sharing tools and/or screen sharing software.

Beyond just want a collaborative system can provide an organization, we must determine whether such a system will be successful within the organization. Some factors of a collaborative system's success are: support from senior management, involving users in the planning and development, standardize and use standards whenever possible, and train users on the system and how to work in virtual teams [Townsend et al. 2002].

Locasto et al. [2002] define three fundamental elements of collaborative systems: user management, content management (and version control), and process management. These are defined as (emphasis added):

> "User management is defined as the **administration of user accounts and associated privileges**. This administration should be as simple as possible to avoid wasted time and confused roles.
>
> Content management is the **process of ensuring the integrity of the data** at the heart of the project. Content management systems often employ **versioning control** that transparently preserves the progression of the project as the associated documents mature and grow.
>
> A workflow is an abstraction of the process that a task takes through a team of people. During the execution of the workflow, it is often difficult and time-consuming to manage individual processes. Process management **handles the interaction between different levels of project contributors**."

Consequently, any collaborative system must demonstrate that it manages users and data while facilitating the process and flow of the work being achieved.

What do collaborative environments look like? Many are text-based and non-graphic intensive; others are highly interactive and graphic intensive. As one example, Benford et al [Benford et al. 2001] discuss the term Collaborative Virtual Environments (CVE) as the convergence of virtual reality and computer-supported cooperative work (CSCW). CVE moves beyond videoconferencing and audio conferencing. Rather, CVE are rich 3D environments that attempt to create a realistic space in which participants may interact with each other. Given the highly-social nature of CVE and the new utilization of 3D environments, some challenges and research questions arise that include: how can such systems scale well and still preserve the interests of the users, how can various heterogeneous architectures be supported, what lessons learned in 2D environments also apply to 3D environments (and which don't), and what new human-computer interaction issues are raised given the new environments.

While the focus of CVEs is to create a rich collaborative space, other CSCW systems have different goals. For example, collaboratories seek to provide access rather than a virtual space. Collaboratories may be defined as "the emulation of a scientific laboratory in which cooperative scientific or technical work may be carried out without regard to geographic location" [Sunderam et al. 1998]. While similar to traditional computer supported collaborative work environments, collaboratories necessitate access to scientific equipment and instrumentation, lab notebooks, access to large databases, and support for near-

real-time scientific visualization that can be broadcast/shared among all participants.

A recent example of a collaboratory is the Emory CCF whose aim was to enable those working in natural sciences - physics, chemistry, and biochemistry - to utilize online interfaces to meet and work even though they were geographically dispersed. Their system utilizes parallel computation (via message passing and distributed computing) and shared resources; the system is modular in nature such that existing features can be refined or removed, and new features can be added with ease. One potential drawback of their system is that it uses a multicast communication mechanism to ensure consistency among all distributed participants [Sunderam et al. 1998].

In addition to supporting geographically-distributed work, collaborative systems often support groups that work at an extreme distance; we define these as world-wide groups (given the extreme geographic distance). Some considerations in world-wide groups (wide-area groups) are: people connect from distributed locations and often from different locations (mobility) for different sessions; synchronous and asynchronous communication must be supported; quality of service can be improved by replicating and distributing artifacts within the system; members of one group may be members of many, various other groups (multiplicity); group size may be small or large and should not be constrained; finally, members of a group should be made aware of the actions and activities of other members within the group [Marquès and Navarro, 2001].

All CSCW systems involve some level of interaction among users – thus the term "cooperative." General interaction principles are applicable, and it is interesting to note that regardless of the intent/goal of the system, all CSCW systems exhibit four general phases. The membership interaction cycle for a group consists of four phases: 1) one member interacts with the system to produce a new (or changed) artifact; 2) the system is made aware of the changed or new artifact (and/or stores the actions that brought about the change in the system); 3) this change information is distributed to other members of the group; 4) recipients of the change information process it and update their awareness of the overall system. Fundamentally, these for phases all involve communication; phases one and two involve the user and system communicating, and phases three and four involve the system communicating (and propagating) changes to other users within the collaborative environment. Thus, collaborative systems involve a high level of communication; consequently communication protocols to update peers within the system (notification algorithms) are vital to the success of any CSCW system.

Of course, not all interactions are the same. Different members often take on different roles at various times within interactions of the collaboration: active (those users that need to receive all changes), passive (those users that generate new changes and need to receive all changes from others in the group, but do not produce changes themselves), and observers (those users that only need to see summary information/events to keep track of how the group is progressing in the meta-sense) [Marquès and Navarro, 2001].

To conclude this section on the overview of CSCW systems, it may be helpful to provide a list of common categories of CSCW tools from Kock [1995]:

- Message systems
- Multiuser editors
- Group decision support systems and electronic meeting rooms
- Computer conferencing
- Coordination systems

The remainder of this paper focuses on the multiuser editor arena of CSCW systems but does discuss some common themes of all CSCW systems – communication mechanisms, coordination of messages and updates, network topology issues, and interface/design considerations.

2. SYNCHRONOUS AND ASYNCHRONOUS COMMUNICATION

To truly be flexible and support the broadest range of applications, collaborative systems must provide synchronous and asynchronous access to the shared space. While it is possible to have synchronous-only applications (chat, ICQ, etc.) and also possible to have asynchronous-only applications (e-mail, Wikis, etc.), the literature suggests that having both synchronous and asynchronous access to shared files and interaction space is advantageous because it allows for users to interact in real-time (synchronous) to achieve the highest level of concurrency while still maintaining history and the ability for new members to join "late" and review what has previously occurred (asynchronous).

Synchronous collaboration is best suited in environments where there is a need for high connectedness and real-time interactions (or when pressing deadlines are present). Witness how people collaborate in research and publications for conferences – there is often a large zone of asynchronous collaboration (trading of Web links, emails, rough drafts of papers, etc.), but when the deadline for publication is near, phone calls, teleconferencing, and instant-messaging (and chat) often dominate the collaborative interaction. There is a need for more immediate interaction in such an environment, and consequently, synchronous methods of communication are beneficial. A drawback of synchronous communication can be that the interaction is lost once completed; questions such as "what did she say about topic X" and "didn't we agree to Y" arise unless the synchronous interaction is somehow recorded for later playback (thus making it simultaneously synchronous and asynchronous).

In contrast, asynchronous collaboration is appropriate when users cannot or do not wish to interact simultaneously. Synchronous communication by its very nature demands an immediate response and can interrupt work and actually cause a decrease in productivity; thus when people are working on very mentally-demanding tasks that require their focused attention, synchronous collaboration can be quite negative. In situations like these, asynchronous collaboration allows users to participate in the collaborative environment at their own pace and when it is most appropriate for them. Asynchronous collaborative environments are also appropriate when users are geographically displaced as time zones often make it difficult to collaborate at the same time. Asynchronous systems also have the advantage that they automatically record the interaction so others can view it later (asynchronously); thus there is an historical record that can be archived and reviewed later by all members of the collaboration.

Now that we've discussed synchronous and asynchronous collaborative models, let's focus on how to enable the

ubiquity of collaborative environments. It can be argued that until adding collaboration into an application is as easy as dropping a visual control on a form, developers will still struggle with implementing fundamentals such as network communication, synchronization, and event propagation. Just as most modern APIs contain controls/classes to facilitate network communication, file I/O, multithreading, etc. (all concepts that at some point historically were novel and difficult to achieve programmatically), collaborative systems will not become pervasive and easily achieved until a similar set of APIs exist for collaboration. Roth and Unger [2000] have created such a set of controls; TeamComponents is a set of visual controls that have at their core the ability and API to communicate collaboratively with other instances of the application into which the controls are used. Because the API is already embedded into the control, coordination and synchronization is already achieved. They make the case that if controls could worry about their internal data states and communicate and coordinate such state with other applications, collaborative interfaces and applications would be much easier to develop; consequently, collaborative environments would expand in their use and acceptance within computing.

Another interesting application to object-oriented development is the idea of encapsulating the contents and communication mechanisms within the controls themselves; thus a control is completely responsible for and able to provide collaboration and synchronization with other instances of the control within other users' applications.

State information such as who is also viewing a control or who is editing a control's state can be shown in a non-intrusive manner through small icons within the visual control [Roth and Unger, 2000].

It is hoped that synchronous and asynchronous modes of communication can be added to programs in such a "drag and drop" fashion and alleviate the developers from the arduous task of "reinventing the wheel" when it comes to collaborative systems development.

3. UNICAST, MULTICAST, AND OTHER NOTIFICATION ALGORITHMS

The change notification approach that a system adopts widely influences the uses that such as system supports. For example, if change notifications are infrequent, then the system will typically be used for asynchronous collaboration; whereas if change notifications are frequent, then the system will typically be used for synchronous collaboration. In a unicast system, notifications are sent to peers sequentially; the advantage of this model is that the network congestion is kept small, but the change propagation delay can be excessive. In a multicast system, notifications are sent to peers in parallel; the advantage of this model is that the propagation delay can be minimized, but the disadvantage is that the bandwidth consumption/congestion can be too costly.

It may be advantageous to adopt a general-purpose, flexible notification mechanism that is adaptable to the needs of the collaborative system; such a system should contain two principle parts: the notification policy (frequency and granularity of notifications) and the notification mechanism (the implementation of the policy). Key elements of such a system involve incoming and outgoing buffers to received and send messages [Shen and Sun, 2002].

Safety properties of multicast communication within group communication systems (GCS) include: delivery integrity (for every receive event, there is a preceding send even for the same message), no duplication (we cannot have two receive events at the same process that contain the same message/content), sending view delivery and same view delivery (message send/receive must be within the same context - i.e. the same view), virtual synchrony (processes must keep consistency among views - i.e. if message m transfers process p from view V to view V', then the same message m must have been received by and processed in process q from V to V'), transitional set (processes are able to locally decide whether they are synchronized with other processes or whether they must transition into a new view to maintain synchronicity), safe delivery (all members of the current view have received the message via the network), reliable FIFO message delivery, and liveness; liveness is ensured by a reliable, independent third party that can guarantee that the communication between separate processes is available and all messages sent will be received [Chockler et al. 2001].

Others have done work with regard to update and communication protocols for collaborative systems. These include the McCanne et al. [1999] system entitled MASH that seeks the enable scalable multipoint collaboration using "lightweight sessions" where thin application-specific protocols have been developed on top of IP multicast; sessions are grouped together to avoid network flooding. This is similar to the notion of super nodes and networks of clusters in KaZaA. Their approach also seeks to go beyond existing APIs that provide protocols and mechanisms for communication (ActiveX, ObjectTcl, etc.) and "extract their commonalities into a high-level architecture that is reusable."

To ensure that replicated objects are synchronized in distributed environments, Vidot et al. [2000] have devised an algorithm that defers broadcasts of operations to others in the system, potentially reducing bandwidth consumption. One drawback of this algorithm is that there is potential latency in that others in the system have stale copies of the objects that have been modified; also, even though the changes are sequenced and ordered causally, achieving "undo" in the deferred broadcast algorithm can be problematic (since the algorithm assumes that no operation will be undone and immediately redone).

Another field of distributed collaborative systems that requires complex notification algorithms is the area of design. Wu and Sarma [2001] developed an algorithm that involves working in complex highly-detailed CAD systems with large data sets. Prior work in this area of collaborative computing dealt with boundary representations (b-reps) to provide update notification for the models using a central database (to ensure consistency) and broadcasts of updates to keep locally-replicated copies coherent. Like distributed collaborative systems in general, the problem with the centralized approach is that it does not provide for a high level of collaboration (and provides on point of failure in the system); the network bandwidth consumption in the broadcast model is too costly and does not necessarily scale well. Wu's and Sarma's [2001] work is novel in that if we assume that edits to the system's data are valid, then our notification algorithm can simply propagate the changes to the peers in the system and only focus on the segmented region of the system that has been changed (rather than the entire system). In text-based systems, this

involves notifying peers of the location and content of the text change; in object-oriented systems, this involves notifying peers of the objects and the actions/methods invoked (or attributed modified); in graphics-based or design systems, this notification involves specifying the region (or boundary representation – b-rep) and the changes made to the element(s) within the region. Consequently, this approach can dramatically reduce the overall bandwidth for notification to peers in the system while maintaining the advantageous distributed nature of avoiding a central copy/version of the system's data.

The choice to adopt unicast or multicast is dependent upon the nature of the collaborative environment and the time delays that are acceptable in change notifications. Segmenting the system's data space can help to reduce the amount of data needed in the notification, and creating a hierarchy of the nodes within the system can help reduce the amount of network traffic needed to accomplish the change notification.

4. DISTRIBUTED MUTUAL EXCLUSION AND FILE ACCESS

We have established various models of how users interact (synchronous, asynchronous, or a combination), and we have discussed algorithms for propagating changes to other users in the system (unicast, multicast, segmented, and hybrid approaches); we now turn our attention to ensuring that users within the shared space have exclusive access to the elements of the shared data and are provided adequate access to the files within the system.

A "Distributed Version Control System" (DVCS) is one in which version control and software configuration control is provided across a distributed network of machines. By distributing configuration management across a network of machines, one should see an improvement in reliability (by replicating the file across multiple machines) and speed (response time). Load balancing can be another benefit of distributed configuration management. Of course, if file replication is employed, then we must implement a policy whereby all copies of the file are always coherent [Korel et al. 1991].

In order for distributed configuration management to work efficiently, the fact that the files/modules are distributed across multiple computers on the network must be transparent to the developer/user. The user should not be responsible for knowing where to locate the file he/she is seeking. Rather, the system should be able to provide an overall hierarchical, searchable view of the modules present in the system; the user should be able to find their needed module(s) without any notion of where it physically resides on the network.

Another interesting aspect of distributed configuration management is the idea that the system provides each user with a public and private space for the files. The public space contains all of the files in the collaborative, distributed system. The private space contains minor revisions or "what if" development files that the local user can "toy with" in an exploratory manner; this provides a safe "sandbox" area that each developer can use to explore possible ideas and changes. When a module is ready for publication to others, it is moved from the private space into the public space [Korel et al. 1991].

Guaranteeing mutual exclusion to the critical section is a classic problem in computing. In the cases of distributed software engineering in a collaborative environment, we need to guarantee that

only one user can be editing any section of the collaborative shared space at any given time. In some cases, we might like to allow k users to have simultaneous access to a shared resource (where $k \leq n$, n = total number of users in the system). This section examines the various mutual exclusion algorithms that are relevant within the context of collaborative systems.

Distributed mutual exclusion algorithms fall into one of two primary categories: token-based and permission-based. In a token-based system, a virtual object, the token, provides the permission to enter into the critical section. Only the process that holds the token is allowed into the critical section. Of interest is how the token is acquired and how it is passed across the network; in some models, the token is passed from process to process, and is only retained by a process if it has need for it (i.e. it wants to enter the critical section). Alternatively, the token can reside with a process until it is requested, and the owner of the token makes the decision as to who to give the token to. Of course, finding the token is potentially problematic depending upon the network topology [Velazquez, 1993].

The other approach to distributed mutual exclusion is the permission-based approach. In the permission-based approach, a process that wants to enter the critical section sends out a request to all other processes in the system asking to enter the critical section. The other processes then provide permission (or a denial) based upon a priority algorithm, and can only provide permission to one process at a time. Once a requesting process receives enough positive votes, it may enter the critical section. Of interest here is how to decide the priority algorithm and how many votes are necessary for permission [Velazquez, 1993].

In the case where we would like to allow some subset of users access to the shared resource (or shared data) simultaneously, the work of Bulgannawar and Vaidya [1994] is of particular interest. Their algorithm achieves k-mutual exclusion with a low delay time to enter the critical section (important to avoid delays within the system) and a low number of messages to coordinate the entry to the critical section. In their model, they use a token-based system where there are k tokens in the system; further, the system is starvation and deadlock free [Bulgannawar and Vaidya, 1994].

The k-mutual exclusion algorithm differs from the traditional mutual exclusion algorithm in that in a network of n processes, we allows at most k processes into the critical section (where 1 <= k <= n). The algorithm developed by Walter et al. [2001a] utilizes a token-based approach that contains k tokens. Tokens are either at a requesting process or a process that is not requesting access to the critical section. In order for this algorithm to work, all non-token requesting processes must be connected to at least one token requesting process. To ensure that tokens do not become "clustered," the algorithm states that if a token is not being used, then it is sent to a processor that is not the processor that granted the token [Walter et al. 2001a].

Distributed token-based, permission-based, and k-mutual exclusion algorithms are all useful in various scenarios. The primary use and impact of distributed mutual exclusion in the context of this survey paper is to manage access to shared code in the software engineering system. This is a vital part of any distributed software development environment with simultaneous users access shared source code.

5. ISSUES RAISED IN AD HOC (UNRELIABLE) AND PEER-TO-PEER NETWORKS

While most modern software engineering is performed within company local-area networks (LANs), as networking becomes more ubiquitous and as the trend of tele-working and distributed working continues, one can expect more software development to involve ad hoc, non-LAN clients and even peer-to-peer clients. To enable more concurrent software development, the system must be accessible and available at any time from any place. Of course, there are security issues raised in such an accessible system, but we will not discuss these in this paper. Rather, we focus our attention to the matter of providing connectivity and access to the distributed software engineering system within ad hoc and peer-to-peer networks.

A mobile ad hoc network is one that "pairs of nodes communicate by sending messages either over a direct wireless link, or over a sequence of wireless links including one or more intermediate nodes" [Walter et al. 2001b]. Communication from a node is only possible within a radius of transmission, and given the mobile nature of such devices, changes in the network topology and communication failure between two nodes is commonplace. Consequently, mutual exclusion algorithms that assume consistent network topologies and reliable communication are not necessarily well suited for wireless ad hoc scenarios [Walter et al. 2001b].

When dealing with wireless and ad-hoc networks, the topology changes frequently (since machines are moving in and out of the transmission range), power consumption levels and battery life and low, and the delay between messages is highly unpredictable and often takes quite a bit of time. Consequently, implementing algorithms that are reliable in such networks requires careful attention to detail and reliability.

Within a peer-to-peer network, many obstacles must be overcome to ensure that the peer is able to communicate efficiently and correctly with other peers in the network. First, a peer must be able to speak the "language" of other peers. In a homogenous P2P network, this is trivially done since all peers speak the same language; but in heterogeneous P2P networks, each peer requires a "wrapper" that can make heterogeneous communication possible by translating to and from protocols and "languages." Additionally, since different peers provide different APIs, a "mediator" is needed at each peer to convert the command from the local peer to the meta-command that the entire P2P network speaks. Thirdly, a "facilitator" is needed to local peers, services, and other agents in the P2P network; the actions of a facilitator involve registering with other peers and "advertising" their set of services (perhaps through reflection). Finally, the last role that each peer must provide is that of a "planner" who is responsible for recovering from failure or lost communication with other peers [Penserini et al. 2002].

While it is not currently easily possible to develop software on wireless phone interfaces, in the future, wireless ad hoc networks will become more pervasive. Thus it is useful to examine these network models and the issues they raise in discussing the future and current work within distributed software engineering.

6. ACHIEVING "UNDO" AND "REDO" IN COLLABORATIVE SYSTEMS

The ability to undo in a single-user system is vital to enable a user to reverse a mistake; the ability to undo in a collaborative system is even more essential because other users can destroy another's work, and the very nature of collaborative systems encourages exploratory work that may need to be undone. In single-user object-oriented systems, undo may be achieved by reversing the operation on the object; in single-user text-based systems, insert and delete are natural inverses of each other, thus undo and redo are easily implemented in text-based systems.

Potential uses for “undo” and “redo” include: [Sun, 2002]

- Recover from unintended or incorrect operation
- Learn new system feature by exploring the interface (i.e. try and undo)
- Explore alternatives (i.e. try and undo multiple times)

In contrast, in bitmapped systems, undo and redo are not as easily achieved because the users are directly manipulating color and pixel information (in a more "freehand" manner).

One approach in achieving undo/redo in bitmapped based systems is to record the actions of the users and only store the pixels that have been affected. If an operation is undone, then the operations preceding the operation undone are replayed to generate the correct state. While this may take more time, the cost associated with storing states is dramatically reduced, since we are only storing the operation performed at each step and the pixels affected. The "reconstruction" of valid state does take some time, but as is often the case, memory requirement is reduced at the cost of this CPU/algorithm performance/time [Wang et al. 2002].

Now that we have examined the various types of systems that can employ the undo and redo operations, let’s examine the scenario that arises in a multi-user (collaborative) text-based system in detail and explore algorithms to efficiently handle concurrent undo/redo in collaborative systems.

As a point of reference so that the reader understands the problem, imagine the simple case of two users ($User_1$ and $User_2$) interacting synchronously in a collaborative text document system. If the shared document contains the string “man” and $User_1$ performs an *Insert*(3,"i") – i.e. insert the character 'i' at the third position, then the resultant string should be “main”. If $User_2$ then performs an *Insert*(2,"d"), then the resultant string should be “mdain”. Now, if $User_1$ performs an “undo” operation, we cannot simply inverse the *Insert*(3,"i") operation previously invoked because the character 'i' is no longer at position 3; it is at position 4 (due to the operation by $User_2$). Thus there is a need for a robust solution to the “undo” and “redo” problem in collaborative text-based systems that transcends simply inverting “insert” for “delete” and visa versa.

Sun [2002] describes a system entitled REDUCE (REal-time Distributed Unconstrained Cooperative Editing) that allows for any operation to be undone at any time. The key to the REDUCE system is that it separates the “undo policy” from the “undo mechanism.” This allows the system to process the undo request in the context in which the original command was

executed relative to the changes that have occurred since the command on the local user's machine. Further, the undo mechanism is broken into two separate layers – the high-level transformation control layer (that is responsible for understanding the concurrency relationships among past actions in the system) and the low-level transformation function (that is responsible for actually carrying out the desired action as it relates to "types, parameters, and other relationships" within the system).

Sun [2000] also defines a consistency model that states a collaborative editing system must ensure the following properties:

1. **"*Convergence***: when all sites have executed the same set of operations, all copies of the shared document are identical.

2. ***Causality preservation***: for any pair of operations O_a and O_b, if $O_a \rightarrow O_b$, then O_a is executed before O_b at all sites.

3. ***Intention preservation***: for any operation O, the effect of executing O in any document state is the same as the intention of O."

Sun's work hinges on the claim that these properties should hold for all group "do" operations and all group "undo" operations.

7. TRANSPARENCIES AND AWARE CSCW SYSTEMS

Heretofore, we have discussed communication methods, ensuring access to shared content, network configurations for collaborative systems, and ensuring proper interaction with shared content. We now turn our attention to two models of computer supported collaborative work systems: transparencies and aware systems.

Transparent collaborative systems are so named because the applications that are being shared among multiple users have no idea of the collaboration - the collaborative interface acts as an intermediary buffer for the application and receives all users' input and relays these interactions to the application; when the application responds and adjusts its output, the collaborative system/agent relays this information to all users' computers such that all users see the same interface. The advantage of such transparent systems is that they can be integrated into most single-user applications without the need to recompile or edit the original application.

Aware collaborative systems are so named because the collaborative interface is embedded within the application itself and the system's core interface and operations support synchronization and distribution/sharing of the system's content. These systems are defined as aware because the application is "aware" that the content is being shared and the interface of the system enables such sharing. While there are many benefits of embedding the collaboration within the application, the disadvantage is that the source of the application must be available and the collaborative API (synchronization, mutex, etc.) must be tightly coupled within the application. This is often not possible, thus the need for transparent systems.

Application sharing and transparency are two different approaches to collaborative systems. Application sharing involves either centralizing the application's execution and distributing the input and output (display) among user machines or creating a replicated,

homogenous architecture in which each user runs the same application across a network; with either model, the user is constrained to use the same application as all other users in the collaborative environment. Even in heterogeneous application sharing environments, considerable concerns must be overcome in supporting the capture, communication, and replication of users' actions.

In comparison, transparency-based systems allow users to share applications without modifying the original program. Transparencies originally involved screen sharing technologies in which the user would share the entire screen to other users. These systems evolved into sharing only specific windows or applications, rather than the entire screen, and are best represented by the X windows protocol.

Under conventional collaborative transparent system, concurrency is not possible - only one user is able to input to the application at any given time; while this is appropriate for presentations and shared meetings, this is too limiting for collaborative software development. "Floor control" is the term used to define which user has access to the input stream (mutex), and this is needed to ensure that event interleaving is avoided.

One promising concept of being able to merge the best of transparent and aware collaborative systems is the modern object-oriented concept of reflection [Li and Patrao, 2001]. If a developer wanted to transform a single-user application into a collaborative multiple-user application but did not have access to the source code, then through reflection, the developer could extend the program and add the communication/synchronization API into the system externally via reflection. Unfortunately, this approach does require a high-level knowledge of the internals of the single-user system, and even without access to the original source code, in-depth knowledge of the internals of the system is often required.

An alternative approach would be to design systems that allow users to establish relationships to objects within the system and extend the collaborative software to support such relationships [Li and Patrao, 2001]. Of course, the prerequisite of this type of system would be that the collaborative API be built into the current system and that the system supports extension by allowing the user to establish relationships between objects. Li and Patrao's model exhibits such an interface by viewing the elements of the collaborative interaction as objects that support emergent sharing and distributed referential integrity. Such objects inherit common attributes and provide a generalized API for modification such that these modifications (small differentials) can be broadcast to the users of the system and tracked; this avoids the more costly low-level messaging (transparency-based) system wherein all display information is broadcast.

Li and Li [2002] discuss current advances in the area of transparencies that should support spontaneous application sharing (i.e. a user can use a single-user application and then later decide to publish/share the application to another user) and support heterogeneous clients and independent views. Additionally, the issue of "late comers" needs to be addressed in modern collaborative environments: how can the system bring new users that were not present at the beginning of the session up to speed quickly; OS hooks such as the Microsoft Windows API provides such capabilities that allow collaborative transparencies to record sessions for replay on future, late arriving clients [Li and Li, 2002].

Begole et al [1999] discuss a synchronous methodology for providing a "transparent" collaboration system that works in coordination with existing applications. This system is different from other existing collaboration transparencies in that it avoids the "conventional" centralized architecture that require that only one person interact with the system at any given time (single token-based mutex). One difficulty that is avoided in such single-controller transparent collaborative systems is that of interaction interleaving; since only one user can "control" the cursor, then interactions cannot be interleaved incorrectly (i.e. the input is by definition sequential in nature and no undesired overlap is possible).

Begole et al [1999] list four attributes that are useful in comparing aware and transparent collaborative systems.

	Transparency	**Aware**
Concurrent Work	Single	Multiple
WYSIWIS	Strict	Relaxed
Group Awareness	Little	Detailed
Network Usage	High	Low

Table 1: Comparing Transparent and Aware Collaborative Systems

These attributes are defined as:

Concurrent work: Does the system allow for multiple users to provide input simultaneously, or is only one user able to provide input at any given time?

WYSIWIS: All users should see the same state at all times; What You See Is What I see.

Group Awareness: How much detail does the system provide with regard to what other users in the system are doing and what section of the document they are viewing? Some systems simply provide a pointer/cursor showing the current "location" of the other users; other systems provide thumbnails and more detailed views.

Network Usage: How much network bandwidth is consumed and needed by the system? In aware systems, operations are typically all that is communicated (and these messages are small), whereas in transparent systems typically rely upon centralized server architectures and broadcast display change information (quite large).

In conclusion, aware collaborative systems consume less bandwidth, allow for concurrent work, more easily provide flexible WYSIWIS interfaces, and allow for more inherently robust group awareness. Transparency-based collaborative systems are useful in situations where the developer needs to create a collaborative system based upon a single-user application but does not have access to the underlying code base of the single-user system; transparency-based systems often consume more system resources and require a centralized server model, but they are often the only option in some circumstances.

8. COLLABORATIVE, DISTRIBUTED SOFTWARE ENGINEERING

Coordination efforts within software development projects is not a new concept; NATO organized a meeting of software developers in 1968 that defined the term "software engineering" and identified the complexity of managing software development as a key challenge within the field [Grinter, 1998]. Thirty-six years later, the issue of coordinating the software development process – including the

technical side and the human side – is still a challenge. Brooks, Parnas, and Conway (among others) all recognize the importance of team structure and collaboration among members as key to the success of software development [Grinter, 1998].

Computer-Aided Software Engineering (CASE) tools are useful in supporting the development of software systems. Lower-CASE tools are primarily focused on supporting the implementation and testing phases of software development; upper-CASE tools are those that are primarily focused on supporting the design and analysis phases of software development [Sommerville, 2001].

It is important to note that most software engineering activities involve the coordination and collaboration of documents related to the software system. These documents consist of system requirements specifications, design documents, project schedules, risk tracking, features list documents, test case documents, and software source code. While many CASE tools exist to help manage the software development process, fundamentally all software engineering activities involve collaborating on documents; even source code, as structured as it is, can be viewed as a text document.

Regardless of whether the system employs lower-CASE and/or upper-CASE tools, modern software engineering of large-scale software systems involves a high level of collaboration and coordination. Software engineering offers an excellent opportunity to examine systems, subsystems, groups, and subgroups within the context of CSCW [Borghoff and Teege, 1993]. Consequently, the features of general-purpose CSCW tools are readily applied to software engineering.

Section 8.1 and 8.2 of the paper examine two main areas within CSCW that are particularly relevant to software engineering – managing collaborative teams and managing collaborative code. Section 9 then continues section 8.2 and goes into more detail with regard to distributed version control.

8.1 Organizational Theory and Group Management

Collaborative editing systems (CES) are central to distributed, collaborative software engineering. Without the ability to collaborate on documents, the system cannot function. Central to the ability to collaborate on documents is the ability to work within a group and coordinate group effort. In a traditional software engineering setting, these activities entail project task scheduling, status reporting (and meetings), and inter-group communication.

Borghoff and Teege [1993] present a model for collaborative editing that "mediates coordination and cooperation" and make the case that such a system can be used in the software engineering domain. They define the multi-user distributed system titled "IRIS" that consists of a data layer an operations layer. Static user information (such as phone numbers, email addresses, etc.) is stored in the data layer so communication is facilitated. Explicit and implicit coordination is provided by the operation layer, where implicit takes care of mutual exclusion for collaborative editing, and explicit allows users to soft and hard lock and communicate their activities to others in the collaborative space. The model also allows users to defined new parts, remove existing parts, and move parts in a structured edit (assumes that the documents in use have structure - SGML, ODA, etc.).

Borghoff and Teege [1993] also have an interesting view in the systems applicability to software engineering. The authors make the case that software engineering consists of document manipulation and coordination of collaborative development. The code of the software system being built can be coordinated using their explicit and implicit coordination structure; versioned automatically because the model contains "history information;" report current activities because the system tracks dynamic user profiles (who has recently done what and is currently doing what); and can extract the latest build/version of the system [Borghoff and Teege, 1993].

What is most novel about the "IRIS" model is that it explicitly separates the structural information of the document from the content information of the document. This allows the transformation operations on the document to more easily be achieved. Of course, the system has the advantage of working only with highly-structured documents, which is often not the case in general-purpose document editing. Fortunately, software engineering documents and source code are most often highly structured; therefore, this model is very applicable to the field of collaborative software engineering.

Collaborative software development by its nature involves four essential problems: evolution (changes must be tracked among many different versions of the software), scale (increased software systems involve more interactions among methods and developers), multiple platforms on which the system will be deployed (coupling the methods and subsystems of the software), and distribution of knowledge (in that many people all are working on the system and each contain a set of the working knowledge of the system) [Perry et al. 1998]. The central question to ask in parallel development is one of managing the scope of changes within the system over time. Certainly we can employ process (as has been done for decades) to manage the software development activities, but more and more, CASE tools are being utilized to help manage the growing complexity and tightly-coupled activities within software development.

A recent study that tracked the number of changes to files by different developers and found that 12.5% of all changes were made to the same file within 24 hours of each other; thus there is a high degree of parallel development with a potentially high probability that changes made by one user would have an impact on the changes made by another developer. The study also reports that there were up to 16 different parallel versions of the software system that needed to be merged - quite a task [Perry et al. 2001]!

Another recent study [Herbsleb et al. 2000] investigated a software development project that spanned six sites within 4 countries on two continents and a seventh site on a third continent acting as a supporting role. The study found that when teams were distributed, the speed of development was delayed when compared to face-to-face teams. Also of note was the fact that team members report that they are less likely to received help from distant co-workers, but they themselves do not feel that they provide less help for distributed co-workers.

The study's finding concludes with the idea that better interactions are needed to support collaborations at a distance. Better awareness tools such as instant messaging and the "rear view mirror" application by Boyer et al. [1998] offer potential for overcoming some of the problems inherent in distributed software system development [Herbsleb et al. 2000].

In the more general sense (transcending beyond software engineering), emergent models of organizational theory suggest that there is a movement away from hierarchical forms of group coordination to utilizing information technology in facilitating more adaptive, flexible structures. Such structures are often termed "network organizations" or "adhocracies" [Hellenbrand, 1995] and offer the possibility for more productive and efficient groups and organizations. By utilizing collaborative computing environments, transaction costs are reduced and coordination of tasks is improved.

Certainly there is a need for coordinating the collaborative nature of software development. This problem is beyond the scope of version control systems and must address the very human side of software development. This view is supported by the very name of the CSCW field in that collaboration (and cooperation) occurs among humans. Consequently, CSCW systems must contain elements that facilitate the communication and coordination of group activities.

8.2 Merging Code

Beyond coordinating the flow of work and communication of the members within a group, a robust collaborative software engineering system must also manage the source code of the software system that is being shared among the users.

Grinter [1998] defines the term "recomposition" as "the process of reassembling the product" and takes a reverse approach that begins with the end and transitions to the origin. He points out (quite correctly), that modern software engineering supports the claim that modularization is key to managing the complexity of large software systems. If software systems are modularized, then at some point they must be recomposed into a whole; this is not done just to deliver the product to the customer but also during the development of the system for testing and demonstration purposes. In a modern rapid application development (RAD) model, recomposition may occur as often as every night [Lory et al. 2003] to ensure the product is always making progress and there is always something to show the customer.

If developers are modifying the source code in parallel, then there will most probably exist some module within the code that has been modified by more than one developer (see results from Perry et al. [2001] in section 8.1); this situation where the same module has edits from multiple developers requires recomposition. Another need for recomposition comes when there are dependencies between modules themselves; for example, if module A produces output for module B, and if a user modifies either module, then the resulting connection between the modules may be broken (i.e. if the two modules are coupled at all, editing one may break the communication mechanism of the two modules). While well-defined change request processes can help mitigate this potential problem, in practice, processes are often not followed as well as we would hope [Grinter, 1998].

Thus the need to merge (or recomposition) distributed source code is established. But what are the properties of a successful/correct model for merging? Harrison [1990] defines three properties essential for correct coordination consistency. These are:

Change-serializability: if a change is made in parallel by two users within the system, then one change will not overwrite the other (i.e. the changes can be serialized as in a DBMS). This

property also ensures that changes can be explicitly undone or overwritten by another explicit action by a user.

Atomicity: if a modification activity by a user is committed to the shared source, then all actions within the modification activity will be written (i.e. the entire source will be updated to reflect all changes). This property ensures that all of the changes will be committed to the shared source or none will be committed.

Completeness: this property establishes a causal relationship between modification activities. For example, if modification A precedes modification B, and modification B precedes modification C, then the source code state that A acts upon also precedes the state that B acts upon, which in turn precedes the state that C acts upon.

One interesting outcome of the completeness property is that since modification operations are causal, then the undo and redo operations are more easily achieved (see section 6).

How can we automate the identification of elements (source lines of code) that have been modified by parallel users that conflict when merged? Sun and Chen [2002] define a "conflict relation matrix" (CRM) that is useful. Given n modifications, M_1, M_2, … M_n, then the potential conflict among these modifications can be expressed as an $n \times n$ matrix **CRM**, where **CRM**[i,j] is true if M_i conflicts with M_j., otherwise **CRM**[i,j] is false. Note that **CRM**[i,i] is always false (a modification cannot conflict with itself). Even given the fact that the matrix is symmetric along the diagonal, there are still $O(n^2)$ possible conflicts. In order to have merging occur in a responsive time scale, we must be able to parallelize the operation of merging changes in a distributed environment.

Now that we have established the need for and properties of correct merging, let us examine ways to accomplish merging in parallel. Kaiser and Kaplan [1993] discuss the concept of a parallel change propagation algorithm that uses the tree-like structure of a programming language grammar to distribute changes to multiple users in the system. The model assumes an object-oriented approach where attributes and methods exist within the language such that a change in one method or attribute of a class must be propagated to all other clients of the class. This is achieved in parallel, and the change propagation is decoupled from the users' ability to continue to edit within the system; consequently, the response of the system is quite high. The authors do discuss the most problematic scenario in which a user repeatedly performs a change and undo pair (i.e. change A, undo, change A, undo, change A, undo ...) and conclude that this is still manageable in their model.

The model assumes segmentation at a modular level, but addresses the issue of multiple edits by different users within the same module; in this case, it is suggested that multiple copies of the module are distributed to the clients and a merge operation is performed to resolve the differentials among the clients performing the change [Kaiser and Kaplan, 1993].

Another interesting aspect of Kaiser and Kaplan's work is that they achieve parallel synchronization by the use of "firewalls" (i.e. mutex). User has a "cursor" (place marker) in the tree that locks the subtree that is being edited. When the modification begins, the firewall is raised; when the modification is complete, the firewall is lowered and the change is propagated to other users in the system. This is necessary to guarantee that no two users are ever able to edit the same element in the system at the same time. While this

model is somewhat pessimistic in its locking mechanism, given the high structure of the system and the fact that the element to be locked is quite small in nature, one may (correctly) hope that collisions with exclusive access would be rare.

9. DISTRIBUTED VERSION CONTROL

Concurrent modification of shared source code can create problems of consistency (see section 8.2). One may consider a simplistic solution to the consistency problem to be version control. While version control and configuration management are key elements of a distributed concurrent software engineering system, these aspects deal only with building and releasing a software system [Harrison, 1990]; thus there is a need to control concurrent modification/merging apart from managing the files themselves. This section addresses the former – managing various versions and files of the software in a distributed, collaborative development environment.

Distributed version control can be approached via three main models: turn taking, split-combine, and copy-merge. All have advantages and disadvantages.

The turn taking approach to collaborative development suffers from the fact that only one participant can edit the document at any given time; this reduces the parallel nature of collaborative development. The split-combine approach assumes that the splits can be static and that there is very little interaction among participants; this is often not the case as different sections of a system can be tightly coupled and dependant upon each other. The copy-merge approach has a high degree of parallelism in the sense that all participants can edit the files/documents at the same time, but the merge step of combining all of these changes can become quite difficult and costly [Magnusson et al. 1993].

Configuration management systems typically take one of two approaches with regard to locking: optimistic or pessimistic locking. In the optimistic approach, developers are free to develop in a more parallel fashion, but conflict occurs at the merge point when two sets of files must be merged together and changes brought together (and avoid losing work and ensuring that changes in one file have not adversely affected changes in the other file). In the pessimistic approach, developers must obtain a lock on a file before being able to edit it; this can reduce the parallel nature of development since at most one developer can edit the file at any time.

Both optimistic and pessimistic configuration management rely upon the user to query the CMS as to the state of the file; a better approach would be one in which the emphasis shifts to a "push" information flow where the system updates the user as to who is also interacting with the files that they are interested in. Palantir [Serma et al. 2003] is one such system that takes an active role in informing users of changes and graphically depicts a heuristic measure of the severity of change with respect to the users' local copy of the file.

Palantir takes an event tracking approach, where events are: populate, unpopulated, synchronized, changesInProgress, changesReverted, changedCommittted, added, removed, renamed, moved, severityChanged [Serma et al. 2003]. All of these notification messages are intended to manage changes at a fine level of detail to keep peers within the system aware of what edits are being made by others in the system.

Since software development by nature necessitates the ability for developers to edit various sections/modules/files of a system at any given time, any collaborative software development system must support the users in having control of the entire system's code base. Locking an entire file (or subsystem) is too costly and potentially blocks other users from being able to at the very least view the document [Harrison, 1990]. Magnusson et al. [1993] defines a fine-granularity approach to revision control that focuses on language elements (classes, methods, attributes, functions, etc.) and merges these smaller elements; since the elements are smaller in nature, it is posited that edit collisions will be reduced, and merge operations will become more manageable [Magnusson et al. 1993].

Software configuration management (SCM) is a critical activity with respect to ISO9000 conformance and is a key activity of the Capability Maturity Model (CMM); SCM acts to help control changes in software products such that accounting, progress, and functionality may be more easily measured; SCM also facilitates developers in making controlled changes to the software system and tracking the inevitable evolution of the software code base.

The evolution and change of software artifacts may be categorized into three actions: sequence (in which elements are added and the artifact is expanded), tree (also referred to as branching in which the artifact splits into two distinct but similar children wherein each child has different functionality than its sibling), and acyclic graph (in which branching occurs but the children then merge and sequence into one artifact with a union of functionality) [Conradi and Westfechtel, 1998].

The problem of merging two software artifacts is problematic and the approaches to this problem may be categorized as "raw merging" (in which the contents of one version are added to the contents of another), "two-way merging" (in which two versions are presented to the user - with the differences noted - and the user must select which elements are to remain in the final, combined version), and "three-way merging" (which is similar to two-way merging with the added feature that the parent of each version is used as a common baseline to reduce the number of elements that necessitate the user's input/choice) [Conradi and Westfechtel, 1998].

10. USER INTERFACE ISSUES

Beyond what underlying algorithms and models a distributed collaborative software engineering system employs, ultimately the user must be presented with an interface in which to interact with the system. Getting the interface correct for collaboration can be problematic; users are loathe to give up their existing tools and single-user applications with which they have a high level of comfort, yet the benefits of collaborative systems bear examination and potential adoption if the users can adopt them and effectively use them.

What degree of immersion should a system provide? Boyer et al. [1998] determined via interviews that team members do not need virtual environments that are overpowering and immersive (which is in direct conflict with the collaborative virtual environment (CVE) work of [Benford et al. 2001]). Rather, they seek tools that are passive, unobtrusive and provide the information that they need about their colleagues without creating unnecessary overhead in a new interface. The system proposed by Boyer et al. uses a progressive-scale model that goes from left to right; those users that you place on the left side of the window are

"important" enough to allow them to interrupt your work and communicate with you; those in the middle allow for "bubble" popup messages that are small and easily ignored if you desire; and those on the right are blocked from interrupting you at all. The overall interface allows users to keep track of who else is in the collaborative system at the same time while still maintaining privacy and giving the user the power to control and avoid interruptions [Boyer et al. 1998].

Another study by Cheng et al [2004] show that meetings, email, software engineering process, and meetings can consume more than half of the average work day. Improved processes and better use of technology can help reduce this burden on development and allow more of the work day to be devoted to developing the software system. The authors make the case that if the Integrated Development Environment (IDE) is the central interface to the developer, why not integrate collaborative technologies that facilitate communication into IDEs. Booch and Brown refer to the intelligent integration of software development tools into the known interface with a positive net effect as "reducing fiction" in the development process; the authors go on to show that configuration management, screen sharing, and email and instant messaging would be useful collaborative tools to integrate into existing IDEs. Adding email and instant messaging to IDEs has the added benefit of automating source code (and requirements) change requests as well as automatically tracking version branching.

The study claims that many modern IDEs contain support for extensibility, but that these are simply additions to the user interface and run externally as scripts or "plug ins." For such collaborative tools to truly be effectively integrated into the IDE, the collaborative tools and interfaces must be tightly coupled to the underlying structure of the system such that automation of configuration management and versioning.

Additionally, the study posits that modern collaboration within IDEs must include the flexibility to support passive peripheral awareness of others working on the system, support audio, video, and text interfaces, integrate with current/existing source control and error reporting/tracking systems, and allow for synchronous and asynchronous communication among team members. "Eclipse" is offered as an exemplar of an open-source IDE that exhibits many of the tools in the paper [Cheng et al. 2004].

The TeamSpace project [Geyer et al. 2001] seeks to provide services beyond traditional distributed conferencing; the goal of the system is to combine synchronous distributed conferencing with captured, annotated collaborative workspace so that participants can view the materials asynchronously. The theory behind the approach is derived from cognitive psychology's "episodic memory" which states that we store and recall events based upon life experiences; leveraging from this, the system's elements are all time-sensitive in that every event and captured content is related across time. Consequently, not only can users view the content of the collaboration hierarchically according to the contents that they seek, the user can also search across time (i.e. "I remember it happened somewhere near the end of the meeting").

The system allows users to share and annotate PowerPoint presentations, agenda items (which can be "checked off" as the collaborative meeting progresses), action items (which can also be "checked off"), and provides for low-bandwidth audio and video to

create the presence and awareness of other users [Geyer et al. 2001].

Fussell et al [2000] performed a recent study to examine the importance of having presence within the collaborative environment. In the experiment, a novice attempted to construct a complex mechanical device with the assistance of an at-a-distance mentor; the participants in the study were able to share a communication channel via voice and video, establishing a "virtual physical co-presence." The study found that given complex tasks, remote users must have certain contextual cues in order for users to collaborate effectively. These "grounding" elements are:

Establishing a joint attention focus: allow users to be sure that everyone involved is viewing the same common element within the system

Monitor comprehension: use nonverbal communication and facial expressions to establish that everyone comprehends what was said/discussed

Conversational efficiency: make it as easy as possible for users to communicate their intentions (i.e. allow gestures and constrain the conversation within the context of the system).

While this study examined assembling a physical device, the findings are also applicable in a distributed environment in which collaborators must establish a shared space in which to communicate about a common task [Fussell et al. 2000].

Koch [1995] reports on a collaborative multi-user editor entitled "IRIS." While this system does utilize the near-deprecated model of a specialized/proprietary system, some interesting interface issues can be gleaned from the "IRIS" work. First is the concept of visualizing the hierarchy and structure of the document that is being shared among multiple users; this allows for users to easily identify who is currently working on each section/unit of a shared document. This "shared meta view" is central to the project's goals and is achieved admirably. Also of interest is the systems ability to integrate the functionality and interface of single-user applications; this is absolutely critical in achieving widespread adoption of any collaborative system. Finally, the "IRIS" interface provides direct communication between authors so that the can pass messages to each other for clarification (or to request an author relinquish control of a section of the document so that another user may edit it) [Koch, 1995].

There is also a considerable amount of research in the area of agents and the ability to manage the complexity and sheer volume of information that users are flooded with. *Moksha* [Ramloll and Mariani, 1999] is a collaborative filtration system that allows users to specify their interest within the shared space at various points/parameters; the filtration agent then acts as an intermediary that selectively exposes the user to only those elements of the shared, collaborative space that the user has interest in. Many collaborative development systems can benefit from this model of filtering based upon individual users' preferences, especially given the scope and size of many modern software system projects (tens of thousands of modules and millions of source lines of code).

To conclude this section, [Schur et al. 1998] enumerate five critical elements from their research that define interface issues critical to collaborative systems. Successful CSCW systems will achieve the following:

Social dialog: enables users to send and receive important concepts, thoughts,

and ideas; this also enables the creation of "place" in which the collaborators interact.

Provide framework: a collaborative environment may enable a more rapid application development (RAD) approach to accomplishing goals in that users can more rapidly cycle through their interactions and processes.

Allow rapid context switching: the interface should allow users to author and then share changes/ideas rapidly without requiring a series of complex key or button inputs (i.e. make the system unobtrusive and easily navigable).

Culture/trust dramatically affect adoption: realize that functionality along will not drive the adoption of a collaborative system – there must be an understanding by users as to what they will gain by using the new system.

Timeliness: the interface and messages within the system must occur rapidly or users will get frustrated.

11. CLASSIFICATION MODELS

If we view collaborations as primarily consisting of communication, then we can organize such communication along three variables: time, space, and modality time [Nickson, 1997]. With respect to time, users can communicate at the same time (synchronously) or at different times (asynchronously). With respect to space, users can occupy similar spaces that are close in proximity (proximal) or occupy spaces that are distant from each other geographically (distal). With respect to modality, users can communicate via text (a document-centered approach), via audio (where audio information plays an important role), and/or via video (where visual information plays an important role).

Certainly if we have more than one user interacting in the collaborative environment, then many different instances of these variables can be in play at any given. Table 2 (from [Nickson, 1997]) summarizes various applications and their features along these variables.

Nickson's taxonomy is useful to compare various CSCW systems in their support for various modalities of use. He provides numerous examples of commercial products in his paper to relate the applications and modalities to everyday products.

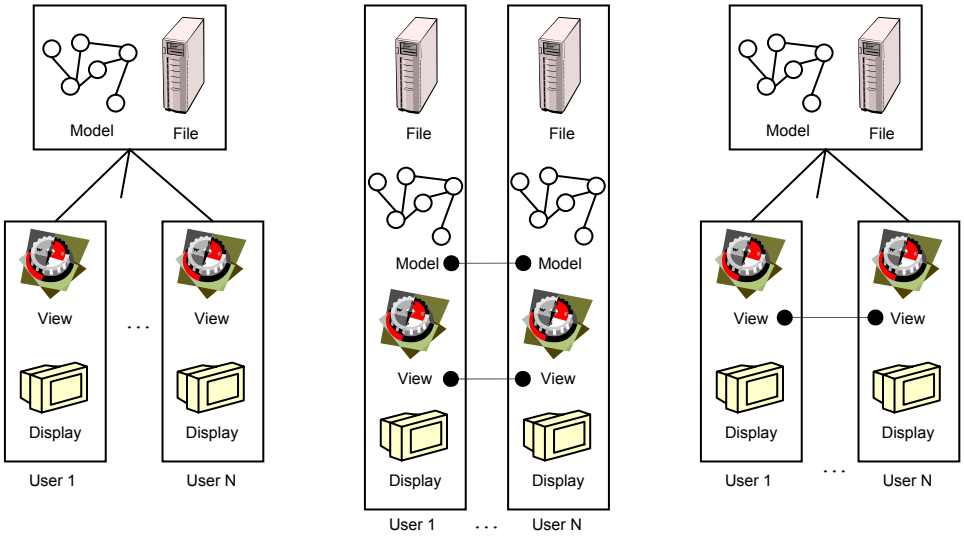
Application	Temporal		Spatial		Modal		
	Synchronous	Asynchronous	Proximal	Distal	Document	Audio	Visual
Messaging		X		X	X		
Information Sharing		X		X	X	X	X
Document conferencing	X			X	X		
Audio Conferencing	X			X		X	
Video Conferencing	X			X		X	X
Electronic Conferencing	X			X	X	X	X
Meeting Support	X		X	X	X		
Group Calendaring/Scheduling		X		X	X		
Workflow Management		X		X	X		

Table 2: CSCW Variables and Applications

Another model to define CSCW systems is Patterson's [Roth and Unger, 2000] that defines groupware into four levels: **display** (renders the application to the user), **view** (contains the application's logical presentation), **model** (the application's state and internal information), and **file** (the persistent information of the application). Based upon these four levels, three different variations can be described. The **shared model** is one in which the different users each have their own displays and views, but the model and file levels are combined in a centralized server. The **shared view** is one in which each user has a separate file, model, view, and display, but the models and views utilize communication mechanisms to ensure consistency. The **hybrid model** is one in which the file and model are centralized and shared on a server, but the system allows for different views and displays (and views are coordinated via communication to ensure consistency).

Other modern models include the window system and coordination agent/subsystem that communication to the presentation and functional core aspects of the model. Based upon this view, the system can be central (contain server that maintains all state), direct communication (a peer-to-peer system), hybrid (combination of server and peer-to-peer), asymmetrical (in which the server resides on a user's machine), and multiple servers (in which there is a hierarchy of servers and communication layers) [Roth and Unger, 2000].

Walpole et al. [1988] did work in this unifying area earlier with specific focus on software development environments (SDEs) that seek to unify version control, configuration control, and modification transactions (edits). Their model is novel in that it adopted the object-oriented view of collaborative software development systems.



A more modern object-oriented approach to CSCW systems can be found in [Teege, 1996]. He advocates a general-purpose CSCW model that integrates various aspects of collaborative work into a unified model called "Object-oriented Activity Support Model" (OOActSM). This model attempts to bring together systems supporting processes, activities, and toolkits (i.e. APIs). This model's fundamental unit is the "activity" that can be viewed as an object that consists of an "executing actor" (i.e. who initiated the activity), the "context" (what is changed and any relevant state information), and "subactivity structure" (which allows for multiple actors and contexts).

12. RECENT AND FUTURE WORK

Recent trends in collaborative editing systems have focused on mainstream document types (PDF, Word, HTML) and away from proof-of-concept/research document types. There is also a significant trend in Web authoring moving away from a one author model to a collaborative model where many authors contribute to a single document/site. Such trends suggest that a more fine-grained concurrency system be adopted in Web authoring that will allow multiple authors to modify a single file collaboratively, rather than maintain course-grain concurrency in which a single author locks an entire file. Given the highly structured nature of Web-based languages like HTML, XML, and other SGML languages, this fine-grained locking/collaboration should be easily implemented.

Recent work in the area of synchronous editing of structured documents involve defining positional addressing schemes and sets of fundamental transformational operations; given these, concurrency among users and granularity of editing can be improved [Davis et al. 2002].

Drury [2001] has developed a set of heuristics that are useful in analyzing behavior within collaborative computing systems; her heuristics are derived from metaphors from activation theory, workspace awareness, coordination theory, distributed cognition, information ecology, and team situational awareness theory. This work is significant in that it examines how users interact within CSCW systems and relates to many fields within the social sciences.

Maybury [2001] discusses the Java Collaborative Virtual Workspace that has evolved from earlier multi-user dungeon (MUD) and MUD object-oriented (MOO) communication protocols. This system has grown to over 3000 users with as many as 400 simultaneous users within the system at any given time. What is particularly interesting about this work is that it provides "rooms" in which users can collaborate, persistence of objects within the shared space, and even supports a Palm client for wireless and mobile users. Maybury claims that "we have discovered three important abstractions are central to all collaborations: conference, context, and participants." It will be interesting to see how these meta elements are present in other CSCW systems.

A recent study [Cadiz et al. 2000] of a collaborative document editing system proves that it is a vibrant field with many research opportunities. Within a 10 month period, 9239 annotations were made on 1243 documents by 450 developers for the Microsoft Office 2000 system. Clearly, large-scale software system development necessitates the ability to make annotations and collaboration. The authors of this paper note that in-context comments/annotations are

needed so that the comments are situated with respect to the element that is being annotated. Further, the authors argue that the ubiquity of the Web lends it as a natural medium by which collaboration can be established.

Microsoft's Web Document and Versioning Protocol was used in cooperation with a SQL Server database to track the changes. One interesting result of the study is that collaborators should be informed proactively when a document that interests them has been modified; this happens automatically via email.

Among regular users, the average number of annotations was 47.5; the average number of documents annotated was 10.5. Among occasional users, the average number of annotations was 9.3; the average number of documents annotated was 3.2 [Cadiz et al. 2000].

I believe that in the near future, we will see voice over IP (VOIP) emerge as a ubiquitous aspect of any modern collaborative environment; given that VOIP is becoming more pervasive (witness DirectX 9.0 that contains a built-in API for VOIP), collaborative systems should also begin to make use of this important communication aspect of how people naturally work.

ACKNOWLEDGMENTS

Many thanks to Dr. Prasad for the opportunity to explore this topic and study parallel computing.

REFERENCES

BEGOLE J., ROSSON M. B., and SHAFFER C. A. *Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems*. ACM Transactions on Computer-Human Interactions, vol. 6, no. 2, pp. 95-132, June 1999.

BEGOLE J., ROSSON M. B., and SHAFFER C. A. *Supporting Worker Independence in Collaboration Transparency*. In Proceedings of UIST'98, San Francisco CA, pp. 133-142, 1998.

BEGOLE J., SMITH R. B., STRUBLE C. A. and SHAFFER C. A. *Resource Sharing for Replicated Synchronous Groupware*. IEEE/ACM Transactions on Networking. vol. 9, no. 6, pp. 833-843, December 2001.

BENFORD S., GREENHALGH C., RODDEN T., and PYCOCK J. *Collaborative Virtual Environments*. Communications of the ACM, vol. 44, no. 7, pp. 79-85, July 2001.

BORGHOFF U. and TEEGE G. *Application of Collaborative Editing to Software-Engineering Projects*. ACM SIGSOFT, 18(3), pp. 56-64, July 1993.

BOYER D. G., HANDEL M. J., and HERBSLEB J. *Virtual Community Presence Awareness*. SIGGROUP Bulletin, vol. 19, no. 3, pp 11-14, December 1998.

BULGANNAWAR S. and VAIDYA N. *A Distributed K-Mutual Exclusion Algorithm*. International Conference on Distributed Computing Systems, pp. 153-160, 1995.

CADIZ J., GUPTA A., GRUDIN J. *Using Web Annotations for Asynchronous Collaboration Around Documents*. In Proceedings of CSCW'00, Philadelphia PA, pp. 309-318, December 2000.

CHAWATHE Y., MCCANNE S., and BREWER E. *RMX: Reliable Multicast in Heterogeneous Networks*. In Proc. IEEE INFOCOM, March 2000.

CHENG L. et al. *Building Collaboration into IDEs*. ACM Queue. December/January 2003-2004. 40-50.

CHENG L. et al. *Jazz: A Collaborative Application Development Environment*. In Proceedings of OOPSLA'03, Anaheim CA, 102-103, 2003.

CHOCKLER G. V., KEIDAR I., and VITENBERG R. *Group Communication Specifications: A Comprehensive Study*. ACM Computing Surveys, vol. 33, no. 4, pp. 427-469, December 2001.

CONRADI R. and WESTFECHTEL B. *Version Models for Software Configuration Management*. ACM Computing Surveys, vol. 30, no. 2, pp. 232-282, June 1998.

DAVIS A. H., SUN C., and LU J. *Generalizing Operational Transformation to the Standard General Markup Language*. In Proceedings of CSCW'02, New Orleans Louisiana, pp. 58-67, November 2002.

DECOUCHANT D., QUINT V, and SALCEDO M. R. *Structured and Distributed Cooperative Editing in a Large Scale Network.*

DRURY J. *Developing Heuristics for Synchronous Collaborative Systems*. In Proceedings of CHI'2001, pp. 447-448, March/April 2001.

EDWARDS W. K. et al. *Using Speakeasy for Ad Hoc Peer-to-Peer Collaboration*. In Proceedings of CSCW'02, New Orleans LA, pp. 256-265, November 2002.

FUSSELL S. R., KRAUT R. E., and SIEGEL J. *Coordination of Communication: Effects of Shared Visual Context on Collaborative Work*. In Proceedings of CSCW'00, Philadelphia PA, pp. 21-30, December 2000.

FUSSELL S. R., et al. *Assessing the Value of a Cursor Pointing Device for Remote Collaboration on Physical Tasks*. In Proceedings of CHI'2003: New Horizons, Ft. Lauderdale FL, pp. 788-789, April 2003.

GEYER W. et al. *A Team Collaboration Space Supporting Capture and Access of Virtual Meetings*. In Proceedings of GROUP'01, Boulder CO, pp. 188-197, September 2001.

GEYER W., VOGEL J., CHENG L., and MULLER M. *Supporting Activity-centric Collaboration through Peer-to-Peer Shared Objects*. In Proceedings of GROUP'03, Sanibel Island FL, pp. 115-124, November 2003.

GRINTER R. E. *Recomposition: Putting It all Back Together Again*. In Proceedings of CSCW'98, Seattle WA, pp. 393-402, 1998.

GRUDIN J. *CSCW Introduction*. Communications of the ACM, vol. 34, no. 12, pp. 30-34, December 1991.

HAO M. C., KARP A. H, and GARFINKEL D. *Collaborative Computing: A Multi-Client Multi-Server Environment*. In Proceedings of COOCS'95, Milpitas CA, pp. 206-213, August 1995.

HARRISON W. H., OSSHER H., and SWEENEY P. F. *Coordinating Concurrent Development*. In Proceedings of CSCW'90, 157-168, October 1990.

HARTUNG J. et al. *A Real-Time Scalable Software Video Codec for Collaborative Applications Over Packet Networks*. In Proceedings of ACM Multimedia '98, Bristol UK, pp. 419-426, 1998.

HELLENBRAND C. *Defining the Influence of Collaborative Computing on Organizational Activity Governance*. In Proceedings of SIGCPR'95, Nashville TN, pp. 239, 1995.

HERBSLEB J. D. et al. *Distance, Dependencies, and Delay in Global Collaboration*. In Proceedings of CSCW'00, Philadelphia PA, pp. 319-328, December 2000.

HORSTMANN T. and BENTLEY R. *Distributed Authoring on the Web with the BSCW Shared Workspace System*. StandardView, vol. 5, no. 1, pp. 9-16, March 1997.

KAISER G. E. and KAPLAN S. M. *Parallel and Distributed Incremental Attribute Evaluation Algorithms for Multiuser Software Development Environments*. ACM Transactions on Software Engineering and Methodology, vol. 2, no. 1, pp. 47-92, January 1993.

KOCK M. *The Collaborative Multi-User Editor Project IRIS*, Technical Report TUM-I9524, University of Munich, Aug. 1995.

KOREL B. et al. *Version Management in Distributed Network Environment*. In Proceedings of the 3rd International Workshop on Software Configuration Management, pp. 161-166, May 1991.

LI D. and LI R. *Transparent Sharing and Interoperation of Heterogeneous Single-User Applications*. In Proceedings of CSCW'02, New Orleans LA, pp. 246-255, November 2002.

LI D. and PATRAO J. *Demonstrational Customization of a Shared Whiteboard to Support User-Defined Semantic Relationships among Objects*. In Proceedings of GROUP'01, Boulder CO, pp. 97-106, October 2001.

LOCASTO M. et al. *CLAY: Synchronous Collaborative Interactive Environment*. The Journal of Computing in Small Colleges, vol. 17, issue 6, pp. 278-281, May 2002.

LORY G. et al. *Microsoft Solutions Framework version 3.0 Overview*. Microsoft Press. http://www.microsoft.com/msf. June 2003.

MAGNUSSON B., ASKLUND U., and MINÖR S. *Fine-Grained Revision Control for Collaborative Software Development*. In Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering, vol. 18, issue 5, pp. 33-41, December 1993.

MARQUÈS J. M. and NAVARRO L. *WWG: a Wide-Area Infrastructure to Support Groups*. In Proceedings of GROUP'01, Boulder CO, pp. 179-187, September 2001.

MAYBURY M. *Collaborative Virtual Environments for Analysis and Decision Support*. Communications of the ACM, vol. 44, no. 12, December 2001. pp. 50-54.

MCCANNE S. et al. *MASH: Enabling Scalable Multiport Collaboration*.

ACM Computing Surveys. vol 31, issue 2es, article no. 2. 1999.

NICKSON R. C. *A Taxonomy of Collaborative Applications*. http://hsb.baylor.edu/ramsower/ais.ac.97/papers/nickers.htm.

OLSON J. et al. *Surprises in Remote Software Development Teams*. ACM Queue. December/January 2003-2004. 52-59.

PENSERINI L., PANTI M., SPALAZZI L. *Agent-Based Transactions in Decentralised P2P*. In Proceedings of AAMAS'02, Bologna Italy, pp. 1288-1289, July 2002.

PERRY D. E., SIY H. P., and VOTTA L. G. *Parallel Changes in Large Scale Software Development: An Observational Case Study*. In Proceedings of the 20th International conference on Software Engineering, pp. 251-260, April 1998.

PERRY D. E., SIY H. P., and VOTTA L. G. *Parallel Changes in Large Scale Software Development: An Observational Case Study*. ACM Transactions on Software Engineering and Methodology, vol. 10, no. 3, pp. 308-337, July 2001.

RAMAN S. *Thesis: A Framework for Interactive Multicast Data Transport in the Internet*. Computer Science at the University of California at Berkeley. 2000.

RAMLOLL R. and MARIANI J. A. *Moksha: Exploring Ubiquity in Event Filtration-Control at the Multi-user Desktop*. In Proceedings of WACC'99, San Francisco CA, pp. 207-216, February 1999.

ROTH J. and UNGER C. *An extensible classification model for distribution architectures of synchronous groupware*. 4th International Conference on Cooperative Systems. 2000.

ROTH, J. and UNGER C. *Developing synchronous collaborative applications with TeamComponents*. 4th International Conference on Cooperative Systems. 2000.

SARMA A., NOROOZI Z., and VAN DER HOEK A. *Palantír: Raising Awareness among Configuration Management Workspaces*. Proceedings of the 25th international conference on Software engineering, Portland OR, pp. 444-454, May 2003.

SCHUR A. et al. *Collaborative Suites for Experiment-Oriented Scientific Research*. interactions..., pp. 40-47, May/June 1998.

SHANDS D., JACOBS J., YEE R., and SEBES E. J. *Secure Virtual Enclaves: Supporting Coalition Use of Distributed Application Technologies*. ACM Transactions on Information and System Security, vol. 4, no. 2, pp. 103-133, May 2001.

SHEN H. and SUN C. *Flexible Notification for Collaborative Systems*. In Proceedings of CSCW'02, New Orleans Louisiana, pp. 77-86, November 2002.

SOMMERVILLE, I. *Software Enginerring 6th Edition*. Addison Wesley, Harlow, England. 2001. pp. 4-17.

STEIN M., RIEDL J., HARNER S., and MASHAYEKHI V. *A Case Study of Distributed, Asynchronous Software Inspection*. In Proceedings of ISCE'97, Boston MA, pp. 107-117, 1997.

SUN C. *Undo as Concurrent Inverse in Group Editors*. ACM Transactions on Computer-Human Interaction, vol. 9, no. 4, pp. 309-361, December 2002.

SUN C. and CHEN D. *Consistency Maintenance in Real-Time Collaborative Graphics Editing Systems*. ACM Transactions on Computer-Human Interaction, vol. 9, no. 1, pp. 1-41, March 2002.

SUNDERAM V. et al. *CCF: Collaborative Computing Frameworks*. SC'98: High Performance Networking and Computing Conference (Orlando, Florida USA). IEEE. 1998.

TEEGE, G. *Object-Oriented Activity Support: A Model for Integrated CSCW Systems*. Computer Supported Cooperative Work (CSCW): The Journal of Collaborative Computing, 5(1), pp. 93-124, 1996.

TEEGE G. and BORGHOFF U. W. *Combining Asynchronous and Synchronous Collaborative Systems*. In Proceedings of the 5th International conference on Human-Computer Interaction, Amsterdam Netherlands, pp. 516-521, 1993.

TOWNSEND A. M., HENDRICKSON A. R., and DEMARIE S. M. *Meeting the Virtual Work Imperative*. Communications of the ACM, vol. 45, no. 1, pp. 23-26, January 2002.

VAN DER HOEK A., HEIMBIGNER D., and WOLF A. L. *A Generic, Peer-to-Peer Repository for Distributed Configuration Management*. Proceedings of the 18th international conference on Software Engineering, pp. 308-317, May 1996.

VELAZQUEZ M. *A Survey of Distributed Mutual Exclusion Algorithms*. Colorado State University Department of Computer Science Technical Report CS-93-116, September 1993.

VIDOT N. et al. *Copies convergence in a distributed real-time collaborative environment*. In Proceedings of CSCW'00, Philadelphia PA, pp. 171-180, December 2000.

WALPOLE J. et al. *A Unifying Model for Consistent Distributed Software Development Environments*. In Proceedings of the third ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments, pp. 183-190, January 1989.

WALTER J. et al. *A K-Mutual Exclusion Algorithm for Wireless Ad Hoc Networks*. Principles of Mobile Computing '01. Newport, Rhode Island USA. 2001.

WALTER J. et al. *A Mutual Exclusion Algorithm for Ad Hoc Mobile Networks*. Wireless Networks, vol. 7, no. 6, pp. 585-600, 2001.

WANG X., BU J., and CHEN C. *Achieving Undo in Bitmap-based Collaborative Graphics Editing Systems*. In Proceedings of CSCW'02, New Orleans Louisiana, pp. 68-76, November 2002.

WU D. and SARMA R. *Dynamic Segmentation and Incremental Editing of Boundary Representations in a Collaborative Design Environment*. Proceedings of the sixth ACM symposium on Solid Modeling and Applications, Ann Arbor Michigan, pp. 289-300, May 2001.

YUNZHANG P. et al. *Totally Ordered Reliable Multicast for Whiteboard Application*. In Proceedings of the 4th International Workshop on CSCW in Design, Compigne France, 1999.