Computer Supported Cooperative Work: The Journal of Collaborative Computing **5:** 447–465, 1996.

Supporting Articulation Work Using Software Configuration Management Systems

REBECCA E. GRINTER*
Bell Labs, 1Q-327, 1000 E. Warrenville Road, Naperville, IL 60566, U.S.A.
E-mail: beki@research.bell-labs.com

(Received 14 September 1995; in final form 23 October 1996)

Abstract. Software product development is a highly collaborative activity, where teams of developers need to collaborate to produce a system. It is also a domain where systems are used to try to help the developers coordinate their work. This paper describes the results of an empirical study of the use of one such system, a configuration management tool. Specifically it describes three aspects of the support that the tool provides: the challenges of representing the work, the need to support both individuals and groups working together, and how the assumptions about software development built into the tool interact with others in the organization. The study suggests that long after the initial adoption the tool and the organization continue to interact with each other. It also opens up questions for empirical studies of the organizational context behind the tool usage.

Key words: Configuration management (CM), computer-supported cooperative work (CSCW), empirical studies, articulation work, coordination mechanisms

1. Introduction

Empirical studies of groupware technologies hold the promise of helping us to understand the reasons why systems fail in practice and some of the ways that we can make them work (Grudin, 1989; Okamura et al., 1994; Grudin and Palen, 1995). One challenge that researchers interested in empirical studies face is finding organizations using groupware technologies. As more groupware technologies become commercially available a number of studies have begun to examine the how groupware works in practice (Orlikowski, 1992; Bowers, 1994). Many of the empirical studies to date have focused on the initial adoption and use of these groupware technologies among small groups of users. It is harder to find groups of users who have been working with groupware for such a long time that the systems have become commonplace in their work.

Software product development provides a promising domain to find groupware-like systems that have been in use for some time for several reasons. First, most currently available commercial software packages were built collaborative by teams of developers. Second, development organizations already have the necessary technical infrastructure to support groupware systems. Third, developers have a good

* This work was supported by the Engineering and Physical Sciences Research Council, United Kingdom. This work was conducted while at the University of California, Irvine.

base of technical skills and an inclination towards learning and adopting computer software. Fourth, academic researchers and commercial organizations have tried to support the development process by building tools to support the work since the mid-1980s.

This paper describes an empirical analysis of one technology that attempts to support collaborative software development work: configuration management (CM) systems. It examines the coordination mechanisms that the tool provides and how they interact with the work of software development. The paper begins with an overview of CM systems, their groupware-like properties, the organizations studied and the methods used to gather and analyze the data collected. Then I introduce the three aspects of the support that the tool provides: the challenges of representing the work, the need to support both individuals and groups working together, and how the assumptions about software development built into the tool interact with others in the organization. The following section discusses the implications that these observations have for CSCW research.

2. Configuration management and coordination work

2.1. Managing the evolution of software

In the last ten years software product development has changed from a proprietary to an open systems industry. This transformation has influenced how software development organizations design products. Many software companies used to control all the design variables; specifically, they designed the hardware, the communications protocols, the operating system as well as their application. Nowadays, most software development companies must design families of software applications that provide the same functionality but operate with a myriad of platforms, operating systems and supporting technologies, such as databases, built by other vendors. For product software developers, designing means developing a family of applications in a continually changing environment.

Most software product organizations find it hard to keep their development environment organized. Questions about the products being built often come up: which piece of functionality belongs to what release, which platform requires a certain piece of code, what part of the documentation needs altering to make it compatible with this release, and how can the variants be tracked. To maintain control over their development environment many product development companies are using CM systems that provide support for developers working on the family of applications.

CM systems control the development of these families and attempt to reduce the inherent difficulties in managing evolving software. Software is hard to manage for three reasons.* First, developers can easily change code. Second, the modifications can affect the behavior of the entire system because of the interdependencies among

* For a complete treatment of the complexities of software development, see Brooks (1987).

modules. Third, because teams develop software, the changes one person makes often impact the work of others. CM systems aim to support the evolution of software by coordinating the efforts of multiple developers working on a family of closely related products.

First generation CM tools used a library metaphor of 'checked-out' and 'checked-in' states to control changes to software. To make any modifications to a software module, developers had to check out the code. When a developer checked a module out, the tool made a new version of the code and prevented others from checking out the same software. When changes had been completed, the developer checked in the code. A checked-in module was stable and usually working. Other developers could read and execute it with their own modules. By checking-out and checking-in code, developers created successive versions of the module that the system stored. Code versioning created stability during development by facilitating backtracking to older versions if necessary and preventing developers from overwriting the work of others.

However, first generation CM tools had two disadvantages. First, they only worked for code; however, software systems contain more than just code, including: libraries, test suites, make files, and documents. Modern CM systems use a database to store all the artifacts that make up a software product. Second, the checked-out state turned out to be very limiting because it prevented others from changing the same module at the same time, which slowed down developers' ability to get their work done. Modern systems solve this problem by allowing two or more developers to work on the same module at the same time and then merge their changes together.

Modern CM tools also support three other layers of functionality on top of the check-out/check-in model (Caballero, 1994). The configuration control layer maintains information about the artifacts that form a software product. It knows which versions comprise a specific system and how they relate to each other. This layer allows developers to pull together all the software artifacts that comprise a specific variant of the software. It also lets developers recreate both previous and current releases of any software stored inside the CM data repository.

The process management layer provides a 'life cycle' for each type of artifact stored in the system. A life cycle consists of a number of states. For example a typical life cycle for a software module consists of the checked-out, checked-in, quality-tested, and released states. While the developers are most concerned with the checked-out and checked-in states, testers of the software use the quality-tested state to signal that a particular version of a software module has passed rigorous system testing.

Finally, the problem reporting layer supports bug and enhancement tracking. Modifications to the artifacts in the system occur as a result of problems with the function of the system or enhancements requested for future products. The problem reporting layer provides a way of linking the bugs or enhancements to the changes themselves. Modern CM tools either have built in process management and problem reporting, or provide the necessary connections to allow users to build

it themselves or purchase another off-the-shelf system and integrate it into the CM tool.

2.2. Configuration management and articulation work

Each of the layers of a modern CM system supports the coordination of design activities. The check-out/check-inlayer coordinates the day-to-day work of developers as they develop modules. The configuration control layer allows developers and managers to routinely gather the work of the entire development team into one product. The process layer synchronizes the activities of various groups involved in design, such as quality assurance and development. Finally, the problem tracking layer coordinates the definition of problems with the actual changes made to the code itself. As Babich (1986) says:

> Controlled evolution means that you not only understand what you have when you are delivering it, but you also understand what you have while you are developing it. Control helps to obtain maximum productivity with minimal confusion when a group of programmers is working together on a common piece of software. (Babich, 1986 p. vi)

One stream of research has characterized the coordination of work as articulation work (Strauss, 1985, 1988). Articulation work is all the coordinating and negotiating necessary to get the work at hand done. Software developers primarily work on designing and building software systems. However, as Bendifallah and Scacchi (1987) point out, as software developers design and build software they must also engage in forms of articulation work. CM systems attempt to support some of this articulation work electronically and can be thought of as a form of groupware.

Schmidt and Bannon (1992) have applied the concept of articulation work to the research problems in the computer supported cooperative work (CSCW) community. They describe how individuals engage in articulation work as part of their daily routines. They say:

> However in 'real world' cooperative work settings ... the various forms of everyday social interaction are quite insufficient. Hence articulation work becomes extremely complex and demanding. In the sesettings, people apply various mechanisms of interaction so as to reduce the complexity and, hence, the overhead cost of articulation work ... These protocols, formal structures, plans, procedures, and schemes can be conceived of as mechanisms ... And they are mechanisms of interaction in the sense that they reduce the complexity of articulating cooperative work. (Schmidt and Bannon, 1992, pp. 18–19, italics in original)*

* Since this paper Simone et al. (1995) have further defined mechanisms of interaction for articulation work as coordination mechanisms.

Examples of these coordination mechanisms include plans, and standard operating procedures. These mechanisms supplement other forms of social interaction like e-mail, video-conferencing, and other forms of communication.

Researchers and practitioners interested in the problems of managing software projects started to develop computer systems to support configuration management in the 1980s. Two of the earlier systems were the Revision Control System (RCS) and the Domain Software Engineering Environment (DSEE) (Lubkin, 1991; Tichy, 1985). Since that time several other systems have been developed as research projects and commercial ventures (Dart, 1992).

The developers of these systems did not build systems that would increase the communications bandwidth, such as e-mail, for two reasons. First, in larger development teams communication cannot support all the articulation work necessary to coordinate the efforts of multiple individuals. Second, coming from the software engineering community, configuration management specialists are used to, and comfortable with, formal approaches to resolving coordination problems (Pickering and Grinter, 1995). Instead they embedded coordination mechanisms into a CM tool.

CM tools provide an opportunity to examine how well these computerized coordination mechanisms actually support software development in practice. In this study I examine three aspects of coordination mechanisms: the difficulties of representing work, the need for different levels of coordination mechanism, and how these mechanisms provide a model of software development. For each aspect I examine both the benefits and challenges of using coordination mechanisms to support collaborative work.

3. Organizational settings and methods

Grudin (1991) describes three kinds of software development context: product, contract and in-house development. In this paper I focus exclusively on product development companies. The two sites I studied compete in different markets, but they share characteristics of software product development organizations.

The first site in my study was a development division of one CM tool vendor that I call 'Tool Corporation', that competes in an oligopoly for this market. Specifically I studied how the developers responsible for building the CM tool use their CM tool to manage their work. The organization was growing rapidly, approximately 200% per year, and during the course of my study the development group grew from 14 to 18 people. The developers use the tool in their daily work, to build the next version of the tool itself.*

* Obviously studying expert users of the CM tool affects the conclusions that I can draw, but it also offers several advantages. First, by studying a group of experts who had used the system for some time I did not find adoption problems. Second, there have been few studies of long-time groupware systems usage. Finally, there have been even fewer studies of systems that support coordination mechanisms rather than enhancing communications among developers.

At Tool Corp. I conducted a three and a half month on-site interpretive study of the company in mid-1994. I adopted participant and non-participant observation, as well as interviewing strategies to collect data (see Jorgenson, 1989). My participant observation included: helping with development activities, including usability testing, multiple user testing, reviewing documentation, and attending meetings. I also had full access to the development environment created by the CM tool so I could watch the work in progress. I used two interviewing strategies, informal interviewing and semi-structured interviewing. The interviews lasted anywhere from 20 minutes to 2 hours. The semi-structured interviews were taped and transcribed and the informal interviews were written up after they took place. In total, 20 semi-structured interviews and approximately 80 informal interviews took place. Supplemental material was gathered by reading journals, reports, electronic discussion lists, and documents.

The second site, 'Computer Corporation', has approximately 700 developers building computer software. The company was in the process from switching from their existing CM system to the one sold by Tool Corp. They were also revising their entire CM strategy to handle the challenges of 'open systems' development. In this study I conducted a series of semi-structured interviews with managers, configuration managers, and developers. All the developers that I spoke to had used the new tool, and many of them had used the old tool prior to that, or some other CM system. The semi-structured interviews, assured some overlap with the issues that I discovered at Tool Corp. The interviews themselves lasted from 30 minutes to an hour. Over the course of two days I conducted a total of 13 interviews, and attended a class designed to introduce developers to the new system. I also gathered public documents about the company.

I began by analyzing the data gathered from Tool Corp. Initially I concentrated on understanding how the developers used the CM tool to coordinate their work. The initial informal interviews helped detail and later confirm usage patterns. At the same time, the distinction between CM tools and more traditional forms of groupware such as e-mail and video-conferencing started to emerge, that led me towards a conception of the CM tool supporting coordination mechanisms, rather than communication. This observation led me to develop the interview guide for the semi-formal interviews. In these interviews I encouraged developers to discuss their use of the CM tool and what they do when the tool does not support their work. The final stages of data gathering and analysis focused heavily on expanding the ideas. I used the data from Computer Corp. to develop the concepts that emerged from the study at Tool Corp.

4. Coordination mechanisms in software development

4.1. Merging software

Developers call the times when more than one person has the same module checked out, 'parallel development'. This happens when different developers have changes

Figure 1. The time-line view of an artifact.

that require them to work on the same module at the same time. When the developers have finished their work, the tool provides a merge facility that allows them to combine their changes into a single module. Although the tool supports parallel development activities, developers at both sites sometimes found merging software difficult.

The CM tool provides a coordination mechanism that informs developers whether they are working in parallel. The tool maintains a time-line view of the evolution of every artifact in the data repository (see Figure 1). The time line shows the history of an artifact's development as a series of boxes and lines that chart its evolution over time. Each box represents a version of the artifact that corresponds to a time in the development of the artifact. The boxes show the name of the artifact (*a.c*), the version number (1, 2, and 3), the person who worked on the artifact (Liz, and Tim), and the state of development then (checked in, and checked out).

Developers use the time-line views of modules to find out whether anyone else is currently working on the code they need to alter. In the time line view shown in Figure 1, Tim has the latest version of *a.c* checked out for changes. All the developers working on this project can also see that Tim has the module checked out, because they have access to the same time-line view. This allows them to make decisions about whether they want to engage in parallel development. Often if developers see that someone has the latest version checked out, they either ask the person working on it to incorporate their changes into that version, or try to work on some other task.

However, sometimes the developers cannot avoid parallel development. Their changes may be too complex to ask another person to work on, or they may be too critical to postpone until parallel development can be avoided. So the developers check another version of the module out. At this point, even if they have looked at the view, the system flags them with a message telling them that they have made a parallel version.

When the developers have completed their changes they usually have to merge their code with the changes made by the other person.* The person who finished last takes responsibility for merging their work with the other person's. This was the same at both sites because the tool was designed so that the last to finish always had to merge their code into the others' work. The tool supports merging by providing a facility that compares the two files and displays the lines that differ.

* Some configuration management systems use an automatic merge facility instead of making developers select when there are conflicts.

The developer responsible for merging selects the lines that need to appear in the integrated module.

Merging can be easy when the developers have changed different parts of the module, for example if someone has changed the comments and another person has altered the functionality. Developers find cases such as this easy because the changes involve distinct parts of the module and that shows up clearly in the merge display. In these easy cases the developer simply merges the modules without consulting anyone.

However, sometimes merging becomes too difficult for a developer to do without communicating with the other person who worked on the module. The times when this happens usually occur when both the developers have modified the same lines of code or algorithm. When this happens the complexity of merging rises because suddenly differences become embedded in the context of how a module works, what problems and enhancements the developers were working on, and which solution developers chose to implement.

At this point the developer responsible for merging finds the other person who also modified the module. They discuss what they did, explaining their programming strategies, the problems they solved, and the functionality that they believe the module possesses. They work together to develop a shared understanding of both modules, and determine the functionality of the merged module. This activity often takes place as a joint merging effort. The developers sit around one terminal and select the lines that should go into the final merged module. When this kind of interaction does not take place the resulting merge is often erroneous, perhaps even causing the system to break.

4.2. Mechanisms for individuals and mechanisms for groups

The coordination mechanisms embedded in the CM tool help developers work together by providing visibility into the work of others. The time-line view allows developers at both Tool Corp. and Computer Corp. to 'see' whether another developer has a module checked out for development. The tool also provides information about the status of the artifacts in the project. The 'work view' shows a set of artifacts, giving the name of the artifact, which developer has most recently completed work on it, and the state of the artifact in the development life cycle.

In order to keep the work view current developers 'update' this view. An update causes the system to refresh the view displaying the latest versions of stable software, potentially revealing changes in an artifact's name, a new developer working on an artifact, or a change in the state of an artifact. Updating the view provides important information to developers because of the changes it shows. For example, if a developer sees that a new version of a related module has been checked in they know their software must work with that code. This view enables developers to continually coordinate their efforts as they alter related modules simultaneously. Often developers find after updating their view that their module does not work

with some of the latest changes made to related modules and they must fix their work.

However, developers at both sites required other ways of coordinating their work, which the tool did not supply. At Tool Corp., the 14 developers working on the project sometimes talked about a lack of software architecture for the product being developed. Software architectures:

> permit designers to describe complex systems using abstractions that make the overall system intelligible. Moreover, they provide significant semantic content that informs others about the kinds of properties that the system will have: the expected paths of evolution, its overall computational paradigm, and its relationship to similar systems. (Garlan and Perry, 1994, p. 363)

For the software developers at Tool Corp. the software architecture represented potential ways to help coordinate work at a higher level of abstraction. While the developers understood their own parts of the system well, they did not find it so easy to create the software architecture from those subsystems. This problem was exacerbated by the continual changes in design that altered the architecture and which occurred throughout the product development life cycle. As a group of 14, the developers were unable to establish and maintain a sense of architecture through discussion and the system provided no coordination mechanisms for showing developer show their individual work fitted into the bigger picture. Without them, and unable to communicate the vision, these developers felt that they could not fully understand the work of developers who worked on sections of the system remote, but possibly related, to their own.

While developers at Tool Corp. had an intuition about the utility of employing the architectural level of abstraction to coordinate their work, the employees of Computer Corp. fully understood the necessity of this higher level for managing the dependency relationships in their software. Dependencies arise when one component relies on another. For example, one module (A) may rely on a function in another piece of code (B). In order for B to function as intended, A must always be working correctly. In software development, dependencies may arise at run time, compile time, or build time. At Computer Corp. the developers routinely dealt with all three kinds of dependency.*

Often software has dependencies that are contained inside one subsystem. These local dependencies were managed at Computer Corp. by good communications. Typically developers knew everyone working on their subsystem, because they were co-located, and in the same development team. Despite their familiarity with each other, and knowledge of the subsystem, occasionally dependencies went unnoticed, causing the system to break or behave in unexpected or undesirable ways. People at Computer Corp. hoped that the tool would help them to see these dependencies. At Computer Corp. developers and managers also encountered enterprise-wide

* Compile-time dependencies occur when a sub-system is being compiled. Build-time dependencies occur when several sub-systems or the entire system is being compiled. Run-time dependencies occur when the executable is running.

dependencies. Many of the subsystems at Computer Corp. are small products in themselves. Due, in part, to the architecture of the entire system, subsystems could have dependencies on other subsystems. Some of these subsystems turned out to be critical systems, because several other subsystems depended on them functioning a certain way. As a consequence of these dependencies, changes to the critical systems impacted development in these dependent systems.

Enterprise-wide dependencies created additional problems for developers and managers at Computer Corp. because many of the strategies that worked for local dependencies did not scale up to enterprise-wide ones. People working on the critical subsystems were not always aware that other teams depended on their system. Informal communications proved difficult, as usually the two teams worked in different buildings and often reported to separate divisions of the organization.

Computer Corp. did have some strategies to handle enterprise-wide dependencies. The organization had various committees tasked with coordinating changes between remote groups with dependency relationships. As well as using committees, Computer Corp. had several organizational units who were in part concerned with finding and resolving enterprise-wide dependencies in the course of their own work. However, despite having these coordination mechanisms in place, enterprise wide dependency problems arose and when they did they created problems for many developers and managers.

4.3. Coordination mechanisms and work

The coordination mechanisms embedded in the CM tool that Tool Corp. and Computer Corp. use support a 'model' of software development. The model is not unified and it does not define everything that developers should do. At points in the development life cycle the tool makes certain choices about how to proceed and at the same time it closes off other potential avenues of action. These choices are based on certain assumptions about how software development should proceed, and have been embedded in the tool and its coordination mechanisms. However, the model provided by the tool must interact and compete with other models of software development generated by the organization, in policies and practices, or by the developers themselves.

Earlier I described two simple coordination mechanisms, the time line and update views. Both of these views give developers certain information about development allowing them to make choices about how to continue with their own work. The developers generally liked these coordination mechanisms, because they created a certain visibility into the history and current state of development. In this sense the model of work that the tool provides, informational, works well with other models that suggest that software developers should make well-informed choices about their work, and models that say that avoiding parallel development saves time.

However, not all of the coordination mechanisms seem to blend so well with these other ways of thinking about software development. When this happens the tool cannot change its structure so the developers make personal choices to either bend or break the model. This happened at both Tool Corp. and Computer Corp.

Developers at Tool Corp. have special privileges to create and assign themselves problems using the problem reporting facility. This was not the intention of the tool that had a concept of a managerial group entering problems and developers responding to them. However, at Tool Corp. the group tasked with entering and assigning problems in the system only met every couple of days. Meanwhile if the developers finished their assignments before the next meeting then they would be unable to start any new projects because the system would not accept revisions to modules without an associated problem. Therefore the developers used system privileges reserved for systems administrators and worked around the constraints imposed by that model of software development work.

While the rapid development times formed the main reason for giving developers these problem reporting privileges, they had an unintended payoff for managers. Often developers needed to make small changes to one artifact and so they created and assigned problems to themselves. Managers benefited because they did not have to read through and assign all these small problems leaving them to concentrate on the major system problems. This work around was given further legitimacy within Tool Corp. because the most senior manager in charge of development took the view that the developers were capable of deciding whether something was a problem, and trusted them to take the initiative to work on addressing key problem areas.

In another case a few developers violated the procedures imposed by the system. One example of this related to the difficulties of testing software. Testing software requires developers to run numerous tests that not only assure the integrity of the artifact, but there liability of its interactions with other artifacts. Developers have to test all the possible interactions a module has with other artifacts, which means generating tests for all the permutations of run-time behavior. Sometimes after checking a module in, developers realize that their code may not work when a certain sequence of actions occurs.

If the developers discover that they need to make further changes they have two options: check out another version of the code, create a problem to assign to that code, and fix the problem, or cheat the system. Often developers go to the trouble of making another version, but occasionally they edit the artifact in the checked-instate. They do this despite knowing that the changes they make in the checked-in state might affect the work of other developers who assume that checked-in files do not change.

At Computer Corp. the developers are still sufficiently new to the tool that they have other problems that developers at Tool Corp. have resolved either at a personal or organizational level. Many of the problems the developers at Computer Corp. described were related to adoption; specifically, trying to align their practices with

the tool. However, one problem not only exemplified the difficulties of adoption, but uncovered other places where developers get models of what constitutes good software development practices.

As I described earlier, when developers engage in parallel development the tool provides the developers with a merge tool that allows the developer that finishes last to merge their work with the other developers' code. The model supplied by the merge coordination mechanisms creates a dilemma for the software developers. On the one hand they want to rush and finish their work first, to avoid merging; on the other hand, they want to produce quality code. As one developer explained to me, the tool broke a convention about what it takes to produce good software. Citing both the ISO 9000 standard for quality in software development and the Software Engineering Institutes Capability Maturity Model (CMM) he explained that the tool seemed to encourage developers to rush to finish their work rather than spend the time ensuring that it was quality work.

5. Discussion

5.1. Merging and the question of representation

Developers at both sites explained that they often try to avoid parallel development because of the potential hazards of merging. The difficulty of coordinating the efforts of multiple developers in a single module cuts into their development time. The mechanism of interaction that formalizes merging tries to eliminate some of these difficulties, but clearly it breaks down when the developers make changes that interact with those made by their co-workers.

Merging highlights the challenges of representing work. In his study of navigation, Hutchins (1990) described how a team of ship navigators worked together to guide vessels into harbor. Part of his study examined the role of navigational instruments. Specifically, he discussed how the instruments provide representations of the navigation problems the team faced. He argued that these instruments supported the work of navigators because of the way that they represent the problems. Much navigation work involves mathematical relations and instruments that have these formulae built into them allow navigators to find the solutions easily. In this way the instruments reduce the complexity of the problems of navigation work.

The CM system provides coordination mechanisms that try to create visibility into the merging process. However this particular case highlights the deceptive nature of thinking about merging as a 'task'. Sometimes merging goes smoothly, and the tool provides an adequate representation of the merging task with its graphics capabilities. However, when merging gets more complex, the 'task' changes and the tool can no longer support the articulation work required to produce a combined module. It is tempting the believe that the developers at Computer Corp. found merging difficult because they were not used to the tool. However, even experienced developers at Tool Corp. who had used the system for two years had

problems merging software. This suggests that to understand why merging is hard we must move beyond adoption explanations.

In a study of CSCW tools and concepts, Robinson (1991) makes a distinction between the formal and cultural levels of language. The formal level consists of elements that can be discussed by multiple participants without requiring interpretation. The tool provides developers with information that clearly shows which modules need merging. The cultural level focuses on the remaining elements, those requiring explanations to become meaningful to other participants. For the developers this level involves understanding how the modules were changed, and how those changes interact with each other, and what the combined functionality needs to be. The merge facility only provides the formal level of language necessary for the work of merging.

This leads to two important observations. First, Robinson (1991) claims that when a system does not support both levels of language then it becomes unusable. In this example, it appears that the merge facility can work in some cases with just the formal level of language. However, when the work of merging contains elements of the cultural level the tool does not support the developers anymore. Second, it demonstrates the importance of organizational context. When multiple developers modify the same module, they change its behavior in different ways. During the merge these multiple behaviors need to be reconciled which requires knowing the reasons behind the changes and how the changes work with each other. This is organizational knowledge in the sense that it was established in the system requirements and design. It is the organization that brings the meaning to the changes made, by providing the context for them.

5.2. SUPPORTING DIFFERENT KINDS OF MECHANISMS

Previous research has shown that individuals help others coordinate with them by making their work activities' public (Heath and Luff, 1991). Researchers have drawn mixed conclusions about the benefits of using computers to make people's work visible to others (Bowers, 1994; Sommerville et al., 1993; Zuboff, 1991). However, these researchers have not made any distinctions between the kinds of coordination that take place at the individual level and those that take place among different groups. At Tool Corp. two levels of coordination stand out, the day-to-day development activities and the process of keeping the entire product together. The day-to-day work is the fitting, aligning and adjusting of work in hand that is necessary to be certain that software works together (Gasser, 1986). At this level the CM tool provides at least two coordination mechanisms, the time-line and work views, to help developers work together. At the same time Tool Corp. was also beginning to experience the issues associated with a higher level of abstraction, the process of keeping the entire product together.

At Tool Corp. both of these levels need to be managed to coordinate the development of their software. Strauss (1988) refers to this total coordination as the articulation process, that is:

> The overall process of putting *all* the work elements together *and* keeping them together represents a more inclusive set of actions than the acts of articulation work. (Strauss, 1988, p. 164, italics in original)

Tool Corp. was growing rapidly as a company and during my study the development group was reaching a point where the individuals in the group could not understand the entire product anymore. While the tool provided support for individuals working on the same part of the product it did not help developers work with others who worked on different subsystems. The architecture became a mechanism for coordinating work at a higher level although it did not provide much support. At Computer Corp. the organization had developed solutions for these higher level problems in the form of committees and organizational units. Despite the existence of these solutions enterprise-wide dependencies, those that spanned groups, took energy and time to resolve.

This study identifies the importance of the interplay between an organization and the technologies it uses. Computer Corp. like Tool Corp. used technology to support a part of the articulation process, the coordination among individuals on the same development team. However, as an organization Computer Corp. coordinated software development among groups that was something which the tool could not do. If researchers try to build systems to support the entire articulation process questions arise: what does the process encompass, can we abstract general features of the process from the organizational context, can a single technology support the process (probably not), and what kinds of representations can capture the essence of that process?*

5.3. Aligning mechanisms and work

There have been many discussions about the role of computer systems in structuring the work of individuals. One debate has focused on the Coordinator™ and the mechanisms it uses to try to enforce social action, through a series of commitments built into the system (Suchman, 1994; Winograd, 1994; CSCW, 1995). This study provides an opportunity to study how models embedded into the CM system interacted and competed with other models held by the developers and organizations of what software development practice should be. It also starts to identify some of the organizational and institutional sources of these models.

Developers at both Tool Corp. and Computer Corp. chose routinely between the multiple models of software development available to them as tools and policies. In a study of the International Monetary Fund (IMF) Harper and Sellen (1995) offer one way of thinking about these models, as organizational logics:

* At the level of enterprise-wide dependencies another question we might ask is what kinds of technology are we building, a kind of organization-ware that supports organization-wide coordination?

> The evidence we have provided turns on the claims that the activities we have described have a fundamental *organisational logic* to them. Professionals prefer paper documents for certain aspects of their work not because they are used to it, but because they afford certain advantages for the achievement of their practical ends. (Harper and Sellen, 1995, p. 128)

Harper and Sellen's definition of organizational logic focuses on the activity at hand. Taking this definition, as a way of thinking about the models of action which coordination mechanisms and other development practices are based on, provides a pluralistic way of analyzing how they manifest themselves in development work and their sources.

At Tool Corp. the developers create and assign themselves problems so they can get on with their work. Although the tool did not really support all developers having access to the privileges that allow them to generate problems, it was customized to support that. Gerson and Star (1986) observe that coordination mechanisms themselves require articulation work; for example, plans require adjusting to accommodate changing production schedules. In this case the coordination mechanisms were aligned with the organizational logic for this activity at Tool Corp. When the senior manager approved of the alignment, the work around became a standard, sanctioned, operating procedure, reinforcing both the organizational logic behind the decision as well as the alignment of the coordination mechanism.

Some logics of practice cannot be so easily reconciled, however. When the developers at Tool Corp. choose to edit a checked-in artifact, they choose a practice of saving time. It conflicts with a different logic, the one embedded into the tool that supports the idea that all shared modules should be stable and must not change. Both logics have their place in software development. Without a stable environment developers cannot ensure that their code works with all the other software around it, and it becomes much harder to guarantee that a working product can be created from the components. However, software product development often happens under pressure to get the software out the door, which means that saving time becomes important.

In a study of the use of a CASE tool, Orlikowski (1991) describes how the model of work built into the system reinforced the organizational status quo. Importantly, she observed that all the rules embedded into the tool were subject to interpretation by the people using it. At Tool Corp. the developers use their own logics of software development to interpret the role of the tool in their work. Tool Corp. also highlights the fact that organizational logics guide the usage of the tool long after system adoption.

At Computer Corp. some developers told me about the tension between avoiding merging and doing good software development. The tension rests between two logics, time versus quality. Developers do not want to merge because it often takes time away from getting on with other development activities. Like the developers at Tool Corp. they have deadlines to meet and systems to release. However, at the

same time, the developers at Computer Corp. want to produce quality software, and that takes time. The trade-off between time and quality is not new to these developers, but in this instance the tool appears to exacerbate the tension.

This case also reveals some of the outside influences that help software developers form their own models of how good software development proceeds. Pickering and King (1995) discuss the role of occupational communities in encouraging the growth of interorganizational computer-mediated communications. They describe occupational communities as:

> Occupational communities are groups of people engaged in work with such a high professional content that the norms of the profession transcend the norms of the organization that employ them. (Pickering and King, 1995, p. 128)

Currently, software engineers do not belong to a well-defined occupational community like those that Pickering and King describe; however, there is debate among software engineers about professionalization.* Clearly outside forces, like the Software Engineering Institute and ISO, have begun to shape software developers' views about what constitutes good software development.

6. Conclusions

In this paper I examined the role of CM systems in coordinating software development work. Instead of providing enhanced or increased communicative capabilities the tool supplies coordination mechanisms. In this study I have explored three aspects of those coordination mechanisms: representation issues, the need for different levels of mechanisms, and how those mechanisms support a model of software development.

This study offers two sets of conclusions, for the designers of CM systems, and for CSCW researchers. The design and development of most CM systems available today has been influenced by the difficulties of controlling the evolution of software. CM specialists have only recently realised that their systems have groupware-like features (Nix, 1994). The developers of future systems could learn from the existing empirical studies of all groupware technologies about the challenges of designing usable and useful systems.

However, this study offers a more substantive conclusion applicable to CM specialists and CSCW researchers. All three of the aspects highlighted the interplay between the organization and the tool that goes on continuously during the use of the tool. The challenge of representing complex merges is that of including organizational context inside the tool by making more information about the development process available to developers. The organization also coordinates software development efforts when the tool provides no support. Finally, the organization where the tool is deployed, and outside professional institutions, provide models about software development that developers must reconcile in their daily interactions

* See Boehm (1994) for a call to professionalize software engineering.

with the tool itself. This study suggests that organizational analysis (see Morgan, 1986) can be usefully applied to empirical studies of groupware technologies.

Understanding the interaction between the organization and the tool is important for designers building CM tools and other groupware systems, and CSCW researchers interesting in understanding the long-term use of collaborative technologies. To date most of the empirical studies have focused attention on interactions among the individuals using the tool in their work. Another stream of research that holds promise for understanding how groupware technologies support the work of organizations consists of understanding the organizations themselves and how they interact with groupware technologies.

Acknowledgments

I would like to thank the Engineering and Physical Sciences Research Council (UK) for financial support. Lisa Covi, Paul Dourish, Jonathan Grudin, Jeanne Pickering, Kjeld Schmidt, Carla Simone, Jim Whitehead and the reviewers have all provided helpful comments. Finally, I want to thank the people at Tool Corp. and Computer Corp. for their time and insights into the world of software product development.

References

Babich, W. A. (1986): *Software Configuration Management – Coordination for Team Productivity.* New York: Addison-Wesley.

Bendifallah, S. and W. Scacchi (1987): Understanding Software Maintenance Work. *IEEE Transactions on Software Engineering*, vol. 13, no. 3, pp. 311–323.

Boehm, B. (1994): Letter from the Executive Committee. *ACM Software Engineering Notes*, vol. 19, no. 4, pp. 1–2.

Bowers, J. (1994): The Work to Make a Network Work: Studying CSCW in Action. In *CSCW '94. Proceedings of Conference on Computer Supported Cooperative Work '94, Chapel Hill, North Carolina, October 22–26, 1994*, eds. R. Furuta and C. Neuwirth. New York: ACM Press, pp. 287–298.

Brooks, F. P. (1987): No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, vol. 20, no. 4, pp. 10–19.

Caballero, C. (1994): Life Cycle: Now the Focus in UNIX CM Market. *Application Development Trends*, August 1994, pp. 49–54, 64, 86.

CSCW (1995): Commentary on Suchman-Winograd Debate. *Computer Supported Cooperative Work (CSCW): An International Journal*, vol. 3, no. 1, pp. 29–95.

Dart, S. A. (1992): *The Past, Present, and Future of Configuration Management.* Technical Report CM U/SEI-92-TR-8. Software Engineering Institute, Carnegie-Mellon University.

Garlan, D. and D. Perry (1994): Software Architecture: Practice, Potential, and Pitfalls. In *Proceedings of 16th International Conference on Software Engineering, Sorrento, Italy, May 16–21, 1994*, ed. M. Kavanaugh. IEEE CS Press, pp. 363–364.

Gasser, L. (1986): The Integration of Computing and Routine Work. *ACM Transactions on Office Information Systems*, vol. 4, no. 3, pp. 205–225.

Gerson, E. M. and S. L. Star (1986): Analyzing Due Process in the Workplace. *ACM Transactions on Office Systems*, vol. 4, no. 3, pp. 257–270.

Grudin, J. (1989): Why groupware applications fail: Problems in design and evaluation. *Office: Technology and People*, vol. 4, no. 3, pp. 245–264.

Grudin, J. (1991): Interactive Systems: Bridging the Gaps Between Developers and Users. *IEEE Computer*, vol. 24, no. 4, pp. 59–69.

Grudin, J. and L. Palen (1995): Why Groupware Succeeds: Discretion or Mandate? In *ECSCW '93 Proceedings of Fourth European Conference on Computer-Supported Cooperative Work, Stockholm, Sweden, September 10–14*, eds. H. Marmolin, Y. Sundblad, K. Schmidt. Dordrecht: Kluwer Academic Publishers, pp. 263–278.

Harper, R. and A. Sellen (1995): Collaborative Tools and the Practicalities of Professional Work at the International Monetary Fund. In *Proceedings of CHI '95, Denver, Colorado, May 7–11, 1995.* New York: ACM Press, pp. 122–129.

Heath, C. and P. Luff (1991): Collaborative Activity and Technological Design: Task Coordination in London Underground Control Rooms. In *Proceedings of European Conference on Computer Supported Cooperative Work.* Amsterdam: Academic Press, pp. 65–80.

Hughes, J., V. King, T. Rodden, and H. Andersen (1994): Moving Out from the Control Room: Ethnography in Systems Design. In *CSCW '94. Proceedings of Conference on Computer Supported Cooperative Work '94, Chapel Hill, North Carolina, October 22–26, 1994*, eds. R. Furuta and C Neuwirth. New York: ACM Press, pp. 429–439.

Hutchins, E. (1990): The technology of team navigation. In *Intellectual Teamwork: Social Foundations of Cooperative Work*, eds. J. Galegher, R. E. Kraut, and C. Edigdo. Hillsdale, NJ: Lawrence Erlbaum Associates, pp. 191–220.

Jorgenson, D. L. (1989): *Participant Observation.* Newbury Park, CA: Sage Publications.

Lubkin, D. C. (1991): DSEE: A Software Configuration Management Tool. *The Hewlett-Packard Journal*, vol. 42, no. 3, pp. 77–83.

Morgan, G. (1986): *Images of Organization.* Newbury Park, CA: Sage Publications.

Nix, K. (1994): Using CM. *Software Development.* December, 1994, pp. 61–65.

Okamura, K., M. Fujimoto and W. Orlikowski (1994): Helping CSCW Applications Succeed: The Role of Mediators in the Context of Use. In *CSCW '94. Proceedings of Conference on Computer Supported Cooperative Work '94, Chapel Hill, North Carolina, October 22–26, 1994* eds. R. Furuta and C. Neuwirth. New York: ACM Press, pp. 55–66.

Orlikowski, W. (1991): Integrated Information Environment or Matrix of Control? The Contradictory Implications of Information Technology. *Accounting, Management and Information Technology*, vol. 1, no. 1, pp. 9–42.

Orlikowski, W. J. (1992): Learning from Notes: Organizational Issues in Groupware Implementation. In *CSCW '92. Proceedings of Conference on Computer-Supported Cooperative Work '92, Toronto, Canada, October 31–November 4, 1992*, eds. J. Turner and R. Kraut. New York: ACM Press, pp. 362–369.

Pickering, J. M. and R. E. Grinter (1995): Software Engineering and CSCW: A Common Research Ground. In *Software Engineering and Human-Computer Interaction: ICSE'94 Workshop on SE-HCI Joint Research Issues*, eds. R. N. Taylor and J. Coutaz. Lecture Notes in Computer Science Series 896. Heidelberg: Springer-Verlag, pp. 241–250.

Pickering, J. M. and J. L. King (1995): Hardwiring Weak Ties: Interorganizational Computer-mediated Communication, Occupational Communities, and Organizational Change. *Organization Science*, vol. 6, no. 4, pp. 479–486.

Robinson, M. (1991): Computer Supported Co-Operative Work: Cases and Concepts originally appeared in *Proceedings of Groupware '91* reprinted in *Readings in Groupware and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration*, ed. R. M. Baecker. San Mateo, CA: Morgan Kaufman, pp. 29–49.

Schmidt, K. and L. Bannon (1992): Taking CSCW Seriously: Supporting Articulation Work. *Computer Supported Cooperative Work: An International Journal*, vol. 1, no. 1-2, pp. 7–40.

Simone, C., M. Divitini, and K. Schmidt (1995): A notation for malleable and interoperable coordination mechanisms for CSCW systems. In *COOCS '95. Proceedings of ACM Conference on Organizational Computing Systems, Milpitas, California, August 13–16*, eds. N. Comstock and C. Ellis. New York: ACM Press, pp. 44–45.

Sommerville, I., T. Rodden, P. Sawyer, R. Bentley, and M. Twidale (1993): Integrating Ethnography into the Requirements Engineering Process. In *RE '93. Proceedings of Requirements Engineering 1993, San Diego, California, 4–6 January*, ed. L. O'Conner. pp. 165–173.

Strauss, A. (1985): Work and the Division of Labor. *The Sociological Quarterly*, vol. 26, no. 1, pp. 1–19.

Strauss, A. (1988): The Articulation of Project Work: An Organizational Process. *The Sociological Quarterly*, vol. 29, no. 2, pp. 163–178.

Suchman, L. (1994): Do Categories Have Politics? The Language/Action Perspective Reconsidered. *Computer Supported Cooperative Work (CSCW): An International Journal*, vol. 2, no. 3, pp. 177–190.

Tichy, W. (1985): RCS: A system for Version Control. *Software Practice and Experience*, vol. 15, no. 7, pp. 637–654.

Winograd, T. (1994): Categories, Disciplines, and Social Coordination. *Computer Supported Cooperative Work (CSCW): An International Journal*, vol. 2, no. 3, pp. 191–197.

Zuboff, S. (1988): *In The Age of The Smart Machine: The Future of Work and Power*. New York: Basic Books Inc.