Computer Supported Cooperative Work **9:** 189–214, 2000.

Workflow Systems: Occasions for Success and Failure

REBECCA E. GRINTER
Bell Labs, Lucent Technologies, 263 Shuman Blvd, 2F-309, Naperville, IL 60566, USA
(E-mail: beki@research.bell-labs.com)

(Received 12 December 1997)

Abstract. Workflow technologies have created considerable discussion within the computer supported cooperative work community. Although a number of theoretical and empirical warnings about the difficulties of workflow systems have appeared, the technologies continue to be built and sold. This paper examines the use of one workflow-like system and outlines three cases when the technology supported the work of its users. Comparing these successful occasions with some reports of difficulties, this paper draws conclusions about the circumstances that led to tool usage.

Key words: configuration management (CM), empirical studies, software development, workflow systems

1. Introduction

The development and use of workflow systems has generated much discussion within the computer supported cooperative work (CSCW) community (CSCW, 1995). Specifically, it is how workflow systems attempt to categorize, formalize and automate work that has raised questions for those interested in the fluid and often unpredictable nature of action (Suchman, 1994; Winograd, 1994; Suchman, 1995). These concerns have been echoed by empirical studies that describe how people found these systems difficult to use in their work. Despite the concerns workflow systems continue to be designed and bought by organizations.

In this paper I review these debates and describe occasions when one kind of specialized workflow tool – a configuration management system – was useful and usable *precisely* because it formalized and automated certain activities. I begin by introducing the questions that workflow systems raise for researchers interested in CSCW, and some of the directions taken to address these concerns. Then the domain of study, the system used and the methods used to collect the data are described. Next, I discuss three cases when the configuration management system supported development work: build management, creating awareness of others through shared feedback, and tracking problems. Finally, I offer some conclusions about the kinds of work that workflow systems may be good at sup-

porting by comparing these occasions with some less successful efforts reported elsewhere.

2. Workflow systems: Theoretical positions and practical reports

Workflow systems have emerged as one solution to the problem of coordinating events, artifacts, and people. The approach adopted by workflow systems attempts to reduce the complexity of coordination in three basic steps. First, the work to be done is reduced to a basic form through a process of categorization. The categorization breaks up work into elements such as activities, documents, and user roles. Once categorized, relations among the different components of work can be formally defined. For example, temporal sequences of events can be chained together, or artifact dependencies can be generated to ensure that entities such as documents may not continue to another state before they have been completed. These sequences or dependencies can be precisely described by the use of a formalism that defines the relations among specified components. The power behind this process of formalization is that the categorization can now be programmed into a computer. Finally, workflow systems may use the formalism to automate some aspects of the work entirely. It is these three steps that give technologies their workflow functionality.

However, the three steps of categorization, formalization, and automation also raised numerous questions for researchers concerned with understanding and improving the quality of work. Specifically, researchers asked questions about whom these systems serve, how adequately formalisms capture the work they attempt to model, and how the tools affect the environment they are used in (CSCW, 1995). Often these questions were grounded in theories about the practical accomplishment of work rather than in empirical observations.

One reason why early discussions about workflow systems may have been largely theoretical was the adoption of Speech-Act theory by the COORDINATOR™. Many of the initial commentaries about the system were critiques of Speech-Act theory, and as such were presented from a theoretical perspective. Although the arguments were sometimes generalized to all workflow systems, they stayed theoretical. Another reason why these debates may have remained theoretical was that workflow systems had their own adoption and use issues (Abbott and Sarin, 1994). Without a large number of systems in place, CSCW researchers interested in conducting empirical investigations had limited opportunities to study the challenges and uses of workflow technologies.

It is only comparatively recently that the evaluation of workflow technologies has taken an empirical turn. For example, Bowers et al.'s (1995) study of the use of a workflow system highlighted the challenges it faced when introduced into a print shop setting. Specifically, the print staff found themselves trying to fit the system's model of work into their own practices. This created a number of additional challenges for the staff and took them away from their work.

Despite theoretical and empirical "warnings" workflow systems continue to be developed for a number of reasons. First, they remain a seductive technology for many corporations. Many companies are continually looking for ways to reduce the costs and time associated with producing their goods and services. Workflow systems have been suggested as one way of managing these challenges. Further, some popular management methodologies advocate the use of information technology, including workflow systems, to support work processes. Second, some successes have been achieved with workflow systems (Agostini et al., 1994).

Moreover, workflow systems present an interesting research challenge: to find ways to support the work of individuals in a useful manner (Medina-Mora et al., 1992; Ellis and Wainer, 1994). One of the emergent trends in workflow research is an attempt to make the systems more flexible to accommodate the contingent aspects of work (Abbott and Sarin, 1994). One interpretation of the flexibility argument has moved some CSCW researchers away from rigid workflows to the idea of spaces where people can come together and develop their own ways of interacting with each other and artifacts (e.g., Fitzpatrick et al., 1995). Another interpretation has been to continue providing flows of work, but increase the amount of control that users have over them (e.g. Dourish et al., 1996; Bardram, 1997).

In addition to research focusing on building systems, other researchers have questioned the assumptions on which the theoretical criticisms of workflow technologies rely. In *Plans and Situated Actions*, Suchman (1987) argues that plans can not be a substitute for the situated activity in which work occurs. They are a resource, that helps individuals orient themselves to actions, but they can not predict or protect those engaged in the actual work from contingencies that arise. By leaving a place for plans as a resource in work, she leaves open the question of whether workflow systems could be used as a resource.

Schmidt (1997) takes this argument one step further, by identifying two different uses of plans. The notion of plan as a map guiding its user towards a potential outcome is similar to Suchman's idea of it as a resource for action. Schmidt also argues that there are plans that serve as scripts, guiding the actions of individuals much more directly. Using examples from software development – like bug logging procedures – he argues that some aspects of the coordination of development projects require much more standardization if they are to be accomplished. He argues that this is especially true when the group working on the activity increases in size.

Finally, Bowers (1992) offers a number of counter-arguments against the theoretical criticisms of formalisms. Two of the arguments that he puts forward about formalisms hold for workflow technologies. First, he argues that although some aspects of work may be complex and uncertain other parts could be routine and dull. Second, he observes that if there is such contingency in work then perhaps individuals can use the formalisms in contingent ways.

In this paper I argue that the question of whether workflow systems can be useful remains open because the answer is that they can be helpful and frustrating simultaneously. Furthermore, only empirical studies can start to identify

the occasions when workflow systems help their users and the limits of those opportunities. It is not my goal to say that workflow systems should always or never be used. Following Bowers and Schmidt it is time to reopen the question about the role of formalisms in supporting collaboration and discover whether there are occasions when workflow systems support individuals and how that is accomplished. These are questions that only systematic empirical studies can answer.

3. Software development and configuration management

Although workflow technologies have not been widely adopted yet, several more specialized technologies with similar properties have been in use for considerable time. One domain that has had some successes with systems that categorize, formalize, and routinize work is software development. For a number of reasons including the willingness and ability of the individuals to use technology – not to be confused with their impatience with systems that do not help them – and the domain of work, commercial software development provides some cases of use of workflow-like technologies.

In this section, I discuss some of the features of modern software development work. Then I describe one of the technologies designed to support the development efforts. I begin by describing the sites and methods used to gather and analyze data.

3.1. THE DOMAIN: THE DEVELOPMENT OF SOFTWARE PRODUCTS

In the autumn of 1967, the NATO Science Committee met to discuss the new field of computer science (Naur and Randall, 1969). In their discussions, they found themselves focusing on software engineering because it was emerging as a particularly difficult part of computer science. The Committee resolved to convene a special meeting to bring together people who conducted research in the area, and others involved in building systems.

That meeting, held in 1968, coined two terms, software engineering – the research discipline of understanding and improving development – and the software crisis. The software crisis was the phrase that participants used to sum up the enormous complexities of developing large systems. Thirty years later the term persists despite the vast improvements in our understanding of what it means to develop software.

Modern software product development is a complex activity for at least four reasons. First, most commercially available software systems are assembled from thousands of components including code, libraries, documents, and utilities. Along with the pieces in the product, modern software development environments contain other components for testing the software, supporting the programming activities, and managing the process. Software development efforts are collaborative endeavors involving many people because there are so many components. This

creates a large coordination overhead simply to manage the work itself (Brooks Jr., 1995). Further, as requirements evolve over time, the relationships among components are subject to change (Parnas and Clements, 1986). Finally, because the components must interact with each other to produce the desired functionality the individuals responsible for those related pieces of code must continually align their development efforts (Grinter, 1998).

Second, most commercial products must compete in a technologically heterogeneous marketplace. In other words, to secure market share software vendors must release versions of their product that run on different hardware platforms, and work with other kinds of software such as operating systems and databases. While these differences do not require that the entire product be rewritten for each platform and operating system, part of the development process involves making multiple versions of components that let the software interact with these technologies. So, if a product claims to work with three operating systems and three databases then there may be nine possible variants of the product. This makes product assembly and testing more complex, because now versions of the product need to be assembled for each combination of technologies the software will interact with.

Third, software is an unusual product to develop because it is very malleable. It is very easy to change software because anyone who has access rights can do so. However, even small changes can alter the functionally of the component, and any other related pieces. Moreover, one small change in a component that is central to the entire product – such as a piece of the editor of a word processor – can radically alter the behavior of many other parts of the software. Therefore, developers need to coordinate and control the changes made to the components to ensure that they do not affect others' work. Simultaneously, developers need to be protected against other people making changes that could affect their own work.

Finally, development times have dropped drastically in many sectors of the software industry. Many software companies are under pressure to develop new releases of their products much faster than they have been used too. This compounds the previous problems by requiring that they all get resolved much more quickly. Software companies have looked for ways of managing and organizing their development environments for all these reasons. One solution that has gained popularity is configuration management.

3.2. THE TOOL: CONFIGURATION MANAGEMENT SYSTEMS

Configuration Management (CM) has its background as an engineering discipline for organizing hardware projects. Initially CM procedures were developed to cope with the increasing complexity of managing military projects. As the technologies became increasingly larger and contained more components, engineers realized that they needed some way of tracking all the pieces of the products during devel-

opment. A legendary story that outlines the need for configuration management is,

> This deficiency became apparent in the race for a successful missile launch in the 1950s. With time being critical, the promulgation of changes was accelerated to resolve incompatibilities among elements supplied by many supporting contractors. When a successful flight was finally made and the buyer, in the euphoria of success, said: "Build me another one", industry found themselves in the following circumstances:
>
> 1. Their prototype was expended (launched into trajectory).
> 2. They did not have adequate records of part number identification, chronology of changes, nor change accomplishment. (Samaras and Czerwinski, 1971, p. 15)

It was only after launching the missile that the developers realized that they did not have enough information to build another one efficiently. Early CM efforts were paper-based systems that recorded: each component, the configuration of those parts into a system, and the changes made to each piece during development.

CM moved from hardware to software projects relatively easily. Several of the first large software development projects were done by military agencies already familiar with CM procedures. They literally transferred the ideas from one domain to another. CM transferred into the commercial software development domain as companies began to experience the same problems with their own large projects. They were also encouraged when some software systems to support these practices were built.

The first generation of CM systems for software development focused on controlling developers' abilities to make changes by using a library metaphor of "checking out" software to revise it, and "checking in" software to indicate that the changes were complete. Check-out and check-in prevented the problems associated with allowing unrestricted access to code because once a developer had a component checked out no-one else could make changes to it. Two early systems that feature this functionality are Revision Control System (RCS) and Source Code Control System (SCCS) (Rochkind, 1975; Tichy, 1985). Both available under the UNIX™ development environment they were widely used and helped to popularize the idea of CM systems.

However, these systems had two disadvantages for commercial software development. First, they only worked for code and simple document components. Modern CM systems support more types of components including libraries and test suites. Second, the check out state turned out to be very limiting because it restricted access to components too much. This slowed down developer's ability to get their work done because they had to wait to make their changes. Modern CM systems allow parallel development where two or more developers can check out the same piece of code make changes and then merge their versions back into a single integrated module.

Modern CM systems also manage the relationships among components and support the development of multiple product variants (Caballero, 1994). The system maintains information about which components make up a software release. Specifically it knows which version of each component goes into a certain release of the software system. It also maintains information about how those components relate to each other. This allows developers to find out exactly which components belong to a specific hardware configuration. Furthermore, it supports the automatic generation of products from the pieces.

The features I have described all support the identification and control of changes to components within the system. Modern CM systems have also broadened their scope to organize software development activities generally. CM tools accomplish this in two ways, problem and process management.

Problem management focuses on relating the development of solutions to specific problems that need fixing. Problems can be either difficulties with the current software or requests for new features to be added to the product. However, most CM systems provide ways of logging problems, creating links between problems and solutions, methods for presenting the problems in order of severity so that they can be assigned to developers to work on, and log the current state of the effort such as started, in progress, or completed.

Process management extends the notion of following problems through the system by providing the same functionality for all the other entities in the system such as components, documents, libraries, and test-suites. For example, a life cycle for a code component often consists of the following states: checked-out, checked-in, unit tested, system tested, and released. Further, process management adds to these life cycles by providing a set of user roles that define what an individual may do to an artifact in a certain state. For example, only individuals with the developer role may be allowed to modify checked-out components, and only those with the tester role may test a checked-in component.

These additional features provided by CM systems provide workflow management for software development activities. CM systems rely on categorizations of the artifacts and activities that comprise development. For example, they classify entities like developer, component, library, and build manager. This classification begins to identify and separate distinct parts of the overall work of software development.

The problem and process management parts of the CM tool formalize software development work by restricting access to artifacts based on the user role of the individual. This formalization is extended through the use of the life cycles which structure the work of software development by prescribing a sequence of steps through which different artifacts must go. Using these features, CM tools allow the construction of a sequence of workflows, where users in one role may perform a specific task, and then transfer their results to the next person with a different responsibility.

CM systems are a domain-specific workflow system for supporting software development. They categorize and formalize the process of software development by specifying entities and relationships among them. Finally, CM systems automate some parts of software development work entirely such as build management.

3.3. SITE AND STUDY METHODS

The data reported in this study were gathered in 1994–1995 as part of a series of studies about configuration management practices and systems. The data are drawn from two sites.

Tool Corp. is the vendor of a CM system, which they use in house to manage the development of the next release. The tool itself contains about 1 million lines of source code. During my time at the company the development group was experiencing rapid growth, increasing from 14–18 people in less than four months. The developers were all co-located on one floor of a single building.

Computer Corp. is the vendor of an operating system. The corporation employed around 700 development staff at the time of this study and their product contained approximately 10 million lines of code. Many of the software developers work at the company's headquarters in Silicon Valley, but they have developers located in other states and countries. Computer Corp. had just started using Tool Corp.'s product in their software development work.

The development efforts at both sites were sub-divided around functional distinctions within the product. Teams were assigned to various sub-systems within the software by using the overall architecture of the product. Individuals had specific areas of expertise within those teams. At Computer Corp. there were many teams organized around the different operating system functions. At Tool Corp. the software production effort was much smaller so the developers worked in two teams again related to a functional distinction between two parts of the product.

The data were gathered using a combination of interviewing and observation techniques. At Tool Corp. I had full access to the corporation and was given a cubicle among the developers. I also had access to their entire software development environment. I visited the headquarters of Computer Corp. and was taken to various parts of the corporation to meet different development groups. I also sat in on a class where developers were learning how to use the CM tool built by Tool Corp.

I was able to conduct interviews at both sites. At Tool Corp. I used unstructured and semi-structured interviewing techniques to gather information about the CM challenges that the developers faced (Bernard, 1988). I collected approximately 100 interviews of which 20 were taped and transcribed. At Computer Corp. 13 semi-structured interviews were conducted, and were taped and transcribed.

I used observational data gathering techniques at the beginning of this study. My primary focus was on how configuration management systems supported the

coordination work involved in developing software. As I began to see reoccurring patterns of tool usage – both positive and negative – I started to develop different categories of software development coordination work using grounded theory techniques (Glaser and Strauss, 1967; Strauss and Corbin, 1990). Over time, the data analysis focused more exclusively on interviewing techniques where I was concerned with expanding, verifying, and correcting the information and explanations I had previously generated.

4. Case 1: Automating the build

The components that belong in each product variant are assembled many times before final release. The purpose of routinely assembling the system from the parts is to test that all the components work together. A second stage of testing involves ensuring that the working product provides the desired functionality. This process of putting the system together from the components is known as "the build." As the product gets closer to its release date the time between successive builds decreases rapidly – it may drop from a week to a matter of hours – because more people are trying to verify that their components work as expected.

Building the system involves finding the latest versions of all the components that belong in the product variant being built. At both sites, the development teams assigned one developer – known as the build manager – to take responsibility for this job. Although build managers were assigned the same responsibility, there was a stark difference in the work that those with CM tools did in comparison with individuals still using manual procedures. I will describe these differences – and the benefits of using the system – from the perspectives of the build managers and the developers.

4.1. THE BUILD MANAGERS

As Computer Corp. was still in the process of migrating all the teams to the CM tool, some groups still used manual build procedures. In these teams the build manager would visit every developer in the team, get their changes, and compile them. Among these build managers there was a desire to have some kind of system organize this work for them. As one build manager put it,

> It doesn't really track "am I getting the right version of this thing" . . . and unless you have, um, a system for doing that, which people have done in a manual way like writing down on bits of paper, talking to 18 different developers . . . that are producers of their dependencies, there is no way.

another build manager described a similar manual procedure,

> I make them tell me stuff like what [files] to grab and what's the version number of them, what bug are you fixing and how am I supposed to know once I install it if that bug got fixed or not. What's the behavior I'm supposed to see and

> then, when it's necessary, I do the [recompile] and install and send out mail. . . . I keep a track of it, and how I track is that every time I do an install I send everybody e-mail saying this is what I installed, this what is was supposed to fix and it's ready for you to use now or whatever . . . [1]

Although build management is time consuming work, the build managers recognized that there was some benefit to visiting everyone in their team routinely. These build managers talked of knowing a lot more about the overall state of the system than their colleagues. The continuous interaction with other team members in the process of gathering the latest changes meant that the build managers had detailed information about the overall progress of the development effort. One build manager who was relatively new to the company also described it as a way to learn about exactly what the product did.

Despite the advantage of knowing the current state of the system being developed, most build managers found the manual procedures exceptionally time consuming. This becomes especially true when the system enters the final weeks of development and builds may take place two or three times a day instead of once a week. Furthermore, as the job of build manager is often an additional responsibility it takes them away from the software development work that they want to do.

Other teams at Computer Corp., and the teams at Tool Corp. used the CM tool to do the build. On command, the system gathered the latest code changes from everyone in the team and compiled them. If the compilation was successful, the tool would produce a new version of the product for testing. Otherwise, the system would notify the build manager by providing information about where the problem occurred. The build manager then investigated the problem until they found its source. Finally the build manager would inform the developer responsible for breaking the compilation and ask them to repair their component.

The build managers who had worked in a manual mode and then switched to the CM tool all preferred the automated process. As one ex-build manager said,

> [The tool] makes it easy for the person in charge of the product to build the latest tested version of the product. It removes the manual process of the developer saying I've finished with this, you can use this now. That's a pretty big advantage, I was a build manager for a part of it [team name] for a couple of months.

Specifically the tool had three advantages over the manual build procedure. First, the tool reduced the amount of time it took to gather all the components for the build. Second, it was much better at finding the newest versions of the components made by all the developers. Finally, it also provided information about problems that broke the build, which often helped in locating the source of the error quickly. These responsibilities are part of the work of build management, but when automated make the job easier for those doing it.

4.2. THE DEVELOPERS

The developers also liked automated build management because of the guarantees it provided them about the product. As one developer said,

> I come in the morning and I [get a system update] I get the latest of everything and I generally don't even have to worry about it. I just know it's going to be there and it's going to work fine. Then I can just go about my business, having gotten everyone else's changes automatically.

The developers saw at least two benefits of using the tool. First, when developers check-in their components they signal that their work is ready to be incorporated into the next build. The tool then automatically collects those latest versions of their components – while ignoring anything they have checked-out to work on – and does the build. Second, when the build is complete the tool has created a version of the product that contains everyone's latest changes, meaning that it is the most up-to-date version of the system. Developers can then use this latest system to test the changes that they are working on currently.

These benefits were contrasted with the problems that used to occur with manual build procedures. In large groups, it took a long time to gather everyone's changes manually. Sometimes, by the time the last developer had provided their changes the first developers had already revised other components. This led to time delays between completing work and seeing the results of that in the build. This would become especially problematic during the final stages of product development when everyone wanted feedback very quickly.

Furthermore, often one change to the software's behavior results in revisions to a number of components. When some of the revised components get into a build, and others do not, the build often fails. The CM tool had a mechanism for ensuring that all the related changes got into the build or stayed out of it, which was the responsibility of the build manager in the manual groups. It was hard for the build managers in the manual groups to track the dependencies between changes, but failure to do so usually resulted in a broken build that slowed down the testing process.

The build provides a case when automation appeared to benefit both build managers and developers. Specifically the build managers found that it reduced the complexity of their responsibility as well as providing useful features to help them in their work. The developers enjoyed the guarantees that the tool provided them, and found it relatively easy to follow the scheme of checking-in work to signal to the tool that they were finished with a component.

5. Case 2: Awareness created by the formalism

Studies of workplaces show that awareness is critical to facilitating collaboration among individuals. For example, studies have shown that people use spatial cues, watch their colleagues working, leave cues for others – such as speaking louder –

to help them orient their work towards others (Heath and Luff, 1991; Anderson and Sharrock, 1993). A challenge for CSCW systems is to find ways of representing this kind of information to others.

Dourish and Bellotti (1992) describe three mechanisms that systems can provide for generating awareness of others. Systems that support informational awareness mechanisms explicitly ask for details of what their users are doing by using artifacts such as on-line activity logs. Role-restrictive awareness comes from mechanisms that assign individuals specific roles within the system. Others can use information about the role to determine what an individual may be doing. Finally, they propose a third approach – shared feedback – where awareness is generated by providing information about what others are doing in a shared workspace.

The CM tool studied provided all three kinds of mechanisms. It used informational awareness mechanisms in the form of logging information about the changes that developers made to various components during development. The tool also had the notion of roles, such as build manager, developer and tester, that let others see what activities individuals were engaged in. I have written about the benefits and limits of these two approaches elsewhere (Grinter, 1995, 1996) and in this section I want to focus on the shared feedback mechanism for providing awareness information. The CM tool provided this kind of awareness to the developers through the formalisms that the tool used to categorize the work of software development.

When the developers launch the tool, they initially see a view of the components under development. This view follows the desktop interface metaphor. The components are arranged by name, and each one shows its latest version number, the state of that component, and current owner of the file (their E-mail handle). Each team shares a collection of views that together comprise the sub-system that they are responsible for developing.

The main view that the developer sees when she launches the tool is the one corresponding to a directory of components she is currently working on, but it is very unlikely that she will be working there alone. Other developers will be changing other files – or possibly the same file – in that directory, and that information is available to her through the shared feedback mechanisms within the tool.

The formalism embedded in the CM tool contains life cycles that describe the states that artifacts can be in, and what it means to be in a certain state. This, in combination with the other information provided for each component allowed developers to make inferences about what their colleagues were working on. This proved especially useful when developers wanted to know whether other people were changing parts of the system related to their own components.

Furthermore, this view was not static. When developers changed components the system updated the version number and life cycle information. The CM tool allowed developers to update their main view any time they wanted to. This process was known as *reconfiguring* the view. I was alerted to the awareness created by the system when the developers talked about “seeing” things inside the system. For

example, two developers described their use of the shared feedback the system provided quite explicitly during interviews,

> In your own personal [sub-systems] you can see what the state of the parts of the project you are working on are because you get everyone's latest versions that others have checked in. When you reconfigure your [view of the sub-system] you see what versions you get, the dates on them, who owned them, who [changed] them, what changes they include.
>
> Sometimes I can tell from just reconfiguring my stuff and I can look and see what, who owns all the versions that I just got in. I can see that certain things have been changing.

The main view helped developers gauge the state of the project and made them peripherally aware of what their colleagues were working on. However, the information also allowed them to reorganize their work as the components in their view changed. For example, one thing that the developers did not enjoy doing was working on the same piece of code at the same time. Merging the two different versions back together can be difficult. However, the main view easily let developers see whether anyone else was working on the component that they wanted to change. As one developer put it,

> I'll look and see and if someone has it checked out, the module I want to modify and mine's not too difficult. I did this last night, I sent them mail and asked can you do this for me in your version ...

The shared feedback provided by the system came from the formalism embedded in the tool. It would be nearly impossible to display all the details of others' work in a view like that, and probably undesirable or useless to the developers using the system. The formalism gave developers shared feedback about current development state by abstracting the details away from the work. When the awareness information was not sufficient to resolve problems, the system often provided pointers to the individual to ask, by showing who was working on what components.

6. Case 3: Tracking problems

The final case where the workflow-like qualities of the CM tool supported the developers' work concerns the integrated problem reporting facility. Like other components in the system, problems also had a life cycle starting from when they were entered to the system and ending when they were resolved. After the initial entry process, a group of people would meet and prioritize the problems in the system and assign them to developers. This assignment would be logged within the system, and then the tool would notify the developer who was responsible for resolving the problem.

The developers used the notification procedure provided by the tool to tell them what outstanding work they had. Each problem assignment also carried a priority

letting the developer know how serious it was, and giving them a sense of the urgency of completing the problem. Although the system told the developers what outstanding problems they had and their significance, it still let them choose specifically which ones to do. In other words, it did not force them to work on urgent problems. This latitude in prioritizing work – that allowed them to take advantage of the peripheral awareness mechanisms to see whether it made sense to start on a change or leave it until someone else had finished something else – created a culture where the problem reporting facility was used as a scheduling device. As one developer put it,

> It's nice, you come in the morning and get a mail message, these are all the problems assigned to you, just look at all of them. No-one actually has to come to my office and say this is a bug, it has to be worked on, I just know because it's automatically generated and sent to me. So I look at that to figure out all the things I have to do.

At Tool Corp. the problem reporting facility was not free of some of the challenges of making workflow technologies work. Specifically the tool requires that all changes to the code must be associated – by hypertext links – to the problems. Consequently, if the developers finished their assignments they could not make any more changes to the components, until they received new problems to work on.

This led to the system becoming unworkable because the people who met to determine the assignment of problems could not meet frequently enough to prevent the developers from running out and consequently being unable to work. The difficulty was solved when the managers decided to let the developers have the ability to create and assign themselves problems. In terms of the system, this meant giving developers the role of supervisor in the problem reporting facility.

Giving developers these system privileges led to increased use of the problem reporting facility by a number of individuals. Having the ability to change and add information, developers often broke initial problem assignments into smaller pieces, and created more detailed problems to work on. This was all done in the context of scheduling and managing their own work. In other words, by being given the opportunity to specify and organize their own work on-line, a number of developers refined the problem reporting system to meet their needs.

> I like to use the tool to organize my work. I use the [problem reporting] facility. I create tasks for just about everything I do.

Furthermore, with the ability to assign themselves problems, the developers could make notes to themselves about problems to be fixed in the future while working on other – possibly related – problems. In addition, the supervisor role now let the developers assign their own priorities to problems and make their own estimates about how long problems would take to complete.

> So it keeps track of all the problems which I have assigned to me and I can put priorities on them, so I know which ones I'm going to do first, also it has a

field for estimated duration, so I can get an idea how long it will take me to do everything, and I can budget my time.

Many developers used the problem reporting facility routinely in their work. Its primary purpose was to give developers up-to-date information about the work assignments. This information was provided daily to them in the form of an e-mail message generated by the tool. However, at Tool Corp. the revised role assignment that developers took encouraged them into using the problem reporting facility as a comprehensive scheduling system for organizing and managing their own work.

7. Discussion

The previous three sections described occasions when the workflow system – the CM tool – supported the work of those who used it. In this section, I want to revisit these cases and examine why they were successful. Specifically, this section offers four reasons why the tool worked on those occasions: (1) the developers understood and accepted the model of work, (2) it provided understandable and useful representations, (3) the "right" work was automated and (4) the corporation was supportive.

However, this is not a recommendation that workflow technologies are universally successful in supporting work. I have previously described some of the difficulties in using CM tools in development (Grinter, 1995, 1996). Furthermore, other studies of computer systems for supporting development work have also highlighted a number of occasions when the technologies created problems for those using them (Bowers et al., 1995). In this section, I use these studies to better understand the reasons why the system worked well in the three cases described.

7.1. UNDERSTANDING AND ACCEPTING THE MODEL OF WORK

Like any workflow technology, the CM tool had a model of work embedded in the system. The developers had to understand and accept that model to use the tool on these occasions. This was very explicit at Computer Corp. where developers were switching from one comprehensive CM system to this new one, and going through some training to make this change.

Although the developers at Computer Corp. were experienced users of configuration management systems, they were encouraged to take part in a training class to explain how the new tool differed from old one. The new CM system differed from the old tool in how it structured software development work and Computer Corp. designed the class to help ease the adoption process by explaining the differences and similarities between the new tool and old ways of work. It was an attempt to get the developers to understand and accept the model of work.

I attended one of these classes, which proved to be an occasion to explicitly discuss how the new system modeled software development activities in comparison

with the old tool they understood. Comparisons were drawn between the two systems continuously during the class, even when they were not included as part of the schedule. Moreover, the instructors found themselves explaining the assumptions behind the new tool's approach to software development to the class attendees.

The class was intended to introduce the developers to the new tool, which they had often started using prior to the class. In practice, it was that, but also an occasion for developers to come to understand the model of work embedded in the tool, and decide whether that model was acceptable to them. The classes itself were only a part of a much longer process of learning the CM system in depth by using it in their development work.

In the three cases described the developers who used the tool routinely both understood and accepted the model of their work. They understood the model well enough to know how the tool functioned. This may sound trivial, but in many cases models are hidden behind abstractions that present limited information about what the system is doing (Button and Dourish, 1996). Furthermore, the developers were able to make the tool's model fit with their software development practices. For a workflow system to work, for any groupware system to work, both stages must occur.

However, it was not always easy to accept the model of work presented. In the case of the bug tracking system, the developers had to find a way around the constraints of the model that did not permit them to work on code without having a problem to assign their changes too. Occasionally during testing activities a few developers would violate the checked-in state rule, by changing code that was marked as being completed (Grinter, 1996). This led to problems for others who followed the model's assumption that checked-in code was finished.

Despite these difficulties, the developers continued to use the CM tool with relative ease. This was not true for the users of the workflow system that was installed at Establishment Printers (EP) (Bowers et al., 1995). The workflow system introduced to the shop floor did not match the prints staff's work. Instead, the system distanced them from their methods of managing the complexities of printing jobs and created additional overhead in the form of problems that needed solving if the work was going to be completed. As the authors put it,

> Rather they [the methods embedded in the formal model of print shop work supplied by the workflow system] offer *another way of organising printwork*, one which is encountered by the workers at EP's sites as alien to *their* methods of organising printwork. (Bowers et al., 1995, pp. 63–64) Italics in original

Perhaps one reason why the developers at Tool Corp. and Computer Corp. found the model acceptable enough to use the system was because the individuals who developed the configuration management system understood development work. The developers at Tool Corp. not only used their tool, they built the system, and so were both users and designers, perhaps the ultimate participatory design experience. The developers at Tool Corp. could use their own experiences of development

to build a system that worked well enough for them and their counterparts at Computer Corp.[2]

The development of understandable and acceptable models of work are one aspect of making workflow systems work. These models are not "good" or "bad" in an abstract sense, but good enough for someone to be able to understand and make it fit into their work practices, or bad enough that the tool causes serious delays and confusion. A workflow system constructs its model of work from the categorizations and formalisms provided. In the case of the CM tool, the categories captured the pieces of the software development work well enough, and further, the formalism built relationships among those elements that were generally not unreasonable. Together, these aspects of the CM tool were a good enough fit to the work being done that it could support the development effort.

7.2. UNDERSTANDABLE AND USEFUL REPRESENTATIONS

The developers used the main view and problem reporting facility in part because the tool provided an understandable and useful representation of information. By representation, I mean: the interface, the presentation of the information inside the windows, and what content the system provides. The developers found that the representations in both the main view and the problem reporting facility contained information about the current state of work that they found useful and usable.

In the main view this happened because everyone shared an understanding of what information the system was representing. As one developer put it,

> In [the tool] the system sets up everything in a standard way. It's easy to find out what is going on. There's rhyme and reason to it all.

Furthermore, the system remained constant in its representation of the information. When developers switched among views, they saw identical types of information about the components in the view and it still meant the same thing. This differs from a development effort where each developer maintains their own directories of files, each potentially using a different structure for organizing their work according to their preferences. By letting the system organize and present the information, no matter what part of the system the developers were looking at, or who owned that piece, the information about what was being worked on was presented in the same way.

As a result of this consistency, the developers could trust what the system was telling them about the current development state. This held true no matter what part of the system was looking at, and irrespective of whether the individual owned any, some, or all of the components in the view. This is not an argument for making all interfaces consistent and arguments suggest that it can be problematic (Grudin, 1989). Instead, this is similar to Sommerville et al. (1993) observation about how air traffic controllers look at the screens of their colleagues to understand what is about to happen in their own domain. In this case, the screens are all shared inside

the development environment. The advantage of this consistency was the ability to be able judge what the state of the development effort was simply by looking at the view itself, rather than having to consult with others.

The main view was also useful because of its ability to provide peripheral awareness to the developers about others' actions. The formalization provided that information readily because it reduced the details of others' work to a brief, consistent, and useful form. Specifically, it allowed any developer to see who was working on artifacts that might share a dependency with their own work. With this information, developers could decide whether to begin making changes that might impact other individuals that appeared to be working on related items. If they were not sure, they could use the information to contact the other developer and find out more details.

It was not only the information that was valuable, but the fact that developers could easily get updates by reconfiguring their view any time they liked. The reconfigure command was used as a way of aligning work with that of others. For example, developers would use the reconfigure command before starting a change to see whether anyone else was working on that component, or before checking in a change to ensure that they did not need to do a merge. The information and the ability to update that information instantly would have been very difficult and time consuming to provide manually, because it requires knowing everyone's exact state of work at the same time. The categorization and formalization of software development work into these abstractions provided just enough details to support this peripheral awareness.

The use of the problem reporting facility also relied on it presenting information in an understandable and useful way. Initially they used the facility as a checklist of their outstanding work. When developers were given the opportunity to use more parts of the facility by taking the supervisor role they did. As supervisors the developers increased their use of the scheduling functions: organizing their work in more detail by breaking down problems, assigning estimates of how long changes would take, and logging work to be done in the future. In other words the developers appropriated and customized the representations and made them even more useful.

However, the tool's representation mechanisms were not always so useful or usable. One case when the representation sometimes failed was in the case of merging code. When two developers made changes to the same component simultaneously, they needed to combine their work to produce the final version. Quite often, this could be done using the tool's mechanism for merging which displayed the different modules side by side and highlighted the places where they varied. The developer responsible for the merge selected what would go into the final integrated version.

However, this representation often failed when the developers changed the same lines of code or the algorithm (Grinter, 1996). When the algorithm or same lines of code have been changed often more information about what has changed and why

needs to be provided to the developer responsible for merging the versions. This leads the developer to need information including an understanding of what problems were being solved, why the design solution was chosen and how the solution was implemented. In other words, merging changes from being the synthesis of two versions to being a new design project.

It is challenging for any tool to present comprehensive design information. In addition to providing detailed and current information about the design requirements, a tool would probably need to show control and data flow, and other representations of the software solution space. Configuration management tools, including this one, could not provide the developer with enough information to help them with their new design task. Instead, the tool provides one critical piece of information, the name of the person who made the other changes. The developers used this information to contact others and begin the process of designing a component that combined all the functionality represented in the different versions.

At Establishment Printers, the workflow system presented the shop floor workers with a number of representation problems. The system made entering information about activities a time-consuming activity that took the print staff away from their printing work. At some of EP's sites, the representations of work were maintained manually as well as on-line. When a print job was being done manual records that accurately reflected the work were maintained. Once the job was completed, one administrator entered the information into the workflow system. In other words, the system was used to reconstruct the work rather than reflect work-in-progress.

Creating understandable and useful representations has long been the central concern of the human-computer interaction community. Groupware systems face the additional challenge of representing other peoples' actions as well as displaying information contained within the system (Grudin, 1990; Bowers and Rodden, 1993). The CM system managed to express some of the work of others in its representations. In some cases, such as the problem reporting facility and the views, the system provide useful and usable representations.

At Establishment Printers, the representations of work, based on the models of how printing should be done, were incongruent with the print staffs' practices. Instead of finding ways to use information given to them by the system, they continued to use their manual methods for expressing the current state of work. The representations ended up serving as reconstructed records of how the jobs were done that only served auditing purposes.

Problems merging code using the CM tool highlight a particularly challenging aspect of creating representations. Sometimes the representation worked, and other times it failed, as merging increased in complexity. The merge representation did not accurately capture that shift in complexity from the simplest cases to the hardest ones.

Merging modules is a convenient handle for describing the work of reconciling multiple changes, but does not convey the heterogeneity of the potential merge scenarios. This is a compelling argument for detailed studies of work that tease apart the subtle complexities hidden by homogeneous sounding activities.

7.3. AUTOMATING THE "RIGHT" WORK

The tool automated some aspects of the software development work almost completely. The build process was an example of this. It was also an example of picking the "right" work to automate because everyone supported the automation and some people benefited from it.

At both sites, the build process was of concern to four distinct groups: developers, build managers, testers, and managers. The build managers supported the automation effort as a way of reducing the amount of time they spent doing that work. The testers and developers benefited from the automated build by being relieved of their part in the process. Moreover, they enjoyed receiving up-to-date code from which to begin their own testing and development efforts with minimal effort on their part. Finally, the managers at Tool Corp. and Computer Corp. were invested in getting new versions of the software released to the market as quickly as possible. They saw the automation of the build process as part of streamlining the development cycle and supported it.

Build management work is also an interesting candidate for automation because there is historical precedent for it. A UNIX™ utility called "make" provides some degree of automated support for building products out of components. Make does not provide all the features that modern configuration management systems do, but it sets the precedent for thinking about automating build management work.

It is less clear whether there was a precedent for automating the print work. Clearly though the workflow system attempted to structure and routinize parts of shop floor work that were anything but routine in practice. Print work involves juggling jobs to optimize the use of machines and people and ensure that everything got done. The workflow system prevented the workers from making these run-time adjustments to their printing schedules.

Build management was a good candidate for automation because everyone involved in that work benefited. Furthermore, it was not difficult to convince people that it could be automated, as technologies like make, have existed for some time to support the process. Finally, previous technologies like make may have influenced build management practices in ways that aligned them with this new automation effort. The print staff were less fortunate, if there were aspects of their work that could have been supported by technologies in ways that freed them to concentrate on other activities, it seems clear that the workflow system did not find them.

The problem reporting system also provided some occasions for automating software development work. In this case, the system gathered all the latest changes

entered – which could be entered by developers or customer support – and collated them for the people involved in problem assignment. Once assigned, the system automatically notified developers of these new problem assignments.

However, in the problem reporting system there was a distinct balance between what was automated and what was not. For example, while the system collated the problems, it did not automatically assign them to individuals. Instead the problem assignment group, or individual developers in the case of the problems they generated for themselves, made judgements about who to give the work too. In addition to looking at expertise in parts of the system, the group would make decisions based on work outstanding and skills at certain kinds of problem solving work.

Another place where the problem reporting system could have enforced some structure, but did not, was in the prioritization of problems to work on. The system could easily enforce a set of conditions where developers had to work on specific problems before starting others. This did not happen; instead, the developers chose what they would work on. This allowed the developers to select based on their expertise at assessing what length of time was required for the work, and find an occasion when their work did not adversely impact anyone else, which were often judgement calls they made in practice.

In both of these cases the problem reporting facility managed a delicate balance between what was automated, and what was better left to individual decision making. In fact, the system was moving between treating problem reporting as scripted action for gathering them together, and as a map for supporting individuals in their work (Schmidt, 1997). It is also reminiscent of Freeflow's approach of allowing the users of that system to make more decisions about how best to structure some of the work, although it does not provide as much direct interaction with the tool.

Finding the right kind of work to automate consists of a number of elements. This study suggests that these elements include finding work to automate that people will benefit from, and when a precedent exists that supports this conclusion. Further, the case of the problem reporting system suggests that some balance can be struck, in this case using the tool to gather and distribute information, but letting individuals make their own decisions based on that knowledge.

7.4. A SUPPORTIVE COMPANY

Many of the previous studies of groupware adoption and use have highlighted the changes, both anticipated and unanticipated on how individuals' accomplish their work (Orlikowski, 1992; Grudin, 1994; Ciborra, 1996). As these studies have pointed out, how individuals respond to these changes is a matter of adjusting processes and adapting technologies in ways that still leave them able to get the work at hand done. However, this adaptation and adjustment is only made possible when it is permissible by the company that the individuals work for.

In this study, the managers at Tool Corp. and Computer Corp. viewed developers as professional staff. This was visible in a number of ways. Managers encouraged developers to attend talks on software development topics, to read the literature, and supported local chapters of professional societies. The developers also enjoyed autonomy in structuring their workday and deciding how to resolve problems. Most significantly, the managers took the developers' opinions.

This kind of culture made it relatively easy for the developers at Tool Corp. to discuss their difficulties with the problem reporting facility and get permission from their management to change from developer to supervisor role. That change was simple to implement technologically, but in environments where users do not have that kind of control or ability to change their circumstances, the modification is impossible to make.

A professional culture could not reconcile the workflow system to the print work at EP because the management was not free to adapt the system. EP adopted the workflow system as part of fulfilling a contractual obligation to a client. They had to continue using the system as is against all of the difficulties that they faced or lose the work. As Bowers et al. (1995) point out contractual obligations and specifically organizational accountability – the system was the way that EP accounted for the work that they did, and how the client audited EP – raise new issues for CSCW.

Grudin and Palen (1995) found in their studies of the use of group calendaring systems that mandating usage can lead to adoption. This was true at EP, but the ways in which that adoption was accomplished, and the costs to those who had to use the system were high. The users at Tool Corp. and Computer Corp. had some discretion how they would use the system, and exercised this right to change difficult things as best they could. One of the differences between the two workflow systems may not have been the flexibility of the technology itself, but the ability of those using the systems to negotiate how they would use them.

8. Conclusions

Workflow technologies have generated an important series of discussions within the CSCW community. While these debates are grounded in theories of work, there have been less empirical studies of workflow systems in use. Further, the studies that exist often point to the difficulties of using these systems. This paper has reported on three occasions when the work of software development was supported by a workflow system.

In this paper, I have outlined four reasons why this system was used. They are: (1) understanding and accepting a model of work, (2) providing understandable and useful representations, (3) automating the "right" work, and (4) having a supportive company. These reasons highlight the *context-dependent* aspects of the use of workflow technologies in particular settings.

Furthermore, this study suggests that questions about whether workflow systems fail or succeed can be hard to answer. Defining success and failure in commercial contexts is harder than it sounds as Blythin et al. (1997) observe. Workflow systems are certainly as complex as other groupware systems and this study shows that in practice they may accomplish success and failure simultaneously. One developer can be learning about the current state of development and reorganizing her work, while another can not figure out exactly what the outcome of a merged module will be. These mixed reactions arise because systems such as CM tools are environments that attempt to support a cross-section of development activities and the individuals doing them simultaneously.

Workflow systems, like other complex systems, can fail and succeed simultaneously. This study provides a basis for grounding some of these discussions about workflow technologies by comparing some of the difficulties with some occasions when the tool was used. Further, the study suggests some opportunities to implement workflow ideas that could lead to a positive outcome for those who use the system.

Acknowledgments

I would like to thank the numerous developers at Tool Corp. and Computer Corp. for patiently explaining their work practices to me. The Engineering and Science Research Council made these field studies possible by supporting me financially during this research project. I would like to thank all the reviewers for their comments, which strengthened this paper considerably, and especially the person who gave me inspiration for attempting the analysis.

Notes

1. Notes in square brackets represent references to artifacts and processes that might identify the company, so I have replaced them with more generic terms to maintain confidentiality.
2. Although the developers at Tool Corp. contributed their own ideas about how development happens to the design of the system, other requirements still shape the development effort. For example, the problem reporting facility contained a model of problem assignment that seems more appropriate for hierarchically organized software development. In my time at Tool Corp. I observed potential customers being attracted to this hierarchical model.

References

Abbott, K. and S. Sarin (1994): Experiences with Workflow Management: Issues for the Next Generation. In R. Furuta and C. Neuwirth (eds.): *Proceedings of ACM Conference on Computer Supported Cooperative Work CSCW '94, Chapel Hill, NC. October 22–26, 1994*. New York, NY: ACM Press, pp. 113–120.

Agostini, A., G. De Michelis, M.A. Grasso and S. Patriarca (1994): Re-engineering a Business Process with an Innovative Workflow Management System: A Case Study. *Collaborative Computing*, vol. 1, no. 3, pp. 163–190.

Anderson, R. and W. Sharrock (1993): Can Organisations Afford Knowledge? *Computer Supported Cooperative Work (CSCW): An International Journal*, vol. 1, no. 3, pp. 143–161.

Bardram, J.E. (1997): Plans as Situated Action: An Activity Theory Approach to Workflow Systems. In J.A. Hughes, W. Prinz, K. Schmidt and T. Rodden (eds.): *Proceedings of European Conference on Computer-Supported Cooperative Work ECSCW '97, Lancaster, UK. September 9–12, 1997.* Dordrecht, Netherlands: Kluwer Academic Publishers, pp. 17–32.

Bernard, H.R. (1988): *Research Methods in Cultural Anthropology*. Newbury Park, CA: Sage.

Blythin, S., J.A. Hughes, S. Kristoffersen, T. Rodden and M. Rouncefield (1997): Recognising 'Success' and 'Failure': Evaluating Groupware in a Commercial Context. In S.C. Hayne and W. Prinz (eds.): *Proceedings of International ACM SIGGROUP Conference on Supporting Group Work GROUP '97, Phoenix, AZ. November 16–19, 1997.* ACM Press, pp. 39–46.

Bowers, J. (1992): The Politics of Formalism. In M. Lea (ed.): *Contexts of Computer-Mediated Communication*. New York, NY: Harvester Wheatsheaf, pp. 232–261.

Bowers, J., G. Button and W. Sharrock (1995): Workflow from Within and Without: Technology and Cooperative Work on the Print Industry Shopfloor. In H. Marmolin, Y. Sunblad and K. Schmidt (eds.): *Proceedings of European Conference on Computer-Supported Cooperative Work, Stockholm, Sweden. September 10–14, 1995.* Dordrecht, Netherlands: Kluwer Academic Publishers, pp. 51–66.

Bowers, J. and T. Rodden (1993): Exploding the Interface: Experiences of a CSCW Network. In S. Ashlund, K. Mullet, A. Henderson, E. Hollnagel and T. White (eds.): *Proceedings of Conference on Human Factors in Computing Systems INTERCHI '93, Amsterdam, the Netherlands. April 24–29, 1993.* ACM Press, pp. 255–262.

Brooks Jr., F.P. (1995): *The Mythical Man-Month: Essays on Software Engineering 20th Anniversary Edition*. Reading, MA: Addison-Wesley Publishing Company Inc.

Button, G. and P. Dourish (1996): Technomethodology: Paradoxes and Possibilities. In M.J. Tauber (ed.): *Proceedings of ACM Conference on Human Factors in Computing Systems CHI '96, Vancouver, BC. April 13–18, 1996.* ACM Press, pp. 19–26.

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

Ciborra, C.U. (1996): *Groupware & Teamwork: Invisible Aid of Technical Hindrance?* Chichester, UK: John Wiley & Sons.

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

Dourish, P. and V. Bellotti (1992): Awareness and Coordination in Shared Workspaces. In J. Turner and R. Kraut (eds.): *Proceedings of ACM Conference on Computer-Supported Cooperative Work CSCW '92, Toronto, Canada. October 31–November 4, 1992.* New York, NY: ACM Press, pp. 107–114.

Dourish, P., J. Holmes, A. MacLean, P. Marqvardsen and A. Zbyslaw (1996): Freeflow: Mediating Between Representation and Action in Workflow Systems. In M.S. Ackerman (ed.): *Proceedings of ACM Conference on Computer Supported Cooperative Work CSCW '96, Cambridge, MA. November 16–20, 1996.* New York, NY: ACM Press, pp. 190–198.

Ellis, C.A. and J. Wainer (1994): Goal-based Models of Collaboration. *Collaborative Computing*, vol. 1, no. 1, pp. 61–86.

Fitzpatrick, G., W.J. Tolone and S.M. Kaplan (1995): Work, Locales and Distributed Social Worlds. In H. Marmolin, Y. Sundblad and K. Schmidt (eds.): *Proceedings of Fourth European Conference on Computer-Supported Cooperative Work ECSCW '95, Stockholm, Sweden. September 10–14, 1995.* Dordrecht, Netherlands: Kluwer Academic Publishers, pp. 1–16.

Glaser, B.G. and A.L. Strauss (1967): *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Hawthorne, NY: Aldine de Gruyter.

Grinter, R.E. (1995): Using a Configuration Management Tool to Coordinate Software Development. In N. Comstock and C. Ellis (eds.): *Proceedings of ACM Conference on Organizational Com-*

puting Systems COOCS '95, Milpitas, CA. August 13–16, 1995. New York, NY: ACM Press, pp. 168–177.

Grinter, R.E. (1996): Supporting Articulation Work Using Configuration Management Systems. *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, vol. 5, no. 4, pp. 447–465.

Grinter, R.E. (1998): Recomposition: Putting It All Back Together Again. In D.G. Durand (ed.): *Proceedings of ACM Conference on Computer Supported Cooperative Work (CSCW '98), Seattle, Washington. November 14–18, 1998.* New York: ACM Press, pp. 393–403.

Grudin, J. (1989): The Case Against User Interface Consistency. *Communications of the ACM*, vol. 32, no. 10, pp. 1164–1173.

Grudin, J. (1990): The Computer Reaches Out: The Historical Continuity of Interface Design. In *Proceedings of ACM Conference on Human Factors in Computing Systems CHI '90, Seattle, WA. April 1–5, 1990.* New York, NY: ACM Press, pp. 261–268.

Grudin, J. (1994): Groupware and Social Dynamics: Eight Challenges for Developers. *Communications of the ACM*, vol. 37, no. 1, pp. 92–105.

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

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 ECSCW '91, Amsterdam*. pp. 65–80.

Medina-Mora, R., T. Winograd, R. Flores and F. Flores (1992): The Action Workflow Approach to Workflow Management Technology. In J. Turner and R. Kraut (eds.): *Proceedings of Conference on Computer-Supported Cooperative Work. CSCW '92, Toronto, Canada. October 31–November 4, 1992.* New York, NY: ACM Press, pp. 281–288.

Naur, P. and B. Randall (1969): *Working Conference on Software Engineering. Garmisch, Germany, October 7–11, 1968.* NATO Science Committee.

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

Parnas, D.L. and P.C. Clements (1986): A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering*, vol. 12, no. 2, pp. 251–257.

Rochkind, M.J. (1975): The Source Code Control System. In *Proceedings of 1st National Conference on Software Engineering, Washington, DC. September 11–12, 1975.* IEEE Computer Society, pp. 37–43.

Samaras, T.T. and F.L. Czerwinski (1971): *Fundamentals of Configuration Management*. New York, NY: John Wiley & Sons.

Schmidt, K. (1997): Of Maps and Scripts: The Status of Formal Constructs in Cooperative Work. In S.C. Hayne and W. Prinz (eds.): *Proceedings of International ACM SIGGROUP Conference on Supporting Group Work GROUP '97, Phoenix, AZ. November 16–19, 1997.* ACM Press, pp. 138–147.

Sommerville, I., T. Rodden, P. Sawyer, R. Bentley and M. Twidale (1993): Integrating Ethnography into the Requirements Engineering Process. In A. Finkelstein S. Fickas (eds.): *Proceedings of Requirements Engineering 1993, San Diego, CA. January 4–6, 1993.* Los Alamitos, CA: IEEE Press, pp. 165–173.

Strauss, A. and J. Corbin (1990): *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. Newbury Park, CA: Sage Publications, Inc.

Suchman, L. (1987): *Plans and Situated Actions: The Problem of Human-Machine Communication*. Cambridge, UK: Cambridge University Press.

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.

Suchman, L. (1995): Speech Acts and Voices: A Response to Winograd et al. *Computer Supported Cooperative Work: An International Journal*, vol. 3, no. 1, pp. 85–95.

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.