Doing Software Development: Occasions for Automation and Formalisation

Rebecca E. Grinter
Bell Labs, Lucent Technologies, United States of America
beki@research.bell-labs.com

Abstract. The use of workflow technology has created considerable discussion within the CSCW community. Although the debates have been grounded in theories of work, less has been written about specific organisational and social settings where workflow systems have been used. This paper presents findings from an empirical study where a workflow-like system was in routine use for some of the work. It draws conclusions about the circumstances that made this possible.

Introduction

The use of workflow systems to support work has generated much discussion within the CSCW community (CSCW, 1995). Workflow technologies aim to reduce the complexity of coordination in three steps. First, the work activities are categorised; in other words, they are reduced to their basic form. This categorisation usually removes details and specifics from the work and reduces the actions to objects like artefacts, procedures, and user roles. Second, these categorisations are formalised as a language that specifies permitted interactions and rules out non-permitted ones. The formalism makes it possible to embed the categorisation into a computer system. Finally, some parts of this formalism may be entirely automated by the system.

It is these three steps of categorisation, formalisation and automation that raise questions for some researchers. Many of the voices in the debate have asked questions about whom workflow systems serve, how well different formalisms support work, and how they affect the environment they are used in (CSCW,

J Hughes et al. (eds), Proceedings of the Fifth European Conference on Computer Supported Cooperative Work, 173-188
© 1997 *Kluwer Academic Publishers Printed in the Netherlands.*

1995). Although the debate is highly grounded in theories of work, the discussions themselves have not focused so much on the empirical details about the companies and people using the technology and the kinds of uses these systems are being put to. Yet it is these details, as previous studies of groupware successes and failures remind us, that are so important to understanding the context in which the technology is embedded.

As the number of workflow systems grows and they are put into use inside companies a question arises as to what conditions support the successful adoption and on-going use of these kinds of systems. This paper examines three occasions when a workflow-like system helped reduce the complexity of work in a way that was helpful to those using the system. Specifically, it focuses on software development and the role of a workflow-like system — a configuration management tool — in supporting that work.

There is a growing body of literature that has examined the failings of groupware systems and as well as their successes. Within the CSCW community though empirical studies of workflow have tended to report the difficulties that people had working with them (Button and Harper, 1992). Even designers of workflow systems have also noted the difficulties of using these technologies (Abbott and Sarin, 1994).

However, despite these concerns workflow systems continue to be developed for a number of reasons. First, they continue to be a seductive technology to commercial corporations (Abbott and Sarin, 1994). Second, some popular management methodologies advocate the use of information technology to support work processes. Third, some successes have been achieved with workflow systems (Agostini, et al., 1994). Fourth, they present an interesting research challenge: to find ways to support the work of individuals in a useful and constructive manner (Medina-Mora, et al., 1992; Ellis and Wainer, 1994; Dourish, et al., 1996).

At the same time that workflow systems are continuing to be built, researchers are beginning to ask questions about the role of formalisms in collaborative work. For example, Bowers (1992) offers a number of counter-arguments against purely theoretical criticisms of formalisms. His counter-arguments also apply to workflow systems, and in this paper I shall provide some empirical support for two of them.

First, he argues that although some aspects of work maybe complex and uncertain other parts maybe routine and dull The build process that I shall describe was a part of software development work that the developers in this study found less interesting than their development work and used the tool to automate the process.

Second, he observes that if there is such contingency in work then why can't formalisms be used in contingent ways? This paper describes two occasions when the formalisms generated by the system produced outputs that allowed the developers to develop an awareness of what their colleagues were doing and

structure their activities in an uncertain environment. These examples highlight some uses of the system that support the contingent work of software development.

It is not my goal to say that workflow systems should always be used, or that they should never be used. Following the example of (Abbott and Sarin, 1994; Bowers, et al., 1995; Dourish, et al., 1996) I want to closely examine the potential of workflow systems for supporting work. Following the guidance set by Bowers (1996) at CSCW '96, it is time to "just go and see" whether there are occasions when workflow systems support work.

The paper begins with a description the domain of study, software development, the workflow tool used, a configuration management (CM) system, and the methods used to gather and analyse data. Next I describe the three cases when the tool supported and enhanced development work for the people using the system. Finally, some conclusions are drawn about why these occasions proved amenable to workflow systems.

Software Development and Configuration Management

In this section I describe the domain of study, commercial software development. I also describe one tool used to support software development work, a configuration management system. Configuration management systems have similar properties to workflow systems.

The Domain: The Development of Software Products

Modern software product development is a complex activity for four reasons. First, most commercially available software systems contain multiple components: pieces of software, libraries, documents, and utilities. These systems are built by a number of developers working on the same product simultaneously which creates coordination overhead (Brooks Jr., 1995). Moreover, software development is coordination-intensive as different parts of the system interact with each other, and so those responsible for these related pieces of code must continually align their efforts during development (Grinter, 1996).

Second, development usually consists of managing the development of multiple versions of the same product simultaneously. To appeal to enough customers most products need to work on a variety of different computer platforms and be compatible with different operating systems. 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 interact with these substrate technologies. Each time the product is tested the correct version of the system needs to be assembled which requires coordinating what goes into the product as well as what stays out

Third, software is an unusual product to develop because it is very malleable. Software can be changed by anyone, with the permission to do so, until the minute it is released. However, because of the dependencies between components of the system one change can potentially affect the functioning of other parts of the system. Developers working on other related parts of the software rely on a module of code behaving in the same way that it did before. Therefore developers need to coordinate their changes to ensure that they do not affect others' work.

Finally development times have dropped drastically in many sectors of the software industry. An increasing number of software development companies are under pressure to develop new releases of their products in much shorter times.[1] This compounds the previous problems by requiring that they all get resolved much more quickly. For all of these reasons software companies have looked for systems to help them organise their development environments. One class of tools that increasing numbers of development companies have turned to are configuration management (CM) systems.

The Tool: Configuration Management Systems

CM systems were designed to help companies who develop software organise their development environments by helping them to track the relationships among components, develop multiple versions simultaneously, and control changes made to software. The first generation of CM tools 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.[2] When a developer had a piece of code checked out no-one else could make changes to it.

These systems had two disadvantages for commercial software development. First, they only worked for modules of code. Software systems contain more than just software, including libraries, test suites, and documents. Modern CM systems support these different types of components. Second, the check out state turned out to be very limiting because it prevented others from changing the same module at the same time. 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 tools also address the problems of understanding the relationships among code and developing multiple versions by providing three other classes of functionality usually called *layers* (Caballero, 1994). The "configuration control"

---

[1] In my contacts with managers at various companies it was not uncommon for them to remark that systems development times had been cut in half during the last five years, while the number of variants necessary to be compatible with different hardware and software configurations has risen

[2] Two early systems were Revision Control System (RCS) and Source Code Control System (SCCS) (Rochkind, 1975, Tichy, 1985).

layer maintains information about which components make a software product. 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. The configuration control level allows developers to find out exactly which components belong to a specific hardware configuration.

The "process management" layer provides a "life cycle" for each type of component in the system. The life cycle consists of different states; for example, software can be checked-out, checked-in, unit tested, system tested, and released. When software is in these different states, certain people — who have corresponding roles — are permitted by the system to manipulate the software. For example, in the unit tested state only people assigned the testing role can access the component.

The "problem reporting" layer supports bug and enhancement tracking. All changes to the system components arise as a result of bugs being reported or enhancements being requested. These bugs and enhancements — collectively known as problems — follow a life cycle. The problem reporting layer also relates the problems with the software components that were changed.

CM systems are domain-specific workflow systems for the coding and testing parts of software development. They allow the formalisation of software development by categorising the artefacts and people involved in the work. Furthermore they support certain kinds of relationships among people and artefacts and only permit the artefacts to follow certain state transition models. Finally CM systems automate some elements of the work entirely.

Sites and Study Methods

The data reported in this study were gathered in 1994-1995 as part of a series of studies about configuration management. In this paper the data are drawn from two sites.

Tool Corp. is the vendor of a CM tool. They use their own CM tool to manage the development of the next versions of the tool itself. During the time I spent at Tool Corp. the development group varied in size from 14 to 18 people. The developers were all co-located working on one floor of one building. The product they built consisted of about 1 million lines of code.

Computer Corp. is the vendor of a computer operating system for specialised hardware. At the time of the study they employed 700 people to develop their software. Many of the software developers work at the company's headquarters in Silicon Valley, but they have developers located in other states, and other countries. The software consists of about 10 million lines of code. Computer Corp. had just started using Tool Corp.'s product in their software development work.

At both companies the development of the overall product was broken up into development teams that were organised by functional parts of the product. At

Computer Corp. there were many teams organised around the different operating system functions; for example, a kernel team, and an interface team. 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 which I used for three and a half months. I also had access to the tool and the software development environment provided through the system. 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.

At both sites I was able to conduct interviews. 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.

The data were analysed using grounded theory techniques (Glaser and Strauss, 1967; Strauss, 1987; Strauss and Corbin, 1990). Grounded theory consists of three stages. The first stage consists of analysing the raw data. The purpose of the analysis is to find as many conceptual groups — known as *categories* — that describe the events and phenomena in the data. The second stage focuses on filling in the categories; for example, characterising their properties, the events that caused them to happen, and any resulting consequences. Finally the third stage puts these substantive categories together to form the theory of action itself. Data analysis consisting of these three stages happens numerous times over the course of the study itself. Intitial gaps and questions surrounding the categories drive further periods of data collection.

In this study the initial cycles of gathering and analysis focused on the observational data gathered. As the theory started to form I used interviews to help inform and revise the categories developed. In the next three sections I describe the examples of the developers' tool usage. Discussion of these cases is deferred until the following section.

Case 1: Automating The Build

During the course of development, software components need to be gathered into the product to see whether they work together and function as intended. The process of putting the system together from the components is known as "the build." The time between successive builds decreases rapidly as development enters the final stages because the closer the product gets to release the more everyone wants to be absolutely certain that the software works.

The actual process of building the system consists of finding the latest changes of all the components and then compiling them. At both sites each development team had a developer known as a build manager who took responsibility for the job of ensuring that the build took place. However, there was a significant difference between the work that build managers did dependent on whether they used the tool or not.

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 take responsibility for the build as well as their development work. That person 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 organise 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

and another 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 .. [3]

There is some benefit for doing this work. The build managers of the systems often end up knowing a lot more about the overall state of the system than the developers. Their continuous interactions with all the developers, to gather the latest changes of code, mean that the build managers have an up-to-date impression of what state the overall system is in.

Despite this advantage, the build managers do not enjoy the work of build management. Although they recognise the advantage of know what's going on, it does not compensate for the amount of time that it takes to do the work. 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 every two days or once a week. Build managers also dislike doing the job because they do not find it as challenging as their software development work.

Other teams at Computer Corp., and all the teams at Tool Corp. let the CM tool do the build. On command, the system would gather the latest code changes from everyone in the team and compile them. If the compilation was successful the tool

---

[3] Notes in square brackets represent references to artefacts and processes that might identify the company, so I have replaced them with more generic terms to maintain confidentiality

would produce a new version of the product for testing and using in further development efforts. Otherwise the system would notify the build manager and provide them with information about where the problem occurred. The build manager would inform the developer responsible for the build failing.

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.

The tool reduced the amount of time it took to gather everything for the build. Also, it helped the build managers find the error that broke the build. Both of these activities are part of a manual build process but when automated improved the build managers' job.

The Developers

The developers also liked the automated procedure because of the guarantees it provided them. As one developer said,

> I come in 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 its going to be there and its going to work fine. Then I can just go about my business, having gotten everyone else's changes automatically

Developers using the CM tool enjoyed the fact that the tool gathered their latest changes automatically, without them having to stop working. Furthermore it gave them everyone else's latest changes. The developers also described two weaknesses with the manual methods that affected their work. First, the time taken to manually gather the changes invariably meant that some developers had made revisions to their code since the build manager visited. This caused time delays in testing whether their new code worked, which were especially problematic during the final stages of development when teams wanted feedback very quickly. Second, changes often need to be tested in groups, as they represent a revision to the functionality of the sub-system. When only some of those changes get into the build, there's a high probability that the build will fail because the code depends on the other changes being there. 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 failed build slowing down the testing process.

The build is one example of a case when automation benefits the build managers as well as the developers of the software. In this case the automated build generated positive reactions from the build managers because it reduced their workload and from developers because it supported their work. The trade-off

between the formalisation of the build versus doing the build manually was acceptable to all using the tool.

Case 2: Awareness Created by the Formalism

In a paper about ways of creating awareness of others in CSCW systems Dourish and Bellotti (1992) describe an approach to collaboration they call *shared feedback*. Awareness of what others are doing is provided through feedback presented in a shared workspace. The CM tool provided this kind of awareness to the developers through the formalisms that the tool used to categorise the work of software development.

When the tool starts up the developers see the main view. To visualise the main view imagine a typical Mac folder with the files viewed by name. Each file in that folder has a name, size, type, and its date of creation. The main view of the CM tool has the same visual arrangement with the software component name, 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, which correspond to folders of sub-system components. The main view that the developer sees when she launches the tool is the one corresponding to the folder — or as the developers call it the *directory* — of files she is currently working on, but it's very unlikely that she'll 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.

The states that components move through were known to all the developers using the tool at both sites. This allowed the developers to read the information from the view and infer that their colleagues were working on certain components related to theirs, whether they were developing or testing them. Furthermore this view was not static. As developers changed components so their state changed in the life cycle and the version number was incremented by the system.

The CM tool allowed developers to update their main view at any time they wanted to, a process they called *reconfiguring* the view. I was alerted to the awareness created by the system when the developers talked about seeing things in the system. 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 to make developers aware of the work going on around them. However, they also used the information to direct their own efforts. One

thing that the developers did not enjoy doing was working on the same piece of code at the same time. Merging the two versions made back together turns out to be very 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 awareness provided by the main view comes from the tool's formalism. It would be impossible to display all the details of others work in a view like this, and for most of the developers it would not be useful. Often times they want a peripheral awareness of what others are working on, and when they need more they establish contact with their colleagues. The formalism provides them with a useful summary which often meets their needs and when it doesn't gives them a pointer as to who to contact.

Case 3: Tracking Problems

The CM tool that Tool Corp. and Computer Corp. used had an integrated problem reporting facility. Problems also had a life cycle starting from when they were entered into the system by managers or customer support. Every few days a team of people that consisted of the project manager, a tester and some senior developers, would meet and prioritise the problems in the system and assign them to developers. Once a developer has been assigned a problem the tool notifies the developer who has been assigned the problem.

Although the problem reporting facility sounds restrictive, the developers relied on it to tell them what they were supposed to be working on. The problem reporting facility thus became a scheduling system of sorts,

> It's nice, you come in 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 facility requires that all changes to the code must be associated — via hypertext links — to problems in the facility. This created a problem because the people who met to assign problems to developers could not meet frequently enough to prevent the developers from running out of problems and not being able to get any more work done until the next meeting.

The problem 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 assigning the developers a supervisor role in the facility. This work around

let the developers carry on using the system. Moreover, it had the unintended payoff of enhancing their use the problem reporting facility as a scheduling tool for organising their own work.

> 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.

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. Also 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.

The problem reporting facility was used by developers 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 place to put their notes about future work and a comprehensive scheduling system.

Discussion

These three cases are occasions when the workflow system — the configuration management tool — supported the everyday work of the developers. It is important to stress that this is not a recommendation that workflow technologies are universally applicable. Specifically, I have written about the limitations and failings of the same tool in other places (Grinter, 1995; Grinter, 1996). More generally, others have written about the failure of other tools to provide support for software development work (Button and Sharrock, 1994). However, in the three cases I describe the tool demonstrably supported the work of the developers and provided them with new opportunities to organise their actions in a changing environment. This section offers four reasons why the tool worked on those occasions: (1) understanding and accepting a model of work, (2) providing understandable and useful representations, (3) automating the "right" work, and (4) having a supporting company.

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 really use the

tool on these occasions. This became very explicit at Computer Corp. where developers were beginning to adopt the tool in their work.

At Computer Corp. I took part in a class where developers who were new to using the tool were learning about how the tool modelled the work of software development. During the adoption phase of the tool developers at Computer Corp. found that their old models of software development — usually based on other tools that they had used for configuration management and other professional sources — clashed with the new tool. Computer Corp. designed the class to help ease the adoption process by explaining the differences and similarities between the new tool and the old ways of work. It was an attempt to get the developers to understand and accept the model of work.

At both companies 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 and how it fit into their work. The developers also believed in the model enough to talk about their usage in positive ways as described in the three cases. For a workflow system to work, for any groupware system to work, both stages must occur.

Perhaps one reason why developers at both sites found the model acceptable enough to use the system stemmed from the fact that the developers had a good user model. The developers at Tool Corp. were both designers and users, perhaps the ultimate participatory design experience. The developers at Tool Corp. could use their own experiences of development to build a system that worked for them and their counterparts at Computer Corp.[4]

Understandable and Useful Representations

Developers used the main view and problem reporting facility in part because the tool provided an understandable and useful representation of the underlying model of work. By representation, I mean: the interface, the presentation of the content inside the windows, and way that the system updates that content. The main view relies on everyone understanding what they are seeing. 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

This representation is maintained throughout the tool. Even when the developer changes their main view the new view they arrive at contains the same types of

---

[4] 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 The problem reporting facility, for example, contained a model of problem assignment that seems more appropriate for hierarchically organised software development In my time at Tool Corp I observed potential customers being attracted to this hierarchical model

information and it still means the same thing.[5] To be used as a mechanism for peripheral awareness the developers had to trust the system to be telling them the same thing where ever they were in it.

However the main view was also useful because of its ability to provide peripheral awareness to the developers about the others' actions. The formalism provided that shared feedback because it reduced the details of peoples' work to a brief, consistent, and above all useful form. In the absence of the formalism it would have been very hard for the developers to find out what their colleagues were working on in such a concise way.

The use of the problem reporting facility also relied on it presenting information in an understandable and useful way. Beyond that the system was useful enough so that when they were given the opportunity to utilise more parts of it they did. When the developers had control over the scheduling functions of the problem reporting system they found it even more useful.

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.

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.

A Supporting Company

At both Tool Corp. and Computer Corp. developers are viewed by managers as professional staff. This was visible in a number of ways. First, I did not see anyone keeping time records. At both companies it was assumed that the developers were putting in the required time unless they told someone otherwise.

---

[5] This is not an argument for making all interfaces consistent and arguments suggest that it can be problematic (Grudin, 1989) Instead the observation 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

Second, the developers were treated as professionals by their management. Finally, and most significantly, developers' opinions were taken seriously by their managers.

This kind of culture made it relatively easy for the developers at Tool Corp. to discuss their initial problems with the problem reporting facility and get permission from their management to simply change the role of developer into the role of supervisor 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 change is impossible to make.

Workflow system builders can not pick the companies that they sell their systems too. However, we can assume that companies who buy these kinds of systems would like them to function as intended, and potentially increase the efficiency of the work done. If the users can not use them or end up spending time working around the system to accomplish their work then these efficiencies will be much harder to attain.

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 few empirical studies of workflow systems in use. Those 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 workflow 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 supporting company. These reasons highlight the *context-dependent* aspects of the use of workflow technologies in particular settings.

These empirical results provide a basis for grounding otherwise abstract and theoretical discussions about workflow technologies. In addition to talking about the theoretical hurdles to implementing workflow it is time to find out when the confluence of these positive forces creates opportunities to implement workflow rather than watching it happen badly without us.

Acknowledgements

I would like to thank the Engineering and Science Research Council for their financial support during the time I conducted these field studies. Many people have commented on this paper and I would like to thank Jim Whitehead, Jim Herbsleb, Neil Harrison, Jonathan Grudin, Paul Dourish, Peter Danielsen, Victoria Bellotti, Al Barshefsky and the anonymous reviewers for their suggestions This work was inspired by Gunn, Megson, and Wark

References

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

Agostini, A , De Michelis, G , Grasso, M A. and Patriarca, S. (1994) "Re-engineering a business process with an innovative workflow management system. a case study", *Collaborative Computing*, vol 1, 1994, pp. 163-190.

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

Bowers, J. (1992) "The Politics of Formalism", in Lea, M (eds.). *Contexts of Computer-Mediated Communication,* Harvester Wheatsheaf, New York, 1992, pp. 232-261.

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

Bowers, J (1996) "PANEL From Retrospective to Prospective: The Next Research Agenda for CSCW", in Ackerman, M S (eds.): *Proceedings of ACM Conference on Computer Supported Cooperative Work CSCW '96 , Cambridge, MA,* ACM Press, pp 440

Brooks Jr., F.P. (1995). *The Mythical Man-Month Essays on Software Engineering*, Addison-Wesley Publishing Company Inc., Reading, Massachusetts.

Button, G and Harper, R H.R. (1992). "Taking Organisation Into Accounts", in Button, G (ed): *Technology in Working Order,* Routledge Press, United Kingdom, 1992, pp. 98-107

Button, G. and Sharrock, W (1994): "Occasioned Practices in the Work of Software Engineers", in Goguen, J. and M Jirotka (eds): *Requirements Engineering,* Academic Press Ltd , London, United Kingdom, 1994, pp. 217-240.

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

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

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

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

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

Grinter, R (1995) "Using a Configuration Management Tool to Coordinate Software Development", in Comstock, N and C. Ellis (eds.) *Proceedings of ACM Conference on Organizational Computing Systems , Milpitas, CA, August 13-16, 1995 ,* 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, 1996, pp 447-465

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

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

Rochkind, M J. (1975) "The Source Code Control System", in (eds.) *Proceedings of 1st National Conference on Software Engineering , Washington, D C., September 11-12, 1975,* IEEE Computer Society, pp. 37-43.

Sommerville, I., Rodden, T., Sawyer, P., Bentley, R. and Twidale, M (1993)· "Integrating Ethnography into the Requirements Engineering Process", in Finkelstein, A and S Fickas (eds.) *Proceedings of Requirements Engineering 1993 , San Diego, California. 4-6 January,* pp 165-173.

Strauss, A (1987) *Qualitative Analysis for Social Scientists*, Cambridge University Press, New York, New York.

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

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

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