Work as coordination and coordination as work: A process perspective on FLOSS development projects

Kevin Crowston[1], Carsten Østerlund[1], James Howison[2] and Francesco Bolici[3]

crowston@syr.edu, costerlu@syr.edu, jhowison@cs.cmu, francesco.bolici@eco.unicas.it

[1]Syracuse University
School of Information Studies

Syracuse NY USA

[2]Institute for Software Research
Carnegie Mellon University

Pittsburgh PA USA

[3]Dipartimento Impresa e Lavoro
Università degli Studi di Cassino
Cassino Italy

Work as coordination and coordination as work: A process perspective on FLOSS development projects

Abstract

Coordination in work teams has been a topic of perennial interest in organizational studies. The starting point of much of the literature is a conceptual separation between the work itself, on the one hand, and activities undertaken to coordinate the work, on the other, labelled articulation work or coordination mechanisms. We suggest that the analytic duality between work and coordination arises in part from an information processing perspective on work that assumes an input, process and outcome model. In this perspective, it is natural to consider the process and coordination of the process as separate activities. However, in contrast, one can envision work processes as clustering around the outcome of the work. Each process contributes to the evolving work at hand but at the same time takes its clues for what to do next (i.e., input) from the evolving work product. In this case, the separation of discrete coordination and work events dissolves in a process perspective, for which analyzing interactions rather than self-standing actions is preferred. Out of this conundrum arises the question: how can we understand coordination as an inseparable part of work and the outcome of work itself?

The paper investigates this question in the setting of free and open source software (FLOSS) development projects. In FLOSS projects, we find little evidence of overt coordination of development activities. Rather, we argue, coordination takes place primarily through the work itself, i.e., through the shared software source code. We present evidence showing how the software code posted by FLOSS developers serves as an outcome of their work and a coordination device at one and the same time. This approach is supported by socio-material nature of software code itself and by commonly adopted approaches to software development in the FLOSS community.

To explain these results theoretically, we draw inspiration from the literature on documenting work demonstrating how documents can be both model *of* work and models *for* work. Extending this perspective we argue that work itself can serve not only as a model *of* work but more importantly a model *for* future work. Specifically, we argue that work outcomes, e.g., software code, can serve as a model for work by 1) invoking typified actions in response to recurrent situations, 2) being part of larger formalized structures of legitimate work activities, and being 3) visible, 4) mobile 5) and combinable.

Work as coordination and coordination as work: A process perspective on FLOSS development projects

Introduction

Coordination in work teams has been a topic of perennial interest in organizational studies. The starting point of much of the literature on coordination of work is a conceptual separation between the work itself, on the one hand, and activities undertaken to coordinate the work, on the other. These two concepts are sometimes labeled "work" versus "articulation work" (e.g., Gerson and Star 1986; Strauss 1985) or "tasks" versus "coordination mechanisms" (Malone and Crowston 1994).

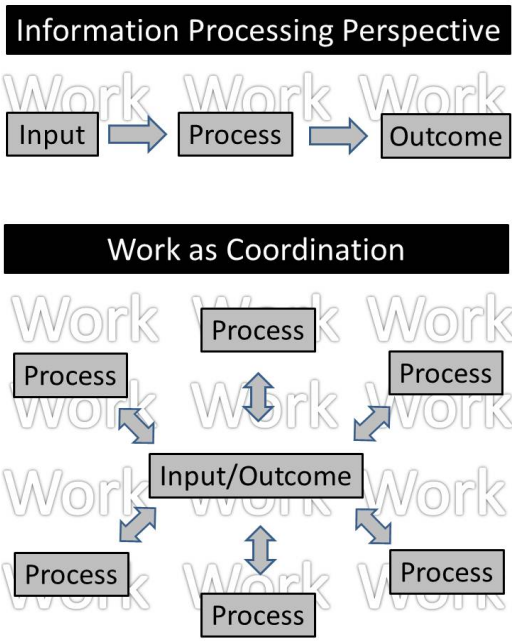

Figure 1. Contrasting an input-process-output view of work with a view of work as coordination.

The duality between work and coordination arises in part from an information processing perspective on the work that assumes an input-process-output model of the work (the upper half of Figure 1, see Ilgen et al. 2005 for a review). In this perspective, it is natural to consider the tasks that create the output (connecting inputs to outputs) as the main part of the process and to conceptualize the coordination mechanisms as separate from this work. In contrast, one can envision work processes as clustering around the outcome of the work (the lower half of Figure 1). Each task contributes to the evolving work at hand and at the same time takes its clues for what to do next (i.e., its input) from the evolving work product, the output of other tasks. In this case, the separation of discrete tasks and coordination

mechanisms dissolves in a process perspective, for which analyzing interactions is preferred to self-standing actions. The perspective also draws our attention to the importance of sociomaterial structures when it comes to articulating the entwined nature of work and coordination. Out of this conundrum arises our research question:

> *How can we understand coordination as an inseparable part of work and the outcome of work itself?*

To address this question we investigate an empirical setting that facilitates the exploration of alternatives to this dual conceptualization of tasks and coordination mechanisms, namely free and open source software (FLOSS) development projects. Coordination has been a perennial topic in empirical software engineering as well and the same conceptual split between work and coordination of work is clear in the software engineering literature from Conway (1968) through Cataldo and colleagues (e.g., Cataldo and Herbsleb 2008).

In this context we present a study of communication and coordination in two successful FLOSS projects. Our analysis has two parts. First, following the information processing perspective, we present an empirical study in which we searched for activities and, in particular, coordination mechanisms managing the work of FLOSS developers. Here, we define coordination mechanisms as work that manages dependencies between the tasks rather than contributing directly to the output of the process. However, after detailed analysis of data from the projects, we found little evidence of overt coordination of the development tasks. FLOSS developers seem to rarely communicate overtly about their coding tasks. Interestingly, on the occasions when they do talk, they often refer directly to the outcome of their work, the software code.

In the second half of the paper we address this lacuna by developing a theoretical perspective on work processes that shows how coordination can take place through the subject of the work itself, i.e., through the shared software source code in a FLOSS team. Drawing inspiration from studies of documenting work we argue that work outcomes, e.g., software code, can serve as a model for work by 1) invoking typified actions in response to recurrent situations and being part of larger formalized structures of legitimate work activities, and being 2) visible and mobile and 3) combinable.

The paper is structured as follow: First, we start out with a background description of software development tasks and coordination issues to set the stage for the following analysis. Second, we outline the method we used to study FLOSS work. Third, we apply an information processing perspective to the FLOSS cases and look for evidence of coordination mechanisms used. Fourth, in response to the case findings, we develop a view of work as coordination and apply it to the empirical cases. Finally, we discuss the findings and draw implications for research and practice.

Background: Tasks and coordination of software development

Because an understanding of software development is helpful background for understanding our data collection and analysis, we start by presenting a basic overview of the tasks and necessary coordination of software development work before describing the study and our proposed theory in more detail. Software development (in general and in the case of FLOSS more specifically) involves two intersecting cycles of work, one public and shared (for anything more than a personal project) and one private and individual, as shown in Figure 2. The two cycles intersect in the code base, e.g., the files containing the source code for the application, supporting files necessary to compile the source code into a running application and often some amount of documentation, e.g., instructions for compilation or a "README" file describing the files in the directory. A defining characteristic of FLOSS projects is that these source code files are made available to the general public in addition to the developers (hence the description "open source").

In larger-scale software development, these files are stored in some kind of source code control system (SCCS), such as CVS or Subversion. (In the discussion that follows, we give only a brief overview of SCCS functionality, omitting a number of details.) To start working on a project therefore, a developer clones the code base, meaning that s/he downloads a complete copy of the files from the SCCS to a local computer. The local copy of the source files can then be edited to fix bugs or to add new functionality. As shown on the left side of the figure, this local development involves cycles of design, editing the code, testing the application and debugging the changes (e.g., by examining program logs, traces of dumps). This development work is generally done in private, though a developer might discuss or seek help on a problem while coding.

When the developer is satisfied with the changes, s/he "commits" the changed files, which means sending them to the SCCS hosting the code in the form of a "patch", along with a short description of what was changed. If the developer does not have permission to change the files in the SCCS (i.e., does not have "commit privileges"), s/he must send the patch to a developer who does; that developer can then commit the changes if s/he agrees with them. The exact details depend on the SCCS used, e.g., some SCCS, such as git, do not require a central server and instead allow developers to send patches directly to each other; some can record commits for multiple files while others require each file to be committed separately.

When a change is committed, the SCCS integrates the changed files into the stored files, along with other patches from other developers. Unless two developers have changed the exact same lines of code, this integration can be done automatically, though there can still be problems (e.g., the name of a subroutine changed in one patch, but called with the old name in another). The new versions of the code base is then available to all other developers to build on, further test or use (starting the cycle on the right of the figure). However, the SCCS stores a record of all changes, allowing them to be "reverted", that is, to be returned to

an earlier version, e.g., if a problem emerges in the new code. The SCCS can be configured to email all developers when a commit is done, making them aware of the changes. SCCS also support “branches”, storing different versions of the files with different sets of changes. Branches can be used to share experiments on the code, while keeping those experiments from interfering with the main release (called the “trunk”). If the experiments are successful, the branch code can be merged back into the trunk, though this likely will require some manual effort.

Periodically, the project makes a “release”, meaning a version of the code that is believed to be in a consistent state is captured and packaged for distribution. The release is generally accompanied by “release notes” that briefly summarize the changes made since the last release (often simply a list of the description of the major SCCS commits included in the release). Depending on the intended audience, a binary version of the program may also be released so non-developers can use the program without having to compile it themselves. FLOSS projects are encouraged to “release often and early” in order to get feedback from users on the latest features and to more quickly find bugs in the program (Raymond 1998).

Users and developers who find bugs in the program can report them on a bug tracking system. A separate tracker may record suggestions for improvements to the program or other issues. Trackers capture basic data about the bug or issue and allow discussion, e.g., a developer might post a request for further information to which the user can respond, or propose a possible technical solution for debate. Developers do not have to use either tracker, since they can simply change the code themselves to fix any bugs they find or features they want to add. However, some projects encourage everyone to use the trackers in order to have a record of the bugs found or features requested and how they were handled. Projects also have mailing lists on which developers and users can interact, sometimes together on one list, sometimes on separate lists for developers and users, though typically anyone can join and post to either. Ideas from the trackers or lists may inspire a developer to further work on the project, thus restarting the cycle of development.

Analysis of the software development process suggests numerous possible dependencies that need to be managed, suggesting the need for some coordination mechanisms in the process. Since all of the developers work on the same codebase, there is a dependency between their work, requiring mechanisms for resource sharing. The code itself has many dependencies, as different pieces of the code are interrelated (e.g., a function that calls other functions or uses shared data). Changing one piece of code can affect these relationships (e.g., changing a function will require changing all places that call that function). An important dependency in most processes is between a task needing to be done and someone to work on it, requiring mechanisms for task assignment (e.g., a bug report might be assigned by a development manager to a developer to ensure that it is fixed by a developer, and not more than one developer).

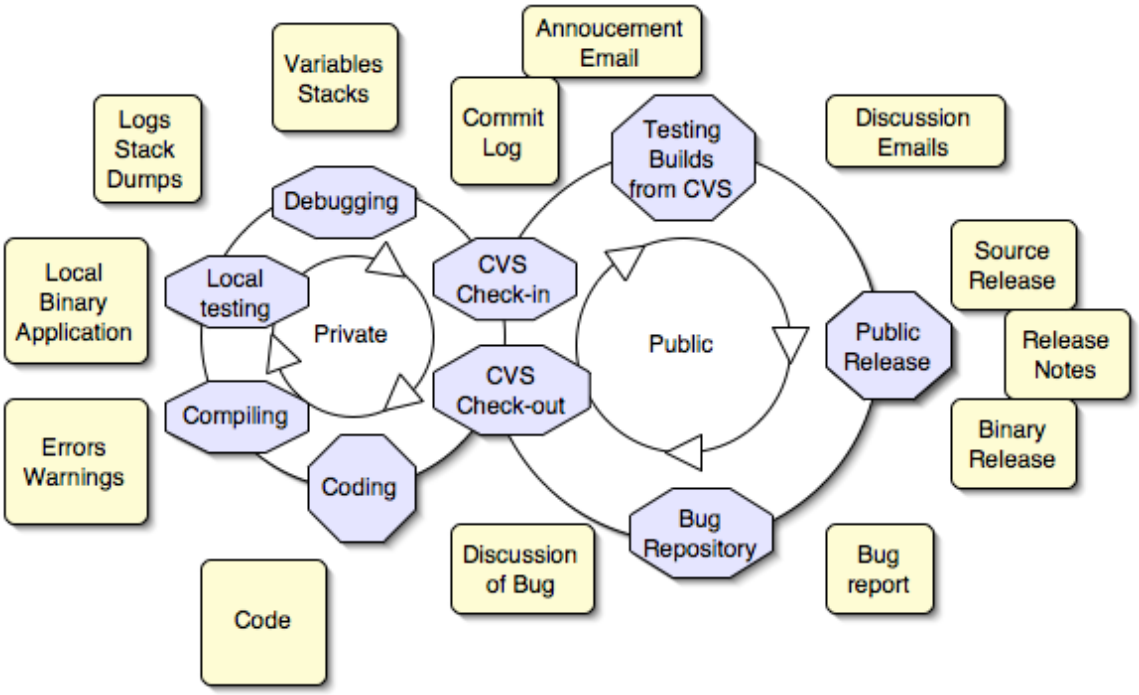

Figure 2. Private and public cycles of software development work.

Method

In this section we describe our data collection and analysis approach. The goal of this analysis was to describe the mechanisms adopted to coordinate the tasks in software development teams.

Context: FLOSS development projects

We set our study in the context of two free/libre open source software development projects. The choice of FLOSS was made for both pragmatic and conceptual reasons. Pragmatically, we sought an environment in which we could obtain data for our study. Research on FLOSS is enhanced by the availability of copious archives of project activity. Although it is somewhat dependent on individual projects, the bulk of such communications are recorded in openly available archives, both for convenience but also in part in fulfillment of an ideology of openness and transparency.

Furthermore, we sought some assurance that we could obtain a complete record of the communications. In a conventional development team, communication for coordination might be carried out face-to-face in meetings, offices and corridors, and thus might not be recorded. Such uncollected communication would be problematic for an inquiry into the nature of coordination. For this reason we sought evidence from teams that operate in a completely distributed mode, with little or no face-to-face interaction, making it more likely that the repository provides a complete record of the interaction. Many FLOSS development

teams have this characteristic (though not all, e.g., Crowston et al. 2007b). Those that do can be described as community-based, without a shared geographical center and in which collaboration is entirely through computer-mediated communications.

Conceptually, FLOSS projects are an interesting setting in which to study coordination. The community-based FLOSS environment brings with it additional organizational features, such as largely volunteer participants and a lack of a formal, shared organizational existence. As a result, these projects face the challenges of coordinating action in distributed environments, with substantial numbers of volunteers, changing and fuzzy lines of authority and limited or no access to traditional channels for *ad hoc* coordination, such as face-to-face meetings or even telephone. FLOSS is often held as a model success for distributed, innovative work (Harrison 2001; Stewart and Gosain 2006), though care must be taken to consider how the context of FLOSS development impacts the generalizability of findings.

Data collection

For data, we drew on a data set created by Howison (2009) in his study of collaboration patterns in FLOSS development. That study gathered data from two comparable FLOSS projects, Fire and Gaim. The projects were selected for their similarity, rather than their differences. Both projects were relatively successful community-based projects without a geographical center; neither project conducted conferences or "sprints" and there is no reason to believe that any participants were co-located. Because both projects produce similar software, namely multi-protocol IM clients, we anticipated finding similar patterns of development and needs for coordination in the two projects.

The primary data for the study came from direct records of the work of the developers as recorded in publicly available data sharing repositories for research, including FLOSSMole (Howison, Conklin and Crowston 2006) and the Notre Dame SRDA (Madey 2009). For each project Howison collected 1) source code repository data, including commit messages; 2) release data, including release notes; 3) mailing lists; and 4) tracker discussions. The dataset does not include data on IRC or Instant messaging between participants, nor email sent between participants privately. Data was collected for two inter-release periods, chosen to include periods that both projects were highly active and successful and each around two months in length (Fire, 56 days; Gaim, 61 days). To facilitate interpretation of the results, this direct evidence of the developers' actions was augmented with an informal interview with a developer from one project.

Data analysis

The goal of the analysis was to identify the work done to carry out project development tasks and the coordination mechanisms involved. From the available data, Howison reconstructed the processes leading to new program features. FLOSS teams do not

always record which archived evidence pertains to which tasks undertaken by the teams (e.g., a code commit might not indicate which bug is being fixed). Therefore Howison manually inspected the archives and re-organized them to understand the tasks and the contributions made to them by developers. The process is time-consuming and laborious but provides stronger validation and understanding of the material than automated heuristics.

The method of reconstruction was as follows (Howison 2009). Projects record the results of their development efforts in two main locations: the release notes and the README file in the source code repository. The outputs listed in these sources formed the basis for re-organization of the evidence, providing 60% and 30% respectively of the outputs identified. The remaining 10% were identified by reading SCCS commit messages to find commits that were not described in the README or release notes (starting with the SCCS messages would have been more difficult, as a single output might include multiple commits to different files).

These identified outputs formed the basis for a free text search of the full data collection for the release period (and extending outside the release period for tracker items in case the bug or issue had been raised earlier). Relevant documents (e.g., bug report discussions, email messages or code commits) were assigned to episode to which they seemed to contribute. Documents could be assigned to more than one episode. The documents were arranged in chronological order and the individual actions for which they provided evidence were extracted. Individual documents could provide evidence of more than one action (e.g., a commit message that thanked a contributor for a patch that was being committed on their behalf would provide evidence for both the work of the contributor and the commit by the developer). Conversely, multiple documents could be merged into single actions (e.g., SCCS commits close together in time could be grouped as one commit of multiple files). The various identifiers used by participants for different archives (e.g., Sourceforge username vs. Real Name/Email address combination) were examined and merged in order to be able to attribute the work to the individual who performed it.

The actions were then classified according to their type of contribution towards the task outcome, using a simple scheme: 1) production work, 2) review work, 3) supporting work and 4) documentation work. Production work was that work which directly produced changes to the shared output of the project, almost always involving changes to the source code in the repository. Review work was assessed when participants checked code in on behalf of others, or provided critique and feedback of other's commits. Supporting work included both user requests and bug reports, and also developer support such as asking or providing help solving programming issues or explaining parts of the code to assist other developers. Documentation work involved creating documentation for the system.

Through the analysis described above, Howison (2009) identified 103 work episodes leading to new features or changes in the code, comprising a total of 786 actions taken by

developers. We took this dataset as our starting point for our analysis of the coordination of the projects.

Input-process-output view of work: Case results

In this section, we present the evidence regarding the coordination of the work revealed through the data collection and analysis described above. Our first step was to examine the number of developers who contributed to the outputs, since while there are likely dependencies between separate tasks, tasks in which more than one participant undertook production work seemed most likely to have dependencies and therefore the most pressing coordination requirements. Participants in such shared work are most likely to need to actively coordinate and to do so via communication, either in advance to plan roles or during work since the work is continuing.

Howison (2009) found that only a single participant undertook production work in 83 of 103 the cases. Only in 20 of the episodes (10 in Fire and 10 in GAIM) did more than 1 participant perform production work (i.e., coding). In other words, only a fraction of the episodes are ones in which the software development activities have been accomplished through collaboration between two or more developers, these 20 episodes are referred to as co-work episodes. In this way Howison (2009) reports that despite the shared and highly interdependent output of the projects, much of the work appears to be done individually rather than collectively.

We therefore focused our analysis of coordination on the 20 co-work episodes. We took these as the critical cases and examined them further for coordinating communication. Specifically all the actions in the episodes were examined for evidence of communication between the developers contributing to the task outcome. We coded the presence of direct communication about the tasks to accomplish, the plan of the activities or any other form of management of dependencies between activities within the task.

The result of this analysis was quite surprising since it shows an absence of direct communication for coordination purpose in most of the episodes (14 of 20). In total, in only 6 episodes (out of 103) was there direct evidence of visible coordination of work; the rest were done by two or more individuals but without explicit coordination. So, a first surprising result of our analysis is that even in episodes with co-work, direct communication and coordination is uncommon (and indeed, is rare across all of the episodes).

Because of the small number of co-work episodes in which there was visible communication, we will discuss them in some detail, while noting that these examples are the exceptions rather than the rule. The first example of visible communication between developers about the next actions to perform can be found in Fire episode 32_3, in which two actors (*gbooker* and *jtownsend*) co-developed a new feature (AIM buddy blocking).

gbooker[1], who seems to drive the implementation of this specific feature, committed new code together with an SCCS log message that reads:

> <*gbooker*> ... Once we get the notification change about the pref change for allow those not in buddy list, we will be good to go!!

Four hours later, *jtownsend* posted new code with the message:

> <*jtownsend*> add[s] notification of block non-buddies pref changing

Analyzing the communication and then the changes in the code, it seems that *gbooker*'s statement ("Once we get the notification ...") was in fact a polite request for a new feature and indicates direct communication to resolve a dependency that he saw on work he hoped or expected to be done by *jtownsend*, the completion of which would allow *gbooker* to drive his own work forward.

Another episode which includes direct communication is Fire episode 21, in which *gbooker* emails to the user list in reply to a user's request for information, asking for support from the community and then proposing a set of features to be developed. After a day, *jtownsend* replied pointing out his idea about the merging of two components and proposing a common plan for the subsequent activities:

> <*jtownsend*> [the merging] could happen within the next two week possibly. The main issue is I want to add MSN file send, and improve the interface consistency with the Yahoo file receive, but these things should be done with the last file transfer infrastructure that Graham has in the branch

Through direct communication (public email exchange in response to a user request) the developers are negotiating and deciding the next steps in the software development, explicitly laying out the dependency structure.

In the co-work episodes for the Gaim project we found only two in which the activity seemed to planned or coordinated though direct communication. Task 34 is one of these cases: *chipx86*, fixing a bug in a patch released by *seanegan*, writes in the SCCS commit message:

> <*seanegan*> ... We should probably remove it from configure.in (the line with src/protocols/icqMakefile in AC_OUTPUT()). Then we can remove it from here

which in the context of the task is *de facto* planning for the next actions needed to complete the task. This slight indication about a possible next step in the software development is the

[1] Developers are identified by the user IDs (shown in italic), as on the system. Missing anaphora is provided in square brackets but quotations are otherwise verbatim.

only direct communication related to the coordination of the activities that we have found in Gaim's episodes.

More often we identified tasks in which coordination is achieved apparently without direct communication. An example of is Gaim episode 5, in which *mallman* in May found and reported a bug (in the "proxy string") and proposed possible changes for addressing the problem. In August *seanegan* uploaded a patch written by *eblanton* that fixed the reported bug. There is no evidence of any direct communication between the developers about this task. Similarly, in Gaim episode 37, we observed apparent collaboration with no direct communication. In that episode *darkrain*, without any previous communication, posted a patch for fixing two bugs. After few days, and again without any intervening discussion, *chipx86* posted a new patch that solves the same problems in a more effective way writing in the SCCS commit message "This looks much better". It seems that other developers agreed without discussion, because after few days *seanegan* thanks him with the standard and brief sentence "*chipx86* fixed it" and then closed the bug report.

In Fire episode 30, the main activity was the development of the file transfer infrastructure, a task mainly realized by *gbooker* with the collaboration of one other developer. During a period of 25 days, the developers change the code 31 times (mainly bug fixing and file transfer implementation) without any trace of direct communication between them being recorded in the public archives. Suitably the description in the SCCS is very simple and begins with this line: "Way too much to describe here...".

We have also found a number of tasks in which some developer uploaded a patch and then in the SCCS thanks some other developer for the work. This is the case of Gaim episode 27 in which *seanegan* writes:

> <*seanegan*> ... Ari and Chip both sent some patches to make things work a bitter better in GTK 2, and Etan rewrote the notify plugin so it's really cool no! Thanks guys!

In this case, as above, we have no evidence of direct communication between the developers for performing those tasks.

Our findings suggest that coordination of work seems to be possible even in the absence of direct communication among the developers. In fact, the analysis of the single developers' behaviour often shows the absence of any form of direct coordination between the actor and the other members of the projects. This lack of evidence is surprising considering the availability and the transparency of FLOSS projects: it should be possible to find the direct, discursive communication (email, thread, forum, etc.) by which developers coordinate their reciprocal activities. Instead, identifying the coordination structure of FLOSS projects seems to be a very difficult and challenging task.

To gain a better deeper understanding of these coordinative dynamics we looked closer at the infrastructure supporting their work. As we analyzed the interaction in the co-work episodes, we found a further interesting result: even in the few cases where developers coordinate through explicit communication, they seem to refer directly to the outcome of their work, the codebase. In the following chat between two Fire developers the most recent software code version embedded in the SCCS (in this case, CVS) serves as a common work artifact for their coordinative communication:

<*reallyjat*> i just noticed that the readme has the wrong month on it…so i'll fix that
<*gbooker*> :)
<*reallyjat*> i made some changes to the about box…did you notice?
<*gbooker*> Just finished downloading. Haven't check out CVS in a while though. This is one long changelog.

The central role of the codebase itself plays out in three ways in this exchange: First, *reallyjat* has checked the SCCS and noticed a (minor) issue, deciding to fix it, acting on his interpretation of the code itself. Second, we notice that *reallyjat* seems to expect that *gbooker* would be watching changes in the SCCS. In other words, developers work by making changes in the code and examining other's changes in the SCCS. Third, as soon as the two developers start discussing, *gbooker* downloads the last software version and examines it so that both developers can refer to the code while discussing.

Indeed, few exchanges occur without direct referral to the code, as illustrated by the following message:

<*jtownsend*> Reading your description above this all sounds like a good idea. However, in looking at the code I'm wondering whether we should be case insensitive on the tags like we were before…

In this example, *jtownsend* seems to agree with a developer's proposal, but as soon as he examines the code he changes his mind and advocates for avoiding a specific technique that had made sense in explicit discussion. Decisions about a development task changes after the interaction between the developer and the code itself. In many situations developers rely on the code itself to communicate accomplished work, as seen in the following example where *Dan Scully* believes that the patch he made to the code is largely self-explanatory:

<*Dan Scully*> I've attached a preliminary patch for RSS Newsfeed support…Most of the patch is self-explanatory, but I'll cover the major ideas here…

A Fire key developer we interviewed informally concurred with these findings. When asked what communication channel dominates coordinating development activities he responded: "SCCS is most important for most tasks." However, the exchanges among the developers do not explicitly address how the codebase can support smooth coordination

without communication in the two FLOSS projects, Fire and Gaim, which is the subject of our theorizing.

On the surface these findings might suggest that FLOSS developers simply rely on "implicit coordination", e.g., by sharing well-developed mental models that allow them to determine what needs to be done even in the absent of explicit communication and coordination (Crowston and Kammerer 1998; Espinosa et al. 2001; Espinosa, Lerch and Kraut 2004). While developers clearly have mental models of the task, it seems unlikely that shared mental models explain all instances of coordination. First, the FLOSS development process is highly complex and ever changing. It seems impossible that developers can keep their mental models up-to-date in the presence of the ever changing dependencies within the code and modifications made by numerous other developers. This is especially true in the sense that the participation of particular developers waxes and wanes over time. Second, FLOSS participants are not all experts but range from newcomers to experienced software engineers. We still need to explain how inexperienced participants develop mental models sufficient to address the coordination needs.

In short, while we do not exclude the existence of implicit coordination, we focus here on the evidence presented above that the developers refer frequently to the codebase. This suggests that the code itself play a central role in coordination that might otherwise be played by implicit or direct communication for coordination.

Theorizing work as coordination

To explain the finding, that coordinated work seems feasible without direct communication, we return to the second half of Figure 1 to theorize that work is coordinated through the outcome of work itself. A recent line of work supports this observation by drawing an analogy to the biological process of stimergy, "a class of mechanisms that mediate animal-animal interactions" (Grasse 1959). As Heylighen writes, "A process is stigmergic if the work ('ergon' in Greek) done by one agent provides a stimulus ('stigma') that entices other agents to continue the job." (Heylighen 2007).

The stigmergic approach suggests that the "shared material" itself can be a coordination mechanism, without recourse to separate coordinative activities. Christensen observed this type of coordination amongst building architectures, arguing that their work is "partly coordinated directly through the material field of work" (Christensen 2008: p 559), "in addition to relying on second order coordinative efforts (at meetings, over the phone, in emails, in schedules, etc.), actors coordinate and integrate their cooperative efforts by acting directly on the physical traces of work previously accomplished by themselves or others."

While studies of stimergy support our general observation that work outcomes can coordinate future activities, the literature does not address our research question: How can we understand coordination as an inseparable part of work and the outcome of work itself?

Stigmergy simply points out that shared materials can serve as coordination devices, but does not explain what specific aspects of this shared work facilitate coordination.

To theorize what elements of work support coordination we turn to the literature on documenting work. As with the document you are in the process of reading, software code is a semiotic product recorded on a perennial substrate which is endowed with specific attributes intended to facilitate specific practices (Zacklad 2006: 217), thus making it a document. While code differs from the present document by serving two audiences, one being a machine, the other software developers, in the following discussion we focus on the properties of software code that allows developers to share their work with colleagues, read, understand and respond to their intentions. Thus, we argue that the software code developed by FLOSS participants can be regarded as documents, along with the other work products of the project members.

Scholars have described how documentation and other accounts of work play a central role in the coordination of work (e.g., Bowker and Star 1994; Bowker and Star 1999; Smith 2005; Suchman 1995; Suchman 1993). In many work contexts, documents serve as both the input and outcome of work, academia being a prime example. These perspectives have long pointed to the double role of documents as both “models of” work and “models for” work. Documents provide an account "of" reality by manipulating text and other symbolic structures so as to parallel them with reality. For example, software engineers may carefully document the code they have constructed to create a report of the work done. By expressing the code in a synoptic form that renders it apprehensible, the software engineers create a model “of reality”. But documents also provide an account that serves as the basis on which people manipulate the world. For example, engineers’ reports are not simply an account of work completed: the reports guide ongoing work by prescribing what is left to be done or enabling collaborators to coordinate their work. The reports are thus an account “for reality” as they provide a blueprint of the software program taking shape. Documents in this way offer a double accountability: when documenting the coding of a software program, engineers mold the account to the reality of the code on their computers and at the same time, mold their ongoing coding to the account.

Now consider the possibility of a construction project without the engineer’s drawings. Could the building itself support such a double coordination process? If so, the building itself would both be the physical representation “of” the work completed and a manifestation “for” the work to be done. For example, a bricklayer showing up at work after a week on the sofa nursing lower back pain might instantly know where to place the next brick simply by looking at the current state of the wall under construction. The mere outcome of the previous week’s labour speaks volumes and thus serves as a coordination device in and of itself. If we accept such an approach we must delve further into what makes some work a model for future activities and thus a coordination device. We will make this argument in four steps: 1) We briefly draw on the basic tenets of a practice theory (i.e., process

perspective). Drawing inspiration from the literature on documents and their double accountability (Brown and Duguid 1994; Buckland 2007; Frohmann 2004; Frohmann 2007; Harper 1998; Lund 2009; Østerlund 2008a; Østerlund and Boland 2009; Østerlund 2008b; Zacklad 2006) we argue that code serves as a model for work by 1) invoking specific work practice genres, and 2) being visible and mobile, and 3) combinable.

Documents and the practice of software development

To work is to engage in practice, an ongoing historical process in which people's doings are caught up and responsive to what others are doing. Taking inspiration from Smith (2005) and Bakhtin (1986), we suggest that a work input is rarely completely original; it is always an answer (i.e., a response) to work that precedes it, and is therefore always conditioned by, and in turn qualifies, the prior activities. What the engineer or bricklayer does when facing somebody's work is responsive and partially determined by what has been going on up till now. Every next act picks up on what has been done and projects it forward into the future. By doing so they are reproducing the structures already established by prior work products and at the same time producing something new by adding to the work product. To put it differently, work is a socio-material configuration under whose guidance new relations and activities are organized. They give meaning and objective form to a socio-material reality both by shaping themselves to it and by shaping it to themselves (Orlikowski and Scott 2008).

Turning to software development, we note that the codebase is part of a practice; a code contribution is like a turn in a conversation. It addresses earlier contributions but at the same time points towards a response. When a coder post a new code to the SCCS the code is always conditioned by, and in turn qualifies, the prior activities to a greater or less degree. It picks up on what has been done and projects it forward into the future. The new code thus works as both a model of work done and more importantly a model for future work. It stands as a question posted in a conversation waiting to be answered.

Three further concepts from document studies stand out as helpful in articulating how work can service as a model for work: genre, visibility and mobility and combinability. We will address these in turn.

Document and code genre

People can recognize a document as a model for some action because they have some background knowledge about the genre of that document, and thus the expectations associated with that type of communication. A genre is defined as typified action invoked in response to a recurrent situation (Yates and Orlikowski 1992). People engage genres to accomplish social actions in particular situations, characterized by a particular purpose, content, form, time, place, and set of participants. The same can be said about work. The engineer and the bricklayer engage in typified actions invoked in response to recurrent

situations. They do so to accomplish something characterized by a particular purpose, material form, place, time and participants. By completing a drawing and leaving it for a colleague to work on, the engineer invoke a specific genre of work. The colleague will be able to pick up and work with the drawing because it invokes a specific genre of work that comes with certain expectations. The first engineer might have created a scaffold of a drawing that simply outlines a structure. In so doing, his work product becomes a model for work associated with specific elements and course of action. It might invoke a sequence of steps or routes to a conclusion. It might invoke certain categories or socio-material arrangements that will have to be used. The bricklayer takes his cues from work completed. If his colleague started out with red bricks he is not likely to randomly switch to yellow ones and thus break a pattern embedded in earlier work. In this way, a piece of completed work serves as a model for future work by hinting at its own genre, i.e., what are the expected outcomes, what materials and forms should be invoked at what places and times and by what types of participants.

Furthermore, documents related to work are often organized into what are called genre systems (Orlikowski and Yates 1994), formalized sequences of documents of particular genres providing more or less standardized methods for analyzing, recognizing what might be done and what gets done as legitimate work activities. For example, the process of publishing a journal paper involves a sequence of documents of specified genres: submission, review, editor's report, decision letter, revision, acceptance letter, final submission, galley proof, copyright release and published paper. Some of these sequences are built into the structure of completed work. As a scaffold, a piece of work might start out as an incomplete frame on which other parts get added in some organized sequence.

For genres and genre systems to enable documents to function as models for work they must be part of the conventions of practice shared among members of particular communities. They are learned as part of membership of such communities where new participants gradually acquire a naturalized familiarity with the socio-material arrangements and prominent genres as they become members.

The code in FLOSS work provides genre expectations and thus serves as "models for" work at two levels. First, the FLOSS development process includes a number of distinct and typified actions involved in response to a recurrent situation and expressed in a set of characteristic documents. As summarized in Figure 2, these documents include source code, bug reports and commit messages. Each of these documents is associated with particular purposes, forms, content, times, places and participants for the related activities. For example, the purpose of the code is to instruct the machine to perform specific applications whereas bug reports are used to send information to developers about observed problems with the program. Code is used nearly exclusively by developers, while bug reports are shared between users and developers. By looking at any of these work outputs experienced FLOSS participants can tell what sort of activities are called for.

Second, the source code itself has a structure in which each component has more-or-less well-defined purposes associated with particular functionalities. There is a genre of source code, as the code collectively has the purpose of providing instructions for the computer, but in a sense, each module of a typical program has its own particular purpose and so its own subgenre. For example, some modules may manage the interface, while others deal with interactions among particular data sources. In a well-structured program, the purpose of each module is clear — the subgenre is recognizable — and so the code is useable. We note further that this communicative purpose is of critical importance for other developers. For example, if a developer wanted to add additional interface functionality, it should be clear which components are appropriate to modify. The general structure of the source code thus serves as a model for what should or could come next. In poorly-structured code, the purpose of particular module may be hard to determine or, in fact, muddled and unclear. This confusion may not directly affect the functionality of the program, but in these cases, the code does not constitute a genre, which means that it does not provide pointers for where new functionality should go. It does not serve as a 'model for' work. Future programmers cannot tell where to add new functionality because the current work outcomes don't make it clear how to add it without negatively interfering with existing functionality. Developers reflecting on FLOSS argue that developing the right program structure, one that communicates well to other developers, is a key to the success of a FLOSS project. A recent book entitled "The Architecture of Open Source" collects narratives of architectural purpose across well-known open source projects; its introduction claims, "If you are a junior developer, and want to learn how your more experienced colleagues think, this book is the place to start" (Brown and Wilson 2011).

FLOSS development further provides expectations about sequential ordering of documents with genres at two levels. First, the broad genres as summarized in Figure 2 are clearly arranged in an organized manner where one activity typically follows another. For instance, a developer would first read a bug report, then change the code and commit with a SCCS commit message. Releases have their own sequence of documents, such as release note, packaged software, binary distributions and so on. In other words, the FLOSS infrastructure supports certain sequences of documents that suggest particular sequences of processes that participants learn as part of membership in the group.

Second, the source code itself develops from a structure much like the outline of an article. Developers build a scaffold which is gradually filled out. Later new functionality can be added to the existing structure and so forth. The SCCS records the full revision history, which include all changes that are made to each file in the system including what files are created or deleted by whom, when. Many changes include short notes that can explain why a change was made (although many changes do not include such explicit notes, apparently expecting the reader to examine the code directly). Such histories not only serve as 'models of' work but can point forward by depicting the generally accepted work process. For a

newcomer, such histories provide a window to how things are done, what activities tend to follow what activities and what is regarded as good and opposed to bad (i.e., reverted) work.

Visibility and mobility

Second, in order for a piece of work to serve as a model for future actions it must be visible and accessible to others. Obvious as it may seem, making work visible is not a straightforward process. As discussed by Suchman (1995), some work may be more visible than other work; some work may cover up previous activities and render them invisible. For example, service work in notoriously hard to make visible: the better the work is done, the less visible it is to those who benefit from it. Understanding what elements of work are accessible and how its visibility may change over time is central to understand how work may and may not service as a model for future activities. Similarly, for work to coordinate activities beyond a physically restricted space, it must become mobile (Latour 1990), meaning that it is conveyed to a context in which others can encounter it. The bricklayer cannot take cues from work done by a colleague if he cannot go to the site or bring it to himself.

Most obviously, the FLOSS development infrastructure supports the mobility of the code by being web-based. Any user can download the source code from the SCCS and have access to others work as a basis on which they can build their own. As a result, software engineers can in many situations use others' work as a model for their own activities because of their ubiquitous access to the server containing the code. Further many projects employ a push mechanism by which other's changes are made available locally in other developers workspaces, usually by email or RSS, rather than waiting for others to seek them out ("commit list"). By being in multiple places, code can co-ordinate work in multiple settings. In addition, some FLOSS projects make use of branches to provide sandboxes where developers can play with code without "breaking the trunk". These branches are public, allowing other participants to follow the progress of a developer's experimentation, and thus make it a potential "model for" their own work.

The infrastructures that have been developed over the past decades further support the mobility of the software development documents by supporting a smooth exchange between the public and private work on the source code depicted in Figure 2. For example, projects can choose between multiple SCCS such as CVS, Subversion or git, each with particular strengths and process affordances. Other packages support other parts of the process, including bug and issue trackers such as Bugzilla, email managers such as Mailman, documentation generators and so on. These packages can be easily accessed on "forges", such as SourceForge, that provide a range of services for hosting FLOSS projects. As a result, a technical infrastructure for a project can be created, or re-created locally, in a matter of minutes, thus supporting the smooth mobility of the code and exchange of work.

Visibility of FLOSS development work is promoted as well through cultural norms about development. A widely acknowledged culture norm in open source is to "check in early, and check in often." In other words, if people do not share their work often they are not making it visible to other participants to build on. Much like a brick wall developers can see the code base grow and take shape, calling for the next piece of code to be added. Contrariwise, a large infrequent commit increases the chance that there will be conflicts and makes it harder for other developers to understand what the change does, again hampering visibility. Indeed, a frequent complaint heard about a contribution to a project is that it is too large for developers to understand.

Combinability

Finally, for work to be a model for future work it must be combinable and improvable in modular increments (Howison 2010; Howison and Crowston 2011; Latour 1990). If a piece of work is done all at once or globally there is nothing left to do, hence Raymond's surprising injunction to "leave low hanging fruit" (Raymond 1998). However, most work tasks and not least those associated with software engineering are layered and complex. It takes time to build and new work contributions can be adjusted and added to existing outcomes.

Combinability in FLOSS development is supported both by the SCCS infrastructure and cultural norms. First, there are strong cultural norms for providing "atomic commits", that is, developers are encouraged to address only one change or topic when committing new code, leading to many smaller commits (Arafat and Riehle 2009). It is easier to combine code with a focused commit than with a commit that does multiple things and touches bits and pieces of dozens of files in the process. Other developers can better see what the new code has been combined and with what consequences. It is likewise easier to revert a focused commit if things should go wrong. Developers are also warned: "Don't Break the Trunk", which means that the main set of files in the SCCS should always compile and run. This practice ensures that any developer who downloads the code will be able to work with it, supporting the individual development described above.

Finally, combinability is supported by the SCCS infrastructure by allowing participants to try out experiments on the code in a branch before committing it to the trunk, i.e., the released system. They can execute and test ideas at any time without interfering with others. This way the developers can run the software with their proposed changes and obtain direct feedback about the combinability and thus success or failure of the current version of the artefact. This allows them to iteratively to enhance their understanding of the task and modify their strategy for managing dependencies between the existing system and what they are trying to accomplish. In this way, developers can interact with the code base as they would engage in a conversation by continuously receiving feedback on their output. This

means that developers can avoid a lot of communication with co-developers, since their active engagement with the artifact provides substantial insights; one has less need to ask another what their intentions were when one can experiment with the codebase.

Discussion

We conclude by discussing our findings and drawing implications for research and practice. We started this paper by suggesting two models of work: an input-process-output view of a process, in which work and coordination are conceptually separate, and a view of work as coordination, in which work clusters around the work product without a clear division between the two. We then presented a study of two FLOSS projects in which we sought evidence of separate coordination mechanisms as suggested by the first perspective. However, this study revealed surprisingly limited explicit coordination of the process. Instead, most work was done by a single individual, and even when multiple individuals contributed, it was mostly without the need for explicit coordination.

Based on the second perspective, we interpret this evidence as showing how the various documents of FLOSS development, and especially the shared source code, serves as a product of the work and a venue for coordination at one and the same time, which is related to stigmergic coordination. Our particular contribution is clarifying the characteristics that work must have in order to be useful in supporting stigmergic coordination. Specifically, we argue that the code itself can serve not only as a model *of* work but more importantly a model *for* future work by 1) invoking typified actions in response to recurrent situations, as part of formalized structures of legitimate work activities (i.e., by having a genre), and by being 2) visible and mobile, and 3) combinable.

In summary then, we understand coordination as a part of work by understanding the work as a socio-material system composed of practices and technologies that support coordination even without direct communication. The social and the material play a role separately and together. The cultural practices of FLOSS projects eliminate the need for some kinds of coordination. For example, an important dependency in most processes is between a task needing to be done and someone to work on it, requiring mechanisms for task assignment. However, in community-based FLOSS projects, developers typically decide for themselves what tasks to undertake (Crowston et al. 2007a), thus avoiding the need for explicit assignment.

Conversely, the discussion of the development process makes it clear that the material features of the SCCS are important in replacing or avoiding the need to coordinate work through explicit communication. For example, since all of the developers work on the same codebase, there is a dependency between their work. Such a situation would normally require explicit mechanisms for resource sharing, but because developers can work on personal copies of the source code files, they can work without the possibility of interfering with other

developers. The SCCS mostly (though not completely) automates the process of integrating these separately-made changes at a point when the work is ready to share.

These approaches to coordination are a combination of social and material. Self-assignment of work is facilitated by the use of technical systems such a bug report or issue trackers that keep track of what work is needed. The technology of the SCCS can only combine work that does not conflict so to facilitate the use of these systems, developers have adopted practice to commit work more frequently in smaller chunks that are easier to integrate technically as well as for other developers to understand. They are also careful to maintain the source code in working order so that others can work with the current version, or to keep experiments in separate branches of the code. Keeping the code well-structured with understandable changes makes it easier for developers to manage the interrelationships within the code.

This understanding of work as coordination has implications for system design, speaking to the notion of modularity as coordination through information hiding (e.g., Baldwin and Clark 2000; Parnas, Clements and Weiss 1981). If one of the functions of the repository is dynamic understanding for dynamic collaboration as requirements change, then enforcing strict information hiding through access controls in the source code repository seems likely to be counter-productive, removing the ability of programmers to track the evolution of each other's work and mutually adjust to it. Information overload is reduced if the repository and its history are available for inspection when the programmer wants, as opposed to only through discursive communications that lose their context over time.

Our analysis of coordination as supported by the socio-material system of FLOSS development also suggests limits on the generalizability of the model and thus the possible adaptation of FLOSS work practices. Specifically, to the extent that the cultural practices or technologies are unique to FLOSS, the general applicability of these mechanisms will be restricted. For example, the technological infrastructure available may be insufficient to combine individual contributions or to allow separated workers to independently access or understand the purpose of different pieces of the product. However, such restrictions are relaxing quickly. For example, Google Docs, which we used in writing this paper, provides a way to share the output and process of writing a document among a distributed group, apparently providing the technological support for a much broader range of knowledge work. Still, managers may not be content to allow team members to pick their own work, that is, self-assignment may not seem to be a viable alternative to explicit coordination. Indeed, in writing this paper, the pressure of the deadline made it problematic to assume that all sections of the paper would eventually get written without some discussion of who would do what. Open source projects often do not face such deadlines and so developers can often afford to wait and see what others do before making their contributions.

These issues point to the need for further research with this framework: under what conditions is it sufficient for coordination of a work process for the documents that comprise the shared output to be generic, visible and combinable and under what conditions is explicit communication needed? If both are feasible, what are the tradeoffs in the quality or efficiency of these approaches? Can all work be coordinated this way with a sufficient infrastructure or are there features of work that always require explicit coordination? For example, returning to our experience with Google Docs, we found that we needed discussion to clarify the task we were trying to accomplish (a kind of coordination problem). But it may simply be that Google Docs is not yet the right tool for such tasks. These problems suggest several fruitful lines for further research.

References

Arafat, O. , and D. Riehle. 2009. "The commit size distribution of open source software." Pp. 1-8 in *42nd Hawaii International Conference on System Sciences (HICSS '09)*. Hawaii, US.

Bakhtin, Mikhail Mikhailovich. 1986. "The Problem of Speech Genres." Pp. 60-102 in *Speech Genres and Other Late Essays: M.M. Bakhtin*, edited by Caryl Emerson and Michael Holquist. Austin: University of Texas Press.

Baldwin, Carliss Y., and Kim B. Clark. 2000. *Design Rules: The Power of Modularity*. Cambridge, MA: MIT Press.

Bowker, G. , and S. L. Star. 1994. "Knowledge and Information in International Information Management: Problems of Classification and Coding." Pp. 187-213 in *Information Acumen: The Understanding and Use of Knowledge in Modern Business*, edited by L. Bud-Frierman. London: Routledge.

Bowker, Geoffry C., and Susan Leigh Star. 1999. *Sorting Things Out: Classification and its consequences*. Cambridge: MIT Press.

Brown, A. , and G. Wilson. 2011. "The Architecture of Open Source Applications."

Brown, J. S., and Paul Duguid. 1994. "Borderline Issues." *Human-Computer Interaction* 9(1):3-36.

Buckland, Michael K. 2007. "Northern Light: Fresh Insights into Enduring Concerns." Pp. 327-34 in *A Document (Re)turn: Contributions from a research field in transition*, edited by Roswitha Skare, Niels Windfeld Lund, and Andreas Varheim. Frankfurt am Main: Peter Lang GmbH.

Cataldo, Marcelo, and James D. Herbsleb. 2008. "Communication networks in geographically distributed software development." Pp. 579--88 in *Proceedings of the Conference on Computer-supported Cooperative Work (CSCW '08)*. San Diego, CA, USA: ACM.

Christensen, Lars. 2008. "The logic of practices of stigmergy: representational artifacts in architectural design." Pp. 559–68 in *CSCW '08: Proceedings of the ACM 2008 conference on Computer supported cooperative work*. New York, NY, USA: ACM.

Conway, M. E. . 1968. "How do committees invent." *Datamation* 14(4):28-31.

Crowston, K., Q. Li, K. Wei, U. Y. Eseryel, and J. Howison. 2007a. "Self-organization of teams for free/libre open source software development." *Information and Software Technology* 49(6):564-75.

Crowston, Kevin, James Howison, Chengetai Masango, and U. Yeliz Eseryel. 2007b. "The role of face-to-face meetings in technology-supported self-organizing distributed teams " *IEEE Transactions on Professional Communications* 50(3).

Crowston, Kevin, and Ericka Kammerer. 1998. "Coordination and collective mind in software requirements development." *IBM Systems Journal* 37(2):227--45.

Espinosa, J. Alberto, Robert E. Kraut, Javier F. Lerch, Sandra A. Slaughter, James D. Herbsleb, and Audris Mockus. 2001. "Shared mental models and coordination in large-scale, distributed software development." Pp. 513–18 in *Twenty-Second International Conference on Information Systems*. New Orleans, LA.

Espinosa, J. Alberto, F. Javier Lerch, and Robert E. Kraut. 2004. "Explicit versus implicit coordination mechanisms and task dependencies: One size does not fit all." Pp. 107--29 in *Team cognition: Understanding the factors that drive process and performance*, edited by Eduardo Salas and Stephen M. Fiore. Washington, DC: APA.

Frohmann, B. 2004. *Deflating information: from Science studies to documentation*. Toronto: University of Toronto Press.

Frohmann, B. 2007. "Multiplicity, Materiality, and Autonomous Agency of Documentation." Pp. 27-39 in *A Document (Re)turn: contributions from a research field in transition*, edited by Roswitha Skare, Niels Windfeld Lund, and Andreas Varheim. Frankfurt am Main: Peter Lang GmbH.

Gerson, E. M. , and S. L. Star. 1986. "Analyzing due process in the workplace." *ACM Transactions on Office Information Systems* 4(3):257–70.

Grasse, P.-P. 1959. "La reconstrution du nid et les coordinations interindividuelles chez Bellicositermes natalensis et Cubitermes sp. La th´eorie de la stigmergie: Essai d'interpr´etation du comportament de termites constructeurs." *Insectes sociaux* 6(1):41-80.

Harper, Richard. 1998. *Inside the IMF: An ethnography of documents, technology, and organizational action*. San Diego: Academic Press.

Harrison, W. 2001. "Editorial: Open source and empirical software engineering." *Empirical Software Engineering* 6(3):193-94.

Heylighen, F. 2007. "Why is open access development so successful? Stigmergic organization and the economics of information." in *Open Source Jahrbuch 2007*, edited by B. Lutterbeck, M. Barwolff, and R. A. Gehring. Berlin: Lehmanns Media.

Howison, J. 2010. "Layered collaboration: A sociotechnical theory of organization in open source software development." in *OCIS Division, Academy of Management Conference*. Montreal, CA.

Howison, J., and K. Crowston. 2011. "Collaboration through superposition."

Howison, James. 2009. "Alone Together: A Socio-Technical Theory of Motivation, Coordination and Collaboration Technologies in Organizing for Free and Open Source Software Development." in *School of Information Studies*. Syracuse, NY: Syracuse University.

Howison, James, Megan Conklin, and Kevin Crowston. 2006. "FLOSSmole: A collaborative repository for FLOSS research data and analyses." *International Journal of Information Technology and Web Engineering* 1(3):17–26.

Ilgen, Daniel R., John R. Hollenbeck, Michael Johnson, and Dustin Jundt. 2005. "Teams in Organizations: From Input-Process-Output Models to IMOI Models." *Annual Review of Psychology* 56(1):517-43.

Latour, B. 1990. "Visualisation and Cognition: Drawing Things Together." in *Representation in Scientific Practice*, edited by M. Lynch and S. Woolgar. Cambridge: MIT Press.

Lund, N. W. 2009. "Document Theory." *Annual Review of Information Science and Technology* 43:399-432.

Madey, Greg. 2009. "SourceForge.net Research Data."

Malone, T. W., and K. Crowston. 1994. "Interdisciplinary Study of Coordination." *ACM Computing Surveys* 26(1):87-119.

Orlikowski, W. J., and S. V. Scott. 2008. "Sociomateriality: Challenging the Separation of Technology, Work and Organization." *The Academy of Management Annals* 2(1):433-74.

Orlikowski, Wanda J., and JoAnne Yates. 1994. "Genre repertoire: The structuring of communicative practices in organizations." *Administrative Science Quarterly* 39(4):541-74.

Østerlund, C. 2008a. "Documents in Place: Demarcating Places for Collaboration in Healthcare Settings." *Computer Supported Cooperative Work (CSCW)* 17(2-3):195-225.

Østerlund, C., and D. Boland. 2009. "Document Cycles: Knowledge Flows in Heterogeneous Healthcare Information System Environments." in *The 42nd Annual Hawaii International Conference on System Science (HICSS-42)*, edited by J. F. Nunamaker, Jr. and R. H. Sprague, Jr. Hawaii, HI: IEEE Computer Society Press.

Østerlund, Carsten. 2008b. "The materiality of communicative practice: The boundaries and objects of an emergency room genre." *Scandinavian Journal of Information Systems* 20(1):7-40.

Parnas, D. L., P. C. Clements, and D. M. Weiss. 1981. "The modular structure of complex systems." *IEEE Transactions On Software Engineering* 11(3):259–66.

Raymond, Eric S. 1998. "The cathedral and the bazaar." *First Monday* 3(3).

Smith, Dorothy E. 2005. *Institutional Ethnography: A sociology for people*. Oxford: AltaMira Press.

Stewart, K. J. , and S. Gosain. 2006. "The impact of ideology on effectiveness in open source software development teams." *MIS Quarterly* 30(2).

Strauss, A. . 1985. "Work and the division of labor." *The Sociological Quarterly* 26(1):1-19.

Suchman, L. A. 1995. "Making Work Visible." *Commuications of the ACM* 38(9):56-65.

Suchman, Lucy. 1993. "Technologies of Accountability: Of lizards and aeroplanes." in *Technology in Working Order*, edited by G. Button. London: Routledge.

Yates, Joanne, and Wanda J. Orlikowski. 1992. "Genres of Organizational Communication: A Structurational Approach to Studying Communication and Media." *The Academy of Management Review* 17(2):299-327.

Zacklad, Manuel. 2006. "Documentarisation Processes in Documents for Action (DofA): The Status of Annotations and Associated Cooperation Technologies." *Computer Supported Cooperative Work* 15(2-3):205-28.