Stigmergic coordination in FLOSS development teams: Integrating explicit and implicit mechanisms

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

[1] *OrgLab, Department of Economics and Law, UniCLAM, Italy*
[2] *Information School, University of Texas at Austin, Texas, United States*
[3] *School of Information Studies, Syracuse University, Syracuse, New York, United States*

Abstract: The vast majority of literature on coordination in team-based projects has drawn on a conceptual separation between explicit (e.g. plans, feedbacks) and implicit coordination mechanisms (e.g. mental maps, shared knowledge). This analytical distinction presents some limitations in explaining how coordination is reached in organizations characterized by distributed teams, scarce face to face meetings and fuzzy and changing lines of authority, as in free/libre open source software (FLOSS) development.

Analyzing empirical illustrations from two FLOSS projects, we highlight the existence of a peculiar model, stigmergic coordination, which includes aspects of both implicit and explicit mechanisms. The work product itself (implicit) and the characteristics under which it is shared (explicit) play an under-appreciated role in helping software developers manage dependencies as they arise. We develop this argument beyond the existing literature by working with an existing coordination framework, considering the role that the codebase itself might play at each step. We also discuss the features and the practices to support stigmergic coordination in distributed teams, as well as recommendations for future research. "Not everything that implicitly exists needs to be rendered explicit" (Sloterdijk, 2009, p. 3).

Keywords: coordination mechanisms, stigmergic coordination, distributed teams, FLOSS teams

1. INTRODUCTION

Coordination in software development teams has been a topic of perennial interest in empirical software engineering research and in management studies. Working under conditions of reciprocal task interdependence and high uncertainty (Faraj & Sproull, 2000; Okhuysen & Bechky, 2009) software development has often been considered a particularly appropriate setting in which to study and test coordination mechanisms and models. The usual starting point of the literature is a conceptual separation between the work itself, on the one hand, and activities undertaken to coordinate it, on the other. This split is clear in

*Corresponding author.
Email address: f.bolici@unicas.it [F. Bolici]

the literature from Conway (1968) through Grinter (1996) and on to the socio-technical congruence work of Cataldo and Herbsleb (2008). This conceptual separation is not limited to the software engineering literature; it also figures in social science, where these two concepts are sometimes named “work” and “articulation work” (Gerson & Star, 1986; Strauss, 1985) and sometimes “tasks” and “coordination mechanisms” (Malone & Crowston, 1991, 1994).

The information processing view of organizations (Galbraith, 1977; Thompson, 1967) traditionally addresses coordination needs in two ways: creating opportunities for communication among interdependent actors or redesigning organizational processes to reduce interdependencies. Other classic organizational theory solutions include defining plans and feedbacks and “administrative coordination” (March & Simon, 1958). These formalized solutions are based on explicit coordination mechanisms that have to be planned and consciously accepted *ex-ante*.

While such explicit coordination mechanisms have been most often identified as important, recent work has focused on implicit coordination mechanisms that allow members to predict and adjust behaviours without overt communication. The literature on “implicit coordination” argues that by sharing well-developed mental models that allow people to determine what needs to be done even in the absence of explicit communication and coordination (Crowston & Kammerer, 1998; Espinosa et al., 2001; Espinosa, Lerch, & Kraut, 2004; Rico, Sánchez-Manzanares, Gil, & Gibson, 2008). In other words, people’s background knowledge allows them to engage in interdependent activities without separate coordination mechanisms or explicit communication. They simply know what to do next based on experience.

This paper makes the argument that the separation of work and coordination work may be disguising an important reality worthy of focused research in software development domain: that developers actively engage in identifying, understanding and resolving emerging dependencies in interaction with the work product itself (i.e., the codebase, that is, the source code of the system under development as it expands through contributions by others). Further, this interpretative process continues even when engaging in explicit discussion, such that the artifacts of work play a crucial and under-explored role in making such discussion effective. Thus, reversing Bendifallah's (1987) idea that the "articulation of work can also become primary work", we focus on the coordinating role of the artifact itself (and the established procedures for sharing and modifying it). Better understanding the capabilities of this mode of coordination is promising because the mechanism we discuss in this paper has low overhead and reduces the need for separate articulation work and therefore the need to maintain congruence between work and coordinating mechanisms.

While artifacts have always been a minor (and under-investigated) feature of collaborative work, their use and usefulness increases with the migration of the collaborative activities towards online environments. In these virtual settings traditional coordination mechanisms (hierarchical direction, mutual adjustment in face to face meeting, etc.) face limitations and the artifact takes on a more important role. In the following sections we will introduce the concept of stigmergic coordination, as an emergent model for integrating explicit and implicit mechanisms where synchronous communication and physical proximity among actors are difficult.

We structure this paper in four parts. First we present an extended literature review on coordination in software development, highlighting contributions examining the active role of the codebase, especially those that make an analogy to stigmergy. We then provide empirical illustrations of this idea, drawing from free/libre and open source software (FLOSS) projects. Next we draw on an existing coordination framework to consider this role systematically. Finally we consider why this mechanism has been under-appreciated and suggest novel strategies for advancing research on this topic.

2. LITERATURE REVIEW

Coordination is a significant concern in empirical studies of software development and has been extensively studied. Research on coordination in software development has taken two basic approaches to the question of interdependencies in software development: elimination or adjustment. Elimination is a strategy that attempts to analyze and plan in advance in order to reduce and ideally eliminate these dependencies, for example by identifying components and specifying their interactions in advance (Baldwin & Clark, 2000; Eppinger, Whitney, Smith, & Gebala, 1994; Parnas, Clements, & Weiss, 1984), often through well-designed and documented component APIs (De Souza, Redmiles, Cheng, Millen, & Patterson, 2004).

Empirical studies, however, have repeatedly identified the inadequacy of such strategies. Curtis and colleagues examined how requirement and design decisions were made, represented, communicated, and how these decisions impacted subsequent development processes for large systems (Curtis, Krasner, & Iscoe, 1988). They found that large projects have extensive communication and coordination needs that are not mitigated by documentation, and emphasize the resulting need for explicit discussion among developers. Consistent with this suggestion, Kraut and Streeter found that the formal and informal communication mechanisms for coordinating work on large-scale, complex software projects were important for sharing information and achieving coordination in software development (Kraut & Streeter, 1995) and, further, that reliance on personal linkages rather than electronic networks contributed to coordination success (Kraut, Steinfield, Plummer, Butler, & Hoag, 1999).

In sum, such studies have found that, regardless of efforts to reduce dependencies, communication between the actors is correlated to the ability to coordinate their work activities (Herbsleb & Moitra, 2001) because such communication helps the actors identify and resolve dependencies as they become apparent through the unfolding of the work. This view has been summarized in Conway's law, which states that the structure of a product mirrors the structure of the organization that creates it (Conway, 1968). Cataldo and Herbsleb (2008), following this second path of argument, have examined the impact on software development productivity of socio-technical congruence between the coordination requirements and mechanisms. They demonstrate that organizations were more successful when there was congruence between the structure of technical dependencies as a source of coordination requirements and the capability to coordinate, as measured by the organization's communication patterns (based on co-location, team membership and explicit discussion). Halverson and colleagues study the role of task visualization as a support to social inferences, thus focusing their analysis on the "articulation of work" (visual representation as support to coordination) rather than on the work itself (Halverson, Ellis, Danis, & Kellogg, 2006).

In recent years, the congruence hypothesis has been tested because increasingly software development is undertaken by teams that are geographically or organizationally distributed. Such virtual work reaches an extreme form in free/libre open source software (FLOSS) development, which is often undertaken outside of any particular organizational context by teams that may rarely if ever meet face-to-face. Conway's law suggests that splitting software development across a distributed team will make it hard to achieve an integrated product (Herbsleb & Grinter, 1999). More effort is required for interaction when participants are distant and unfamiliar with each other's work (Seaman & Basili, 1997). Curtis and colleagues noted that coordination breakdowns were likely to occur at organizational boundaries, but that coordination across these boundaries was often extremely important to the success of the project (Curtis et al., 1988). The additional effort required for distributed work often translates into delays in software release compared to traditional face-to-face teams (Herbsleb & Moitra, 2001; Mockus, Fielding, & Herbsleb, 2000). And yet the success of at least some FLOSS development projects provides a visible counter-example.

The mirroring relationship between product-structure and organization-structure also implies that the complexity and emerging characteristics of collaborative and distributed objects (as FLOSS) can rarely be anticipated *ex-ante* through prescribed and formalized organizational solutions. Thus, building on the well-established concepts (Schelling, 1960), several researchers have investigated how *implicit coordination* mechanisms can manage interdependencies among emergent and unfolding activities (Bechky, 2003; Giustiniano & Bolici, 2012; Puranam, Singh, & Chaudhuri, 2009). In a study of requirements development, Crowston and Kammerer (1998) found that collective mind was an important factor in coordination: in successful groups, developers knew what each would do and could therefore make their work fit the whole better. Faraj and Sproull (2000) found a strong relationship between expertise coordination and team performance, over and above the contribution of team input characteristics, presence of expertise and administrative coordination. Espinosa and colleagues (2001) tested whether implicit mechanisms such as shared mental models are used for coordination for geographically distributed projects. In a later study, they concluded that an effective strategy for coordination success involves finding a mix of explicit and implicit coordination mechanisms appropriate for the task, which may change as the task progresses across time (Espinosa et al., 2004).

2.1 The role of the codebase in coordination

While the role of the codebase has not been at the center of the approaches to coordination reviewed above, it has not been entirely absent. Sørgaard distinguished between two types of coordination, “one is by explicit communication about how the work is to be performed...another is less explicit, mediated by the shared material used in the work process” (Sørgaard, 1988, p. 321). Schmidt and Simone (1996) refer to this “shared material” as the “field of work” paying attention to the shared, visible workspace and its changes, as indirect interaction between actors. They argue that “cooperative work is constituted by the interdependence of multiple actors who, in their individual activities, in changing the state of their individual field of work, also change the state of the field of work of others and who thus interact through changing the state of a common field of work” (Schmidt & Simone, 1996, p. 95). At that time they did not, however, focus on the field of work (visible artifacts and their interpretation) as a primary coordination mechanism, preferring to focus on separate structures of articulation work realized in separate coordination protocols. Within these coordination protocols, however, they do consider the role of artifacts, describing a coordination mechanism as a “coordinative protocol imprinted upon a distinct artifact, which...stipulates and mediates the articulation of cooperative work so as to reduce the complexity of articulation of work” (Schmidt, 2011, p. 117). The artifact in question in their work is an objectification of the processes of articulation work, rather than the work itself.

De Souza and colleagues (2005) originally considered the codebase as “pure inscriptions...that describe the forms and patterns of software system structure and operation”. In later work, however, De Souza and Redmiles (2009) present an ethnographic analysis of the use of API as a coordination mechanism. APIs are, of course, part of the shared codebase, and they find them to play three roles: as contracts they facilitate planning, as boundaries they help to assign individual tasks, and they ground and drive developers' discussions. Other work has highlighted the importance of active interpretation of the codebase, although it has focused on textual comments embedded in the code, rather than the code itself (Ying, Wright, & Abrams, 2005), sometimes improved by, for example, social tagging (Storey, Cheng, Bull, & Rigby, 2006) within codebases.

The role of mediating artifacts has been addressed also in the management literature, through two related concepts: Trading Zones and Boundary Objects. Trading Zones, introduced by Kellog and colleagues (2006), is a "coordination structure that facilitates cross-boundary coordination in fast paced, temporary, and volatile conditions", and thus "[e]ngaging in a trading zone suggests that diverse groups can interact across boundaries by agreeing on the general procedures of exchange even while they may have different local interpretations of the object being exchanged" (Kellogg et al., 2006, p. 39). The researchers identified three practices that enact the trading zone: display (to make the work visible), representation (to express the work in a particular form that can be used by others) and assembly (to refer to, reuse, revise and align the work products of other communities in the construction of their own independent products). The trading zone concept is very useful because it focuses the attention on the workspace and the practices actors can perform in the workspace as organizational coordination mechanisms.

Boundary objects, introduced by Star (1989) and Griesemer (1989) are artifacts that allow the coordination of the perspectives and meaning among different communities. Common artifacts that embed ideas and concepts belonging to different communities that do not usually share practices make it possible to create an inter-group connection without any direct form of communication between the actors of the different communities.

2.2 Stigmergic coordination

A promising line of work has approached the role of the "shared material" in coordination through an analogy to the biological process of stigmergy, "a class of mechanisms that mediate animal-animal interactions" (Grassé, 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, p. 7).

In stigmergic coordination, each insect (ant, bee, etc.) influences the behavior of other insects by indirect communication through changes to their shared environment (e.g., chemical traces or building material for the nest). The action of an actor produces changes in the environment and these changes can provide a stimulus for other actors who respond with another action triggered and shaped by the previous one. An example is termites building nests by dropping their mud on existing piles, rather than starting piles of their own or ants finding food by following the pheromones of previous scavengers (Heylighen, 2007). This process allows the building of complex and interdependent structures without central coordination and direct communication. Stigmergic social insect behavior explains how simple agents, without deliberation, communication or central coordination, can contribute to a common result simply responding to stimuli provided by other individuals and by the environment.

The stigmergic approach suggests that the "shared material" itself can be a coordination mechanism, without recourse to separate articulation work. Christensen (2008) observed this type of coordination amongst building architectures, arguing that their work is partly coordinated directly through the material field of work, "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." (Christensen, 2007, p. 17)

While stigmergy has been used in cognitive science (Susi & Ziemke, 2001) and multi-agent systems simulation in particular (Ricci, Omicini, Viroli, Gardelli, & Oliva, 2007), only recently has it been applied to the coordination of software development, particularly in efforts to explain coordination in open source software development (Dalle & David, 2003; den Besten, Dalle, & Galia, 2008; Heylighen, 2007; Robles, Merelo, & Gonzalez-Barahona, 2005). "This theory suggests that communities would cognitively and collectively react to some of the signs (stigma, in ancient Greek) that characterize the collective output of the community the code base..." (den Besten et al., 2008, p. 318). It has been used as the basis for a simulation (Robles et al., 2005), a comparison between the structure of the code and the division of labor (den Besten et al., 2008) and a high-level analogy for the organization of open source production (Heylighen, 2007).

3. EMPIRICAL ILLUSTRATION

Below we turn to an existing framework for understanding coordination in order to systematically consider the roles that could be played by the codebase, but first we provide empirical illustration of the codebase playing a role in coordination. In order to find empirical illustrations of stigmergic coordination in software development projects, we analyze a virtual setting in which traditional coordination mechanisms face limitations and thus alternative mechanisms seem to be more applicable. We focus in particular on a FLOSS development project. FLOSS projects provide an interesting setting in which to study coordination as they 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 mechanisms of ad hoc coordination, such as face to face meetings or even telephones. Research on FLOSS is enhanced by the excitement with which it is held as a model success for distributed, innovative work (Basili, 2001). FLOSS appears to eschew traditional project coordination mechanisms such as formal planning, system-level design, schedules, and defined development processes (Herbsleb & Grinter, 1999). Characterized by a globally distributed developer force and a rapid and reliable software development process, effective FLOSS development teams somehow profit from the advantages and overcome the challenges of distributed work, making their practices potentially of great interest to mainstream development (Alho & Sulonen, 1998).

Accordingly our empirical illustrations come from two comparable FLOSS projects, Fire and Gaim. Both projects were relatively successful community-based projects developing a multi-protocol IM client. A first example of stigmergic coordination in Fire development project emerges from a chat between two developers:

> <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 words of the first developer, reallyjat, illustrate a first point: he checked the CVS and he noticed a (minor) issue, deciding to fix it, acting on his interpretation of the artifact itself. Secondly, we notice that reallyjat seems to expect that gbooker would be watching changes in the CVS. Thus, it seems that their expected way of working is to make changes in the code and examining other's changes in the CVS. The third consideration is that, 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.

The role of the code itself as an active element in coordinating development activities can also be seen in another example:

> <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 against a specific technique that had made sense in explicit discussion. This example shows that decisions about a development task change after the interaction between the developer and the code itself.

Other developers in their development activities can directly interpret artifacts of work:

> <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…

The importance of the artifact of work in FLOSS development project is also confirmed by the words of a Fire key developer that interviewed about what communication channel was predominant in coordinating development activities who said that "CVS was most important for most tasks."

These examples illustrate, in a manner limited by the evidence issues we consider below, the role that the code itself plays in shaping software development activities. We have illustrated examples in which development tasks are influenced by developers' interaction with the artifact of work itself and the manner in which the code plays a role in coordination among developers.

4. ANALYSIS: COORDINATION FRAMEWORK

In this section we analyze stigmergic coordination using the well-known coordination framework proposed by Malone and Crowston (1994). Doing so, we can analytically introduce a structure to the concept of stigmergic coordination, defining the potential role of the code itself as coordination mechanism. Malone and Crowston define coordination as “managing dependencies between activities.” Considering a programmer facing a shared codebase this has four components:

1) recognition of an activity (e.g. a goal to be accomplished)
2) recognition of dependency
3) understanding of dependency (type, source, importance)
4) management of that dependency: eliminate or satisfy (accommodate)

The codebase can play a role at each of these stages, either alone or together with other mechanisms. While new activities might normally be identified through requirements analysis, the codebase itself can suggest new activities to developers, especially as it changes through the work of others. This can be as obvious as identifying a bug, or an improvement on another’s work, or as subtle as noticing that a newly added library provides services that suggest a new feature to the developer. The first illustration above includes identifying a task (noticing an error in the readme file).

Once an activity has been identified, accomplishing it in a coordinated matter is a matter of knowing how to do it without interfering with:

1) what is there already,
2) what is currently being done elsewhere, or
3) what is to be done in future.

The codebase is particularly well suited to recognizing dependencies between an activity and what is there already. Unlike even the drawings and sketches used by architects (Christensen, 2008), software code is an active artifact: it can be executed and tested at any time. This is important because a developer can run the software and obtain direct feedback about the success or failure of the current version of the artifact with their changes. They can iteratively enhance their understanding of their task and modify their strategy for managing interdependencies between what is there already and what they are trying to accomplish. In this way a developer interacting with a codebase is similar to an explicit discussion, where rapid rounds of feedback can occur. This means a developer can avoid direct discussion with others, since their active engagement with the artifact can provide substantial insight.

Further, if direct discussion is needed (and it often is (Gutwin, Penner, & Schneider, 2004)), developers can engage in highly contextualized discussion supported by their shared artifact. We argue that it is a common experience of programmers, especially in distributed teams, to experiment before seeking explicit discussion, and then to ground that explicit discussion in the context of continuing experimentation. The codebase also seems likely to play an important role when developers are seeking to understand whom to talk with, especially the observation of recent changes in different areas.

It is more challenging to see a role for the codebase alone in understanding what is currently being done elsewhere and what is to be done in the future, since it is likely that there is little evidence about those in the codebase itself. This is the issue pointed out by studies focusing on raising awareness of what others are doing, prior to them checking code in and thus altering the shared artifact (Sarma, Noroozi, & Van Der Hoek, 2003; Tam & Greenberg, 2006). Similarly, while the current codebase may give hints in regard to future plans of others and the team, since these are not yet realized in working code, the ability to iteratively experiment discussed above is not available. Such plans may exist in other artifacts of the team, such as collected user stories, but the mental effort required to transpose those to code is substantial and the literature reviewed in our introduction suggests that interdependencies only emerge in a concrete way as the code is written. This does help to re-conceptualise the useful role of intermediate representations of future plans, such as executable specifications (also known as tests that fail), as practiced in the Ruby community through RSpec.

Once dependencies have been identified, a coordination mechanism is necessary. The codebase can play the role of a coordination mechanism for several kinds of dependencies. We consider in turn goal decomposition, prerequisite constraints and usability.

Goal decomposition (as part of task/subtask decomposition) consists in the decomposition of goals into elementary activities. These are dependencies, as the subtasks need to be selected to achieve the desired goal. A developer that wishes to contribute to a project can understand which tasks should still be addressed (bug resolution, new feature, etc.) simply looking to the code itself and then can provide her/his contribution. We have an empirical illustration of this when user darkrain, without any previous communication or even a bug report, posted a patch for fixing two bugs. After few days, and with out any intervening discussion, user chipx86 posted a new patch that solves the same problems in a presumably more effective way, writing in the SVN “This looks much better.” We found no further discussion of these alternatives until after a few days user seanegan, the lead developer, thanked him with the brief sentence “chipx86 fixed it” and then closed the bug report.

Prerequisite constraint is a specific form of producer/ consumer dependency that emerges when a certain task must be completed before that another activity can begin. When this dependency exists, there should be some for of notification that will allow the beginning of the next activity, as in this illustration, where two actors (gbooker and jtownsend) co-develop a new feature (AIM buddy blocking). gbooker, who seemed to be driving the implementation of this specific feature, committed new code together with an SVN log message that read,

> 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 that, "add[s] notification of block non-buddies pref changing".

Usability is another form of producer/consumer dependency. Describing usability, Malone and Crowston (1991, p. 95) describe the dependency by noting that "whatever is produced should be usable by the activity that receives it." The code itself, with its language and its compiling rules, can be seen as a source of standardization for developers' activities. This use of the code could explain episodes from the FLOSS cases where developers seem to prefer to point directly to the code rather than explain or describe what they have done. For example, in one task from Fire the main activity is 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 (fixing bugs in the file transfer implementation) without any trace of explicit communication between them being recorded in the public archives. Suitably the description in the SVN release notes is very simple and begins with this line:

> Way too much to describe here...

In another example we find a developer posting:

> 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 . . .

In both the examples the code itself plays a role in the communication among developers and provides a reference point to which to compare each developer's activity.

5. DISCUSSION

The possibility and importance of stigmergic coordination through software repositories raises two important implications. The first is a challenge to the current formulation of socio-technical congruence (Cataldo & Herbsleb, 2008). The second explores recommendations flowing from understanding source code repositories as communicative and coordination venues: what features and practices best support stigmergic coordination?

Some researchers (Cataldo & Herbsleb, 2008; Cataldo, Wagstrom, Herbsleb, & Carley, 2006) frame the question of inquiry into socio-technical congruence as one between a set of actors (social frame) and a set of artifact/technical objects (technical frame) and argue that the two sets should fit in order to have better performance. Further it focuses on measuring the social frame through a set of interaction measures including co-location, co-presence on a sub-team and evidence of direct discursive communication.

In contrast the work in this paper suggests that the social and the technical are continuously interacting through an additional venue: the actors are leaving traces of their actions in the code and they are reading and reflecting on the code written by others in order to take coordinated action. In such a situation the code influences the actors' behaviors and actors' behavior simultaneously influences the shape of new code. However, this type of coordination is difficult to analyze through the congruence measures suggested by (Cataldo et al., 2006), since the social and technical frames cannot be separated for analysis. The implication is that analyses seeking to assess social-technical congruence, indeed all analyses of coordination, should also consider the extent of stigmergic coordination—the extent to which developers are able to resolve emergent dependencies by examining the changing codebase.

The second implication focuses on the communicative aspects of the code repository and its role in stigmergic coordination. This conceptualization directs attention to the affordances of the repository: a good artifact for stigmergic coordination ought to be widely available and readily understandable, both as a final product (readable code) and, more novelly, as a dynamic product. Dynamic understandability explains the development and widespread acceptance of FLOSS project development norms such as atomic commits, meaning that logically linked changes can be bundled together but should be separated from logically distinct changes. Indeed entire tool development efforts, such as SVN and git have focused on supporting these practices. The practice of making only atomic commits reduces the size of each commit and ensures that each has a single purpose, making them much more understandable by other developers. In contrast, a large, multipurpose contribution (also known as a code bomb) is much less useful, since it requires considerable work to understand. Where accessible clear code and comments are insufficient, programmatic descriptions of developer intent such as test suites and executable specs can extend the coordinative capacity of repositories. Further, this conceptualization helps to convey how good documentation practices provide resources for developers to identify and resolve emergent dependencies. Conceptualizing such practices as key in coordination ought to help with their design and evaluation: how well do the practices and artifacts serve developers in their active identification, understanding, and interpretation of interdependencies between activities?

Our understanding of the coordinating role of shared code repositories also continues the questioning of the function of modularity as coordination through information hiding (Baldwin & Clark, 2000; Parnas et al., 1984). If one of the functions of the repository is dynamic understanding for adaptive collaboration as requirements change and dependencies become clearer, then enforcing strict information hiding through access controls in the source code repository seems likely to be counter-productive, removing the ability of developers to track the evolution of each other's work and mutually adjust to it; this is similar to the argument regarding the pros and cons of fixed APIs (De Souza & Redmiles, 2009). Information overload is reduced if the repository and its history are available for inspection when the developer wants, as opposed to only through explicit discussions that lose their context over time.

The primary advantage of stigmeric coordination is that it is possible in contexts where synchronous communication and physical proximity among actors are difficult or impossible. This is because stigmergy is enabled by the interaction between an individual and the artifact itself, not multiple individuals. Thus, at any time each individual can access the artifact of work so that they can interpret the changes made by the other developers and eventually leave their own. Thus, stigmergic coordination can be reached at any time and from any place, since it is independent of the presence of the other actors involved in collaborative activities.

While a transparent, changing codebase has intriguing advantages it also has clear limitations as a coordination mechanism. As discussed above it does not seem likely to play a significant role when the dependencies are generated by work yet to be visible in the codebase. This suggests that it will find its primary usefulness in iteratively evolving software. Since this is most clear in the FLOSS context this helps to explain why we and other authors have noticed it there. A second limitation is that the artifacts need to be interpreted by developers. Thus, in some cases, coordination through artifacts can lead to potential misunderstanding if developers do not share a similar "professional vision" (Goodwin, 1994) or are not able to translate the existing codebase into a "shared mental model" of others' intentions.

5.1 Strategies for research

We believe that the stigmergic mechanism of coordination has been under-appreciated in the literature because it is difficult to observe and measure. This is because the work of stigmergy occurs primarily in the heads of developers and in their non-recorded interactions with the code in their private workspaces. Research on coordination in software development has not ventured into this territory. In this section we consider two broad strategies that might be pursued to examine these ideas further and discuss the challenges attendant to each. The first is proof by elimination and the second is proof by positive demonstration.

In principle it ought to be possible to create a convincing demonstration of developers' use of stigmergy by eliminating other known coordination mechanisms, demonstrating an explanatory gap that the perspective presented in this paper can credibly fill. This argument is possible because in a pure case of stigmergic coordination there will be no record left at all, unlike explicit plans, procedures or discussion. Yet the absence of data as a form of proof is particularly hard to rely on, since the possibility reasonably exists that additional, uncollected communication, such as the use of unarchived IRC, direct instant messenger or non-archived emails, or even face to face or telephone communication occurred but has not been collected. For example, we examined the dataset collected by Howison (2008), focusing on tasks in which more than one developer wrote code and found that 14 of 20 had no explicit discussion between the developers in the publicly archived data. Yet, since the participants were not able to provide IM or IRC logs from that period, we could not rule out the possibility that the dependencies were in fact identified and resolved through explicit discussion, rather than active interpretation of shared artifacts alone.

The difficulties of negative evidence suggest instead the second strategy of proof by positive demonstration, that is to seek clear evidence of how developers use the codebase in coordinating. One opportunity is to search the archives of explicit communication within a group for references to uses of code. The empirical illustrations quoted above do, we hope, provide evidence of expectations and practices consistent with the operation of stigmergic coordination. Yet the reality persists that an invisible process only becomes visible in this way when it fails in some way, coming up against its limitations. In this way all evidence that leaks into explicit discussion is likely to be relatively ambiguous.

A second form of proof by positive demonstration, however, may be more productive. It may be possible to ask developers to explain out loud how they were able to manage emergent dependencies in a programming task, highlighting in detail when they identified a dependency and how they explored it and came to choose their course of action. Conducting such an interview could be augmented with click-stream data of their interactions with the codebase (and other tools in their environment), assisting their recall and providing the interviewer a resource for directed questioning. This method, of course, would be qualitative with both the positive and negative implications that come with such an approach. Negatively it would be invasive, time-consuming and not representative, in that one could only conduct detailed interviews with a limited number of participants and tasks. Positively, however, this method might provide the most useful detail on how the process works, when it is useful and when it is not and, importantly, what software engineers might do to support and extend this coordination mechanism.

6. CONCLUSION

This paper has argued that the literature on coordination in distributed development teams would be improved by consideration of a currently under-appreciated line of reasoning inspired by stigmergy. Here the active, interpretative role of the developer, especially as they interact with and observe a dynamic codebase is understood as an important coordination mechanism. Stigmergic coordination emerges between the individual and the collective level: looking at the behavior of a group of developers, they seem to be cooperating in an organized and coordinated way for the production of complex software; but at each individual level, they often seem to be working alone (Howison & Crowston, 2014). We provide illustrations of this concept, reasons why we believe this mechanism has been under-appreciated, and strategies for further research on stigmergy in software work.

REFERENCES

Alho, K., & Sulonen, R. (1998). Supporting virtual software projects on the Web. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 1998.(WET ICE'98) Proceedings., Seventh IEEE International Workshops on* (pp. 10–14). IEEE.

Baldwin, C. Y., & Clark, K. B. (2000). *Design rules: The power of modularity* (Vol. 1).

Basili, V. R. (2001). Editorial: Open source and empirical software engineering. *Empirical Software Engineering*, *6*(3), 193–194.

Bechky, B. A. (2003). Sharing meaning across occupational communities: The transformation of understanding on a production floor. *Organization Science*, *14*(3), 312–330.

Bendifallah, S. (1987). Understanding software maintenance work. *Software Engineering, IEEE Transactions on*, (3), 311–323.

Cataldo, M., & Herbsleb, J. D. (2008). Communication networks in geographically distributed software development. In *Proceedings of the 2008 ACM conference on Computer supported cooperative work* (pp. 579–588). ACM.

Cataldo, M., Wagstrom, P. A., Herbsleb, J. D., & Carley, K. M. (2006). Identification of coordination requirements: implications for the Design of collaboration and awareness tools. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work* (pp. 353–362). ACM.

Christensen, L. R. (2007). Practices of stigmergy in architectural work. In *Proceedings of the 2007 international ACM conference on Supporting group work* (pp. 11–20). ACM.

Christensen, L. R. (2008). The logic of practices of stigmergy: representational artifacts in architectural design. In *Proceedings of the 2008 ACM conference on Computer supported cooperative work* (pp. 559–568). ACM.

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

Crowston, K., & Kammerer, E. E. (1998). Coordination and collective mind in software requirements development. *IBM Systems Journal*, *37*(2), 227–245.

Curtis, B., Krasner, H., & Iscoe, N. (1988). A field study of the software design process for large systems. *Communications of the ACM*, *31*(11), 1268–1287.

Dalle, J.-M., & David, P. A. (2003). The allocation of software development resources in “open source”production mode.

den Besten, M., Dalle, J.-M., & Galia, F. (2008). The allocation of collaborative efforts in open-source software. *Information Economics and Policy*, *20*(4), 316–322.

De Souza, C., Froehlich, J., & Dourish, P. (2005). Seeking the source: software source code as a social and technical artifact. In *Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work* (pp. 197–206). ACM.

De Souza, C., Redmiles, D., Cheng, L.-T., Millen, D., & Patterson, J. (2004). Sometimes you need to see through walls: a field study of application programming interfaces. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work* (pp. 63–71). ACM.

De Souza, C., & Redmiles, D. F. (2009). On the roles of APIs in the coordination of collaborative software development. *Computer Supported Cooperative Work (CSCW)*, *18*(5-6), 445–475.

Eppinger, S. D., Whitney, D. E., Smith, R. P., & Gebala, D. A. (1994). A model-based method for organizing tasks in product development. *Research in Engineering Design*, *6*(1), 1–13.

Espinosa, A., Kraut, R., Lerch, J., Slaughter, S., Herbsleb, J., & Mockus, A. (2001). Shared mental models and coordination in large-scale, distributed software development. *ICIS 2001 Proceedings*, 64.

Espinosa, A., Lerch, F. J., & Kraut, R. E. (2004). Explicit versus implicit coordination mechanisms and task dependencies: One size does not fit all.

Faraj, S., & Sproull, L. (2000). Coordinating expertise in software development teams. *Management Science*, *46*(12), 1554–1568.

Galbraith, J. R. (1977). Organization design: An information processing view. *Organizational Effectiveness Center and School*, *21*, 21–26.

Gerson, E. M., & Star, S. L. (1986). Analyzing due process in the workplace. *ACM Transactions on Information Systems (TOIS)*, *4*(3), 257–270.

Giustiniano, L., & Bolici, F. (2012). Organizational trust in a networked world: Analysis of the interplay between social factors and Information and Communication Technology. *Journal of Information, Communication and Ethics in Society*, *10*(3), 187–202.

Goodwin, C. (1994). Professional vision. *American Anthropologist*, *96*(3), 606–633.

Grassé, P. (1959). La reconstruction du nid et les interactions inter-individuelles chez les bellicositermes natalenis et cubitermes sp. la théorie de la stigmergie: essai d'interprétation des termites constructeurs. *Insectes Sociaux*, *6*, 41–83.

Grinter, R. E. (1996). Supporting articulation work using software configuration management systems. *Computer Supported Cooperative Work (CSCW)*, *5*(4), 447–465.

Gutwin, C., Penner, R., & Schneider, K. (2004). Group awareness in distributed software development. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work* (pp. 72–81). ACM.

Halverson, C. A., Ellis, J. B., Danis, C., & Kellogg, W. A. (2006). Designing task visualizations to support the coordination of work in software development. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work* (pp. 39–48). ACM.

Herbsleb, J. D., & Grinter, R. E. (1999). Architectures, Coordination, and Distance: Conway's Law and Beyond. *IEEE Software*.

Herbsleb, J. D., & Moitra, D. (2001). Global software development. *Software, IEEE*, *18*(2), 16–20.

Heylighen, F. (2007). Open Source Jahrbuch, chapter Why is Open Access

Development so Successful. *Stigmergic Organization and the Economics of Information. Lehrmanns Media*.

Howison, J. (2008). Alone Together: A socio-technical theory of motivation, coordination and collaboration technologies in organizing for free and open source software development.

Howison, J., & Crowston, K. (2014). Collaboration through open superposition: A theory of the open source way. *Mis Quarterly*, *38*(1), 29–50.

Kellogg, K. C., Orlikowski, W. J., & Yates, J. (2006). Life in the trading zone: Structuring coordination across boundaries in postbureaucratic organizations. *Organization Science*, *17*(1), 22–44.

Kraut, R., Steinfield, C., Plummer, A., Butler, B., & Hoag, A. (1999). Coordination and virtualization through electronic networks: empirical evidence from four industries. *Organizational Science*, *10*(6), 722–740.

Kraut, R., & Streeter, L. A. (1995). Coordination in software development. *Communications of the ACM*, *38*(3), 69–81.

Malone, T. W., & Crowston, K. (1991). Toward an interdisciplinary theory of coordination. *MIT Centre for Coordination Science*, (Working Paper 120).

Malone, T. W., & Crowston, K. (1994). The interdisciplinary study of coordination. *ACM Computing Surveys (CSUR)*, *26*(1), 87–119.

March, J. G., & Simon, H. A. (1958). *Organizations*. New York: John Willey & Sons.

Mockus, A., Fielding, R. T., & Herbsleb, J. (2000). A case study of open source software development: the Apache server. In *Proceedings of the 22nd international conference on Software engineering* (pp. 263–272). Acm.

Okhuysen, G. A., & Bechky, B. A. (2009). 10 Coordination in Organizations: An Integrative Perspective. *The Academy of Management Annals*, *3*(1), 463–502.

Parnas, D. L., Clements, P. C., & Weiss, D. M. (1984). The modular structure of complex systems. In *Proceedings of the 7th international conference on Software engineering* (pp. 408–417). IEEE Press.

Puranam, P., Singh, H., & Chaudhuri, S. (2009). Integrating acquired capabilities: When structural integration is (un) necessary. *Organization Science*, *20*(2), 313–328.

Ricci, A., Omicini, A., Viroli, M., Gardelli, L., & Oliva, E. (2007). Cognitive stigmergy: Towards a framework based on agents and artifacts. In *Environments for Multi-Agent Systems III* (pp. 124–140). Springer.

Rico, R., Sánchez-Manzanares, M., Gil, F., & Gibson, C. (2008). Team implicit coordination processes: A team knowledge–based approach. *Academy of Management Review*, *33*(1), 163–184.

Robles, G., Merelo, J. J., & Gonzalez-Barahona, J. M. (2005). Self-organized development in libre software: a model based on the stigmergy concept. *ProSim'05*, 16.

Sarma, A., Noroozi, Z., & Van Der Hoek, A. (2003). Palantír: raising awareness among configuration management workspaces. In *Software Engineering, 2003. Proceedings. 25th International Conference on* (pp. 444–454). IEEE.

Schelling, T. C. (1960). The strategy of conflict. *Cambridge, Mass.*

Schmidt, K. (2011). *Cooperative Work and Coordinative Practices: Contributions to the Conceptual Foundations of Computer-Supported Cooperative Work (CSCW).* Springer Science & Business Media.

Schmidt, K., & Simone, C. (1996). Coordination mechanisms: Towards a conceptual foundation of CSCW systems design. *Computer Supported Cooperative Work (CSCW)*, *5*(2-3), 155–200.

Seaman, C. B., & Basili, V. R. (1997). Communication and organization in software development: an empirical study. *IBM Systems Journal*, *36*(4), 550–563.

Sloterdijk, P. (2009). Spheres theory: Talking to myself about the poetics of space. *Harvard Design Magazine*, *30*, 126–137.

Sørgaard, P. al. (1988). Object oriented programming and computerised shared material. In *ECOOP'88 European Conference on Object-Oriented Programming* (pp. 319–334). Springer.

Star, S. L. (1989). The structure of ill-structured solutions: heterogeneous problem-solving, boundary objects and distributed artificial intelligence. *Distributed Artificial Intelligence*, *2*, 37–54.

Star, S. L., & Griesemer, J. R. (1989). Institutional ecology,translations' and boundary objects: Amateurs and professionals in Berkeley's Museum of Vertebrate Zoology, 1907-39. *Social Studies of Science*, *19*(3), 387–420.

Storey, M.-A., Cheng, L.-T., Bull, I., & Rigby, P. (2006). Shared waypoints and social tagging to support collaboration in software development. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work* (pp. 195–198). ACM.

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

Susi, T., & Ziemke, T. (2001). Social cognition, artefacts, and stigmergy: A comparative analysis of theoretical frameworks for the understanding of artefact-mediated collaborative activity. *Cognitive Systems Research*, *2*(4), 273–290.

Tam, J., & Greenberg, S. (2006). A framework for asynchronous change awareness in collaborative documents and workspaces. *International Journal of Human-Computer Studies*, *64*(7), 583–598.

Thompson, J. D. (1967). *Organizations in Action: Social Science Bases of Administrative Theory*. New York: McGraw-Hill.

Ying, A. T., Wright, J. L., & Abrams, S. (2005). Source code that talks: an exploration of Eclipse task comments and their implication to repository mining. In *ACM SIGSOFT Software Engineering Notes* (Vol. 30, pp. 1–5). ACM.