Coordination without discussion? Socio-technical congruence and Stigmergy in Free and Open Source Software projects

Francesco Bolici
Cassino University
Cassino Italy
francesco.bolici@eco.unicas.it

James Howison
Carnegie Mellon University
Pittsburgh PA USA
jhowison@cs.cmu.edu

Kevin Crowston
Syracuse University
Syracuse NY USA
crowston@syr.edu

Abstract

The idea of congruence between the structure of technical and work dependencies has been demonstrated in commercial software development but has not been explored in detail in free and open source software (FLOSS) development. Previous work identified 103 task episodes, selected from two FLOSS projects, and found that 83 were performed by single actors. We analyze the 20 tasks with multiple actors and find that 14 were performed in the absense of any discursive communication between developers. The qualitative analysis of this evidence shows the paradox that, even if the developers do not seem to communicate explicitly, the software is nonetheless built as result of a collective effort, apparently without central coordination. In answer to this puzzle, this paper turns to the concept of stygmergic coordination as possible explanation. Stigmergy explains how actors can affect the behavior of other members of the community through the traces that their activities leave in shared artifacts. Such collaboration has implications for the socio-technical congruence analysis and the design of collaborative systems.

1. Introduction

The idea of socio-technical congruence as a predictor of coordination success in software development, as introduced by Cataldo et al. [4, 3], is powerful and timely, particularly the matrix comparison formalization. The approach argues that the task of software development generates dependencies that are managed through a set of social and organizational structures that allow developers to coordinate their actions, through discursive (i.e., conversation-like) communication. Congruence means that the available coordination capacity from the communications matches the coordination needs based on the structure of the software.

While the perspective has provided insight into the organization of software development, the focus of research has been on conventional software development (with [27] as a recent exception). In the mean time, free/libre open source software (FLOSS) development has grown as an alternative approach. FLOSS is software developed in an open fashion, meaning that the source code is available for inspection and reuse. Often development is undertaken by a distributed group of volunteers, a mode sometimes referred to as "community" open source. 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 [1]. Traditional organizations have taken note of these successes and have sought ways of leveraging FLOSS methods for their own distributed teams. However, we still do not know the generalizability of these approaches, in particular how socio-technical congruence plays out in such settings.

In order to provide a contribution to this question, we have studied communication in the FLOSS projects. We report on an empirical study, drawing on data from free and open source software projects, which suggests coordination even in the absence of discursive communication. As a result of this finding, we explore a challenge to the usual framing of socio-technical congruence by arguing that coordination can occur more directly through the code itself, particularly as that code is dynamically constructed in a code repository. We provide a theoretical argument for this view drawing on literature from organizational studies (itself inspired by biology). We conclude by examining the limitations and implications of our approach and study.

2. Literature review

Consideration of socio-technical congruence has a long history. The basic notion was first described as Conway's law [5], which states that the structure of a product mirrors

the structure of the organization that creates it. Researchers following this path have examined the impact of alignment between the coordination requirements and mechanisms on software development productivity [4, 3]. It has been argued that organizations will be successful when there is a congruence between the structure of technical dependencies as a source of coordination requirements and the capability to coordinate, as affected by the organization's communication capabilities and structures (with the capability meeting or exceeding the requirement).

One implication of this work is that reducing the dependencies between the different components of the system can facilitate development by reducing the need for coordination [8]. Modularity is one of the most important principles in software engineering thought to reduce the complexity of software development by creating a set of relatively independent modules [17, 18]. However modularity principles can reduce but not completely eliminate the coordination needs among modules and among the activities of a single module. Since eliminating coordination needs is rarely completely possible other work has focused on increasing coordination capabilities and several studies have found that the communication between the actors is correlated to the ability to coordinate their work activities (e.g. [12]). This consideration has been validated and accepted in several contexts, as in the product design literature, where the communication among the engineers is considered a needed requirement for assuring coordination in product design decision making [5]. Based on this review, in this study we sought to examine situations where dependencies would seem to call for discursive communication (assumed to be supporting coordination) in a free and open source software development team, in order to assess the role of socio-technical congruence perspective in this domain.

3. Data and Method

In this section we describe our data collection and analysis approach. Our goal was to describe the degree of communication in software development teams. We describe in turn the setting for our study, the particular projects studied, the data collected and our approach to data analysis.

3.1. Setting

Studies of socio-technical congruence in commercial software development have drawn on evidence regarding the progress of software development contained in software repositories. As development proceeds, evidence of the processes and interactions between tasks and participants is stored in repositories such as email lists, issue trackers, source code management systems and so on. A possible confound to this approach is that communication for coordination might be carried out face-to-face in meetings, offices and corridors, and as such would be unlikely to be recorded in software repositories. Such uncollected communication would be problematic for an inquiry into the nature of coordination, and so has been proxied by factors such as team membership or co-location.

In constrast many open source development teams have the characteristic that their interactions are almost entirely through archived venues (though not all, e.g. [6]). Those that do can be described as community-based, without a shared geograpical center and where collaboration is entirely through computer-mediated communications. Although it is somewhat dependent on individual projects, the bulk of such communications are recorded in archives, both for convenience but also in part in fulfillment of an ideology of openness and transparency. The community-based FLOSS environment, however, also brings with it additional organizational features, such as largely volunteer participants and a lack of a formal, shared organizational existence. We consider the implications of these features in the discussion below.

3.2. The Dataset

For data, we draw on a dataset created by Howison [14] in his study of collaboration patterns in FLOSS development. This dataset forms the basis of the analysis presented in this paper and so its preparation is reported in some detail.

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. Both projects produce similar software, namely multi-protocol IM clients, and, at a high-level at least, share an architecture of core GUI with library based protocol implementation. Based on these similarities and on the ideas of socio-technical congruence we anticipated finding similar patterns of communication in the two groups.

Howison collected as much data on development in these two projects as possible, drawing on publicly available data sharing repositories for research, including FLOSSMole and the Notre Dame SRDA [15, 9, 11]. For each project he collected 1) source code repository data, including log messages; 2) Release data, including Release Notes; 3) Mailing lists and Forums; 4) Issue Tracker discussions. The dataset does not include data on IRC or Instant messaging between participants, nor on email sent between participants privately, and limitations deriving from this are re-

ported below.

The goal of the analysis was to identify the work done to carry out project development tasks, and more specifically, the pattern of communication leading to each. From the available data, Howison reconstructed the task-level processes leading to new features. Unlike software development organizations which implement formal processes, such as those assessed in the SEI's Capability Maturity Model certification [23], there is no reason to expect that participants will record which archived evidence pertains to which tasks undertaken by the teams. Therefore Howison manually inspected the archives and re-organized them to understand the tasks and the contributions made to them by developers. This is time-consuming and laborious but provides stronger validation and understanding of the material than automated heuristics. For reasons of practicality, [14] restricted its inquiry to two inter-release periods, chosen while both projects were highly active and successful and each around two months in length (Fire: 56 days, Gaim 61 days).

Howison defines four concepts we will use in the discussion below: 1) *Task Outcome*: A change to the shared output of the project. 2) *Action*: Work which contributes to a Task Outcome. 3) *Task*: The sequence of Actions contributing to a particular Task Outcome. 4) *Participant*: A distinct individual involved with the project. Therefore it is possible to speak of a Task as a set of Actions leading to a particular Task Outcome.

The method of reconstruction in [14] was as follows. The projects record their understanding of their Tasks in two main locations: the release notes and the README file in the source code repository and these formed the basis for re-organization, providing 60% and 30% respectively, of the 103 Task Outcomes identified; the remainder were identified by reading code repository log messages which did not make it into the README or Release Notes. These Task Outcomes 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). Relevant documents were assigned to tasks to which they seemed to contribute, and documents could be assigned to more than one task. The documents were re-arranged in chronological order and the Actions for which they provided evidence were extracted. Individual documents could provide evidence of more than one Action (if, for example, they thanked a contributor for a patch that was being checked in on their behalf) or conversely multiple documents could be merged into single Actions, as was common for code repository check-ins very nearby in time. During this process the various identifiers used by Participants for different archives (e.g. Sourceforge username vs Real Name/Email address combination) were examined and merged.

The actions so identified were then classified according to their type of contribution towards the task outcome, using a simple scheme: 1) Production work 2) Review Work 4) 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 check-ins. 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.

Through the analysis described above, Howison [14] identified 103 episodes of work leading to new features or changes in the code (Tasks), comprising a total of 786 Actions. For the purposes of this study, the Tasks were classified according to the number of unique Participants who performed Production Work for the Task. Strikingly, of the 103 Tasks, there were 83 in which only a single participant undertook production work, which we called Solo Production Work and only 20 in which more than one did, called Co-Production Work (10 in Fire and 10 in GAIM, examples are available in). In other words, only a fraction of the Tasks are ones in which the software development activities have been accomplished through collaboration between two or more code-writers.

3.3. Data Analysis

We took this dataset as our starting point for our analysis of socio-technical congruence. While there are likely dependencies between tasks, the co-work tasks are those where more than one participant undertook production work. Therefore they are most likely to have the clearest interdependencies and therefore the most pressing coordination requirements. Participants 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. We therefore took these as the critical cases and examined them further for coordinating communication. Specifically all the actions in the co-work tasks were examined for evidence of communication between the programmers contributing to the task outcome.

We expected, following socio-technical congruence approach, to find intense communication exchange among developers that cooperate in a specific single task. We coded the 20 episodes for 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.

4. Findings

We report both quantitative and qualitative results. The quantitative results are presented in Table 1 and illustrative qualitative results are reported below that. Overall we were only able to find discursive communication between the two developers involved in the co-work tasks in 6 of the 20 tasks, in the remaining 14 tasks there was no evidence of discursive communication. In other words, not only is joint work on Tasks relatively uncommon (only 20 of 103 tasks), even in the cases were multiple developers do collaborate, they often do so without any discursive communication.

Table 1. Co-work tasks with direct communication for coordination

Fire	32_3	21	30	36	15	1	3	2	12	4
	✓	✓	-	✓	✓	-	-	-	-	-
Gaim	5	37	43	27	8	2	15	7_2	34	28
	-	-	-	-	✓	-	-	-	✓	-

Analyzing the data qualitatively enables us to provide a selection of tasks that illustrate the presence or absence of direct communication for coordinating developers' activity. A (rare) example of direct communication between developers about the next tasks to perform can be found in Fire task 32_3, where two actors (gbooker and jtownsend) co-develop a new feature (AIM buddy blocking). gbooker, who seems to drive the implementation of this specific feature, commits new code together with an SVN log message that reads "... 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 posts new code that "add[s] notification of block non-buddies pref changing". Analyzing the communication and then the changes in the code, and identifying gbooker as a key player in the development of the whole project, it becomes clear that gbooker's statement ("Once we get the notification ...") was in fact a polite request for work and indicates direct, discursive communication to resolve a dependency in the flow of his work.

A second of the six tasks that include direct communication is Fire task 21, in which gbooker emails the user list in reply to a user's request for information, asking for support from the community and proposing a set of features to be developed. After a day, jtownsend replies pointing out his idea about the merging of two components and proposing a common plan for the subsequent activities: "[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 [gbooker] has in the branch". Through direct communication (public email exchange in response to a user request) the developers are negotiating and deciding their next steps.

In the co-work tasks for the Gaim project we found only two in which the activity was planned or coordinated through direct communication. Task 34 is one of these cases: chipx86, fixing a bug in a patch released by seanegan, writes in the SVN "... 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 and quite indirect planning for the next actions needed to complete the task.

The above examples represent a small fraction of the total work analyzed. More often (14 of 20) we identified tasks in which coordination is achieved apparently without discursive communication. An example of Gaim task 5, which begins in May when mallman finds and reports a bug (in the "proxy string") and he proposes possible changes for addressing the problem. In August, seanegan uploads a patch apparently written by eblanton that fixes the reported bug. There is no evidence in the archives of any discursive communication between the developers about this task.

Similarly, in Gaim task 37, we found an absence of direct communication when darkrain, without any previous communication, posts a patch for fixing two bugs. After few days, and with out any intervening discussion, chipx86 posts a new patch that solves the same problems in a more effective way writing in the SVN "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.

We have also found a number of tasks in which a developer uploads a patch and then in the SVN thanks some other developer for the work. This is the case of Gaim task 27 in which seanegan writes "... 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!". This is interesting because we have no evidence of direct communication between the developers performing those tasks.

Another consideration that emerges from the qualitative data analysis is that developers often seem to prefer to point directly to the code rather than explain or describe what they have done. In Fire task 30 the main activity is the development of a file transfer infrastructure, a task mainly realized by gbooker with the collaboration of one other developer. During a period of 25 days, these developers change the code 31 times (mainly bug fixing and file transfer implementation) without any trace of discursive communication between them being recorded in the public archives. Suitably the description in the SVN is very simple and begins

with this line: “Way too much to describe here...”.

5. Discussion

We investigated the communication pattern in free and open source software development activities in order to analyze its congruence to the evolution of the code. Surprisingly, there is little to compare to assess congruence. The data from these two projects presented in [14] suggests that contrary to expectations, software development work in FLOSS teams is highly individual. We were only able to identify 20 (out of 103) tasks in which developers worked together. However, even in these tasks, when examined in detail, the majority (14 of 20) have no discursive communication at all between the collaborators, despite the face that the work was completed and incorporated into the software.

This section considers four alternative explanations for this finding. The first three are 1) missing data, 2) a lack of dependencies and 3) low quality work. The fourth we develop in more detail, to develop an alternative theory of communication through artifacts and the observation of other's work.

1. *Missing data*. A first possible explanation for our finding is that the communication did happen, but was not captured in our data. Indeed, we have some evidence of “ghost communications” in the dataset, meaning mentions of communications that apparently happened but that were not captured in the data collection. However, these mentions cluster in specific topical areas, such as translations (e.g., a developer thanking a translator for his or her work) rather than being a wide spread phenomenon. There is the possibility that the developers did undertake some discussion via IRC and instant messaging (the topic of the projects), and we do not have access to these hypothetical communications. However the qualitative analysis of the episodes seems to suggest that the use of alternative communication tools should not be extensive, since they are not referred to or even mentioned in other communications.

2. *No Dependencies* A second possible explanation is that there are actually no dependencies between the programmers' work and so no need to communicate (i.e., there is congruence between the needs and the low level of communication). However, since the various pieces of software code being developed by the various developers must work together, we consider this explanation unlikely, although it is possible that the work is trivially combinable. It is also more difficult to support this assumption when more developers collaborate for the production of a specific patch or for solving a bug.

3. *Low quality work* A third possible explanation is that coordination requirements are present but unmet, as suggested by the low level of communication. Such a situation would be expected to lead to a low quality outcome, given the hypothesis about socio-technical congruence. Since we have no evidence regarding quality, such as a count of defects or re-work, this explanation can not be definitively ruled out. On the other hand, both of these pieces of software were relatively widely adopted, at least in the period studied and by the standards of FLOSS projects, which argues against this explanation.

While these three explanations cannot be ruled out by the current study, we suggest that none of them is sufficient to adequately explain the findings presented above. This leaves a puzzle: how are the developers in the FLOSS projects studied accomplishing their coordination when their discursive communication appears to be completely absent? Or put another way, if discursive communication is truly absent, but work is coordinated, what might explain this?

5.1. Implicit Coordination and Stigmergy

As an alternative perspective for explaining our empirical findings we turn to the concepts of implicit coordination and stigmergy. We consider as implicit coordination that reached without discursive communication, shared plans or even previous commitment among the actors. In detail we are looking for a perspective that can explain the absence of direct communication between actors that nonetheless are able to collectively accomplish the production of a complex artifact (the software).

This contrast between the individual and the collective level has been studied for a long time in biology where it has been called the coordination paradox. Looking at the behavior of a group of social insects, they seem to be cooperating in an organized and coordinated way; yet at each individual level, they seem to be working alone as if they do not have any interaction with the other actors of the community.

In order to address and to explain this coordination paradox Grassé, writing in 1959 in [10], defined the concept of stigmergy as “a class of mechanisms that mediated animal-animal interactions”. Heylighen discussed stigmergy in open source software development in very general terms, presenting a definition, “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.” [13]. Stigmergy provides an explanation to the coordination paradox above: each insect (ant, bee, etc.) influences the behaviour of other insects by indirect communication

through the use of artefacts (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 by the previous one, and so on.

Thus the traces left by an individual, or the result of its work, can act as a direct source of stimuli for others. Considering the examples of the ants, this process allows the building of complex structures without central coordination and direct communication. Stigmergic rationalization of social insect behaviour 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.

Stygmergic coordination thus gives an alternative explanation for our empirical findings. For example Gaim task 5, described above, can now be interpreted though this lens as a stygmergic coordination process where the action of mallman (upoading a patch) modifies the environment of the project (allowing new improvement or showing new critical points), acting as a stimuli for the following contribution by eblanton (new patch). The absence of communication is thus motivated by the fact that all the information needed by the developers was already embedded in the traces of their work (patches).

While we do not suggest that human activities are equivalent to those of insects, we do aim to examine to what extent stigmergic principles can contribute to explaining human collaboration and coordination understanding. Considering socio-technical congruence and coordination in FLOSS projects, we can now identify three ways in which stigmergic coordination helps:

1. coordination can be reached through indirect communication among actors;
2. stigmergic interaction is always mediated by artefacts or other traces left by the actors in the environment;
3. the environment and the artefact have an active role, as mediators, nexus of stimuli and ruler of interactions.

As in the biological examples, any artifact uploaded to the project's web site (changes in the lines of code, new or revised document, email, thread post, etc.) is an alteration in the community environment; it is a trace left by other members. When a member of the community discovers something in his virtual environment (a bug, an error, a possible new feature) he may try to address the issue (and post the solution, thus leaving a new trace of his activity) or wait until someone else finds a solution. Thus not only is there the possibility of coordinating action through stigmergy, but of motivating action as well.

Stigmergic principles have already been applied to human actions, especially in the context of cognitive science [26] and of computer science, in the field of multi-agent systems simulation in particular [20]. In our literature review we discovered very few and specific attempts to use the stigmergic principles in the study of software development in general or FLOSS specifically. Heylighen [13] focused on the theoretical analysis of the open access development as alternative approach to both the centralized planning and the "invisible hand" of market models. Still in the field of simulation, Robles et al in [21] tried to simulate the evolution of FLOSS projects using stigmergic principles for their agents. Elliot, writing in [7], connected stigmergy with open source software development but argued that it would only play a role for groupwork involving larger numbers (which he placed at over 25 people).

5.2. Trading Zones, Boundary Objects and the "field of work"

Introducing the concept of stigmergy we have argued that coordination is possible also in the absence of communication, when the coordination between actors is managed by reaction to artifacts (in progress or complete) and traces left by other members of the communities in a shared workspace. Similar coordination structures have recently been addressed in the management literature, particularly focused on the interactions among different communities. The concepts of trading zones, introduced by Kellog et al. in [16] and of community of practices by Wenger in [28] have been developed to address the problem of crossing the boundaries of communities assuring inter-organizational coordination. In separate but similar work in the field of Computer Supported Collaborative Work by Schmidt and Simone in [22] introduce the concept of coordination through observed changes in the "field of work".

A trading zone 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 ..." [16, 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 focus attention on the workspaces and the practices actors perform in those workspaces.

Our empirical evidence shows that the FLOSS projects are characterized by the three characteristics typical of the trading zones: 1. display: the code and the communication between the members of the community are transparent and

accessible by everyone; 2. representation: the programming language used in the project and the norm and rules adopted by the FLOSS community will facilitate the correct understanding and use of the code; 3. assembly: in our empirical evidences, as well as [14], the developers seem to add lines of code on top of lines of code uploaded by others, as in the biological metaphor termites add material to build their nest without direct coordination.

In order to have another perspective on the relationship between the artifacts and the collective activities in a community we can also refer to the concept of the Boundary Objects, Star and colleagues [24, 25] as well as the Community of Practice approach [28]. Boundary objects are those artifacts that allow the coordination of the perspectives of different communities [24]. Inter-group connections created in this way are "reified" since they embody in objects (real or reified) ideas and concepts that can be shared by groups that do not normally use the shared practices. Boundary objects enable conversation by presenting a reified representation of practices without enforcing a unique interpretation of meanings. This is especially necessary when developing software systems, since it is desirable that developers and users, while learning from each other, still maintain their own separate understanding of their own practices.

Star [24] and Wenger [28, p. 55] indicate four qualities for objects to work as boundary bridges. 1) modularity: different perspectives could be combined inside the object in a modular way (each perspective could add an element to the common object). 2) abstraction: only some—and not all—the characteristics of the different experiences are represented in the object. Real differences are needed to make the object interesting for a specific purpose, but a common ground is needed because otherwise the object will not be understood at all by one of the parties. 3) adaptation: boundary objects could be used in a variety of different activities. 4) standardization: the intangible resources in the object are formalized in a standardized structure, so each group knows how to use the object in its own context. For our purposes the concept of boundary object is particularly interesting because it provides a detailed characterization of those artifacts that are considered the trace of the actors' work in the stigmergic coordination view. Considering Task 5 of Gaim, we can see how stigmergic coordination is realized between mallman and eblanton through particular artifacts (the patches in this case) that can be characterized as boundary objects, since their role is to be a nexus around and upon which developers can negotiate and construct new meaning and layers of work.

We have pointed out how the trading zone and boundary object concepts fit with our empirical evidence. However, there is also an important difference that we have to recall. The FLOSS projects that we have examined are settled in a single community characterized by strong identity and shared values and norms. By contrast the boundary object and trading zone work had considered environments with differentiated professional identities and values. Our work, therefore, expands the applicability of these concepts to coordination within a single community, rather than in the interaction between different communities.

Another approach based on indirect coordination was developed in the field of Computer Supported Collaborative Work by Schmidt and Simone in 1996, who refer to coordination through the "field of work" [22]. This concept pays attention to the workspace and its changes as indirect interaction between actors and goes so far as to 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" [22, p 158].

6. Conclusions

This paper has examined the idea of socio-technical congruence for coordination in the FLOSS context. During our initial study we found surprisingly little evidence of discursive communication being undertaken to coordinate actions of developers, even when only viewing tasks with more than a single participant contributing code. As a solution to this puzzle we suggest that developers are coordinating implicitly, through the code repositories. We suggest that this is a form of stigmergy and flesh this out with reference to the literature on trading zones and boundary objects.

The data presented in this paper comes from fully distributed, volunteer FLOSS projects and this context faces limitations in generalizing, both across FLOSS projects and to other forms of software development. Howison, in [14], argues that the high incidence of individual work is driven by a co-evolution of this way of working with the motivations, media and organizational circumstances, such as freedom from deadlines, faced by FLOSS projects. He notes that this is a slow and potentially unreliable manner in which to develop software. Similar observations apply to the work reported here: it may be that the developers undertake stigmergic coordination primarily because they do not have access to suitable discursive communication channels or temporal overlap, and that the lack of such channels not only slows the work down but decreases its quality. Certainly software development in commercial environments does face deadlines and does have the resources to promote richer discursive interactions, including face to face meetings. On the other hand, even in commercial software development, code repositories, and their history of small changes, provide a high-context way and low overhead way to communicate about code that is perhaps even more suit-

able than discursive emails or face to face discussion.

6.1. Practical implications

Stigmergic coordination through software repositories raises two important implications. The first is a challenge to the current formulation of socio-technical congruence in [3, 4]. The second explores recommendations flowing from understanding source code repositories as communicative and coordination venues: what features and practices support good stigmergic coordination?

Cataldo et al, in [3, 4] 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 says that the two sets should fit in order to perform best. Further the approach focuses on measuring the social frame through a set of interaction measures including co-location, co-presense 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 code repository. 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 situations the code influences the actors' behaviors and actors' behaviors simultaneously influence the shape of new code. This understanding, however, is difficult to analyze through the congruence measures suggested by Cataldo et al. since the social and technical frames cannot be separated for matrix style analysis. To the extent that these practices are shared with commercial environments the socio-technical congruence approach may be improved by considering how to account for such non-discursive coordination.

The second implication focuses on the communicative aspects of the code repository and its role in stigmergic coordination. In this way attention is directed to affordances of the repository and the features identified as desirable by Kellog et al, in [16, p. 39], are highlighted.

For example a good trading zone 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 why norms like atomic commits, where logically linked changes are bundled together but separated from logically distinct changes, have emerged as best practice. Indeed entire tool development efforts, such as SVN and git have focused on supporting such practices. Even accessible, clear code and comments are sometimes insufficient programatic descriptions of programmer intent and the coordinative capacity of repositories can be further extended, as through test suites, continuous integration and, possibly, programming by contract.

This understanding of code repositories also speaks to the notion of modularity as coordination through information hiding (e.g. [19, 2]). 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 communication which are more costly and which lose their context over time.

6.2. Future work

The work in this paper is illustrative, rather than confirmatory. It is desirable to continue the examination, moving forward to consider dependencies between solo tasks as well as examining other periods of the projects. It would also be desirable to extend this analysis to other FLOSS projects and other distributed software production projects. It is unclear whether the work can be usefully extended to software development projects where co-location plays a large role as this implies a great deal of communication that is un-archived, and it would therefore be difficult to separate the impact of co-location from stigmergic coordination.

Our evidence come from the absence of archival evidence, and is therefore strongly threatened by the possibility of communication that is systematically excluded from those archives. It would be good, therefore, to support the work with "positive" evidence of stigmergic coordination. Perhaps it is possible to find evidence that developers go directly to the code in order to understand what they ought to do and how they can coordinate it with the work of others. This could be pursued through interviews and possibly think-aloud studies of programmers at work. It might also be possible to work with software developers to instrument their computers to assess how and when they read other's code and the changes in the code repository.

Finally, the features of repositories and their dynamic role in understanding and coordinating could be explored directly through design science experiments, building on the ideas presented for trading zones and boundary objects, and clarifying implications for design. This future work could contribute in enlarging the application of the boundary object and trading zone concepts not only in settings where different communities interact but also in situations in which the interactions take place among actors of the same community.

References

[1] K. Alho and R. Sulonen. Supporting virtual software projects on the web. In *Paper presented at the Workshop on Coordinating Distributed Software Development Projects, 7th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '98)*, 1998.

[2] C. Y. Baldwin and K. Clark. *Design Rules: The Power of Modularity*. Harvard Business School Press, Cambridge, MA, 2001.

[3] M. Cataldo, J. D. Herbsleb, and K. M. Carley. Socio-technical congruence: A framework for assessing the impact of technical and work dependencies on software development productivity. 2009.

[4] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley. Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In *Proceedings of Computer Support Collaborative Work 2006 (CSCW 2006)*, Banff, Alberta, Canada, November 2006.

[5] M. E. Conway. How do committees invent? *Datamation*, 149(4329):28–31, 1968.

[6] K. Crowston, J. Howison, U. Y. Eseryel, and C. Masango. The role of face-to-face meetings in technology-supported self-organizing distributed teams. *IEEE Transactions on Professional Communications*, 50(3):185–203, 2007.

[7] M. Elliott. Stigmergic collaboration: The evolution of group work. *M/C Journal*, 9(2), 2007.

[8] S. Eppinger, D. Whitney, R. Smith, and D. Gebala. A model-based method for organizing tasks in product development. *Research in Engineering Design*, 6(1):1–13, 1994.

[9] Y. Gao, M. V. Antwerp, S. Christley, and G. Madey. A research collaboratory for open source software research. In *the Proceedings of the 29th International Conference on Software Enginering + Workshops (ICSE-ICSE Workshops 2007)*. Minneapolis, MN, 2007.

[10] P. P. Grassé. La reconstrution du nid et les coordinations inter-individuelles chez bellicositermes natalensis et cubitermes sp. la théorie de la stigmergie: Essai d'interprétation du comportament de termites constructeurs. *Insectes Sociaux*, (6):41–81, 1959.

[11] Greg Madey (ed). The sourceforge research data archive (SRDA), 2007+. http://zerlot.cse.nd.edu/.

[12] J. D. Herbsleb, A. Mockus, T. A. Finholt, and R. E. Grinter. An empirical study of global software development: Distance and speed. In *the International Conference on Software Engineering (ICSE 2001)*, pages 81–90, Toronto, Canada, 2001.

[13] F. Heylighen. Why is open acess development so successful? stigmergic organization and the economics of information. In B. Lutterbeck, M. Baerwolff, and R. A. Gehring, editors, *Open Source Jahrbuch 2007*. Lehmanns Media, 2007.

[14] J. Howison. *Alone Together: A socio-technical theory of motivation, coordination and collaboration technologies in organizing for free and open source software development*. PhD thesis, Syracuse University, School of Information Studies, 2009. http://james.howison.name/.

[15] J. Howison, M. Conklin, and K. Crowston. FLOSSmole: A collaborative repository for FLOSS research data and analysis. *International Journal of Information Technology and Web Engineering*, 1(3):17–26, 2006.

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

[17] R. N. Langlois. Modularity in technology and organization. *Journal of Economic Behavior & Organization*, 49(1):19–37, 2002.

[18] A. MacCormack, J. Rusnak, and C. Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015–1030, 2006.

[19] D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, 11(3):259–266, 1981.

[20] A. Ricci, A. Omicini, M. Viroli, L. Gardelli, and E. Oliva. Cognitive stigmergy: Towards a framework based on agents and artifacts. In D. Weyns, H. Parunak, and F. Michel, editors, *E4MAS 2006*, number 4389 in LNAI, pages 124–140. 2007.

[21] G. Robles, J. J. Merelo, and J. M. González-Barahona. Self-organized development in libre software: a model based on the stigmergy concept. In *Proc. 6th International Workshop on Software Process Simulation and Modeling*, 2005.

[22] K. Schmidt and C. Simone. Coordination mechanisms: Towards a conceptual foundation of CSCW systems design. *Computer Supported Cooperative Work. The Journal of Collaborative Computing*, 5(2-3):155–200, 1996.

[23] Software Engineering Institute, CMU. Capability maturity model (software). Industry Standard, 1994.

[24] S. L. Star. The structure of ill-structured solutions: Boundary objects and heterogeneous distributed problem solving. In M. Huhn and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*. Morgan Kaufman, Menlo Park, USA, 1989.

[25] S. L. Star and J. R. Griesemer. Institutional ecology, 'translations' and boundary objects: Amateurs and professionals in berkeley's museum of vertebrate zoology, 1907-39. *Social Studies of Science*, 19(3):397–420, August 1989.

[26] T. Susi and T. Ziemke. Social cognition, artefacts, and stigmergy: A comparative analysis of theoretical fraemworks for the understanding of artefact-mediated collaborative activity. *Journal of Cognitive Systems Research*, 2:273–290, 2001.

[27] P. Wagstrom. *Vertical Communication in Open Software Engineering Communities*. PhD thesis, Carnegie Mellon University, 2009.

[28] E. Wenger. *Communities of Practice. Learning, Meaning, and Identity*. Cambridge University Press, NY, 1998.