The under-appreciated role of stigmergic coordination in software development

Francesco Bolici, James Howison,Kevin Crowston

The under-appreciated role of stigmergic coordination in software development

Abstract—Coordination in software development teams has been a topic of perennial interest in empirical software engineering research. The vast majority of this literature has drawn on a conceptual separation between work and coordination mechanisms, separate from the work itself, which enable groups to achieve coordination. Traditional recommendations and software methods focused on planning: using analysis to predict and manage dependencies. Empirical research has demonstrated the limits of this approach, showing that many important dependencies are emergent and pointing to the persistent importance of explicit discussion to managing these dependencies as they arise. Drawing on work in Computer-Supported Collaborative Work and building from an analogy to collaboration amongst insects (stigmergy), we argue that the work product itself plays an under-appreciated role in helping software developers manage dependencies as they arise. This short paper presents the conceptual argument with empirical illustrations and explains why this mechanism would have significant implications for Software Engineering coordination research. We discuss issues in marshaling clear positive evidence, arguing that these issues are responsible, in part, for the under-consideration of this mechanism in software engineering and outlining research strategies which may overcome these issues.

1 Introduction

Coordination in software development teams has been a topic of perennial interest in empirical software engineering research. The starting point of the literature on this topic 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 the literature from Conway [1] through Grinter [2] and on to the socio-technical congruence work of Cataldo and colleagues [3]. This split is not limited to the software engineering literature; it also figures in the management literature, where these two concepts are sometimes named "work" and "articulation work" [e.g. 4, 5] and sometimes "tasks" and "coordination mechanisms" [6].

The work that addresses this topic has taken two basic approaches to the question of interdependencies in software development: elimination or adjustment. Elimination is a strategy which 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 [7, 8, 9] often through well-designed and documented component APIs [see 10].

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 [11]. They found that large projects have extensive communication and coordination needs which 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 [12] and, further, that reliance on personal linkages rather than electronic networks contributed to coordination success [13].

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 [e.g. 14] because such communication helps the actors identify and resolve dependencies as they become apparent through the unfolding of the work. Cataldo and colleagues, following this second path of argument, have examined the impact on software development productivity of socio-technical congruence between the coordination requirements and mechanisms [3]. They demonstrate that organizations were more successful when there is a 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).

This paper makes the argument that this separation of work and coordination work, while rhetorically useful in combating the planning strategy, may be disguising an important reality worthy of focused research: that developers actively engage in identifying, understanding and resolving emerging dependencies in interaction with the work itself (the codebase and its unfolding in contributions by others) before turning to explicit discussion. Further, even when engaging in explicit discussion this interpretative process continues and the artifacts of work play a crucial and under-explored role in making such discussion effective.

2 Stigmergic coordination

Recent work outside Software Engineering has introduced the metaphor of stigmergic coordination. Drawing from an observational study of building architects, Christensen argues that the work is "partly coordinated directly through the material field of work" [15, p 559]. This language draws on earlier work on Computer Supported Collaborative Work by Schmidt & Simone [16], who refer to the visible location of work as the "field of work". This concept pays attention to the shared, visible 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" [16, p. 158]. At that time Schmidt & Simone did not, however, focus on this mutually changing field of work, these visible artifacts and their interpretation, as a primary coordination mechanism, preferring to focus on separate structures of articulation work realized in separate coordination protocols. This line of thinking was also presented by Robinson [17], quoting Sørgaard, "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" [18].

Christensen [15] draws on the Grasse's work in coordination amongst insects, especially termites, to describe how architects he observed were coordinating their 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.*

Grasse coined the concept of stigmergy as "a class of mechanisms that mediated animal-animal interactions" [19]. 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." [20]. Each insect (ant, bee, etc.) influences the behavior of other insects by indirect communication through the use of 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 by the previous one. Thus the traces left by an individual, or the result of its work, can act as a direct source of stimuli for others, both stimulating them to act and directing their action. Considering the examples of the ants, 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. We are not the first to apply this analogy to the organization of open source developers [21, 22, 20].

Software developers, of course, are not mindless insects, responding in preprogrammed and unavoidable ways to their environments. Indeed Christensen [15] acknowledges this amongst architects, drawing on Goodwin's concept of "professional vision" [23]:

> *We could suggest that practices of stigmergy are based on the actor's professional vision directed at the material field of work (e.g. sketches and drawings) where traces of work previously accomplished are recognized and acted on to a coordinative effect*

To us this suggests a clear and appealing picture of coordination in software development teams drawing attention to the active interpretative process in which the shared workspace created by software repositories allows developers to identify dependencies between their current work and the collected past work of the team.[1]

We do, however, want to go further than Christensen was able to in the domain of architecture. Unlike drawings and sketches, software code is an active artifact: one is able to do things with it, one is able to ask questions of it, run scenarios and test changes and observe their effects. In this way a developer interacting with a codebase is, in some ways relevant to coordination, similar to an explicit discussion. Further a developer is not merely faced with a static codebase, but is able to observe changes to it as other developers choose to make them available. These changes not only indicate what others are working on, (or rather were as discussed by [24]) but provide interpretable clues regarding likely future actions. In other ways, of course, active interpretation of even a changing shared workspace is limited; we will consider limitations below.

2.1 Empirical illustration

In order to find empirical illustrations of stigmergic coordination in software development projects, we analyze those virtual settings in which traditional coordination mechanisms face limitations and thus alternative mechanisms seem to be more applicable. We focus on free/libre and open source software (FLOSS) development projects as 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 [25]. FLOSS appears to eschew traditional project coordination mechanisms such as formal planning, system-level design, schedules, and defined development processes [26]. 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 main-

1. We are grateful to our anonymous reviewers for encouraging more recognition of the active role of the developer

stream development [27]. 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 is a while though. This is one long changelog.

Reallyjat's words 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 others 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 emerges also from 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 suggesting to avoid a specific technique that had made sense in explicit discussion. This example shows an interesting setting in which the decision about a development task changes after the interaction between the developer and the code itself.

Our final empirical illustration show that artifacts of work can be directly interpretated by other developers in their development activities:

<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 says:

CVS was most important for most tasks.

These examples illustrate, in a manner limited by the evidence issues we consider below, the crucial 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.

2.2 Affordances and limitations of stigmergic coordination

As shown by our empirical illustrations the artifacts of work can play a role in coordination among developers. Such coordination has both advantages and limitations.

Since stigmergy is enabled by the interaction between an individual and the artifact itself, stigmergic coordination works also in contexts where synchronous communication and physical proximity among actors are difficult or impossible. Thus, the key point for stigmergic coordination is that 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. This is the case of FLOSS projects, where the CVS and the other artifacts are always accessible by everyone, most of the communications are technology mediated (often in an asynchronous way) and the developers are geographically distributed. Thus, stygmergic 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.

The codebase has another affordance that is important in its coordinative role: the software can be instantiated and tested at any moment. This is a crucial characteristic 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. In this way they can iteratively enhance their understanding of it and modify their strategy for managing interdependencies between what is there already and what they are trying to accomplish. In this way a developer can avoid direct discussion with others, since their active engagement with the artifact can provide substantial insight. If direct discussion is needed, developers can engage in highly contextualized discussion enabled by their shared artifact.

The application of stigmergic coordination is also promising because it has low overheads, and it reduces the need for structures of articulation work and therefore the need to maintain congruence between work and coordinating mechanisms.

While a transparent, changing codebase has intriguing advantages it also has clear limitations as a coordination mechanism. The first 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" [23].

Moreover in the artifacts the developers can find the activities that have already been accomplished, while merely planned actions are not necessarily represented. 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 [e.g. 24, 28]. 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 affordences of inquiry discussed above are 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 reconceptualise the useful role of intermediate representations of future plans, such as executable specifications (also known as tests that fail), as practiced, for example in the Ruby community through rspec.

3 Implications for Software engineering coordination research

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

Cataldo and colleagues [29, 3] 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, copresence on a sub-team and evidence of direct discursive communication.

In contrast the conceptual work in this paper suggests that the two sets 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, Wagstrom, Herbsleb & Carley [29], 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 why norms like atomic commits, where logically linked changes are bundled together but separated from logically distinct changes, have become a norm in the FLOSS community. Indeed entire tool development efforts, such as SVN and git have focused on supporting these practices. Where accessible clear code and comments are insufficient programatic descriptions of developer intent can extend the coordinative capacity of repositories, such as test suites. Further this conceptualization helps to convey how good documentation practices provide resources for developers to identify and resolve emergent dependencies.[2]

This understanding of code repositories also continues the questioning of the function of modularity as coordination through information hiding [e.g. 30, 31]. 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 [10]. 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 which lose their context over time.

4 Strategies for research

We believe that this mechanism of coordination has been under-appreciated in the literature because it is difficult to observe and measure. This is because it 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 which may 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 by eliminating other known coordination mechanisms, demonstrating an explanatory gap which the perspective presented in this paper can credibly fill. This is 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 [32], focusing on tasks

2. We are indebted to an anonymous reviewer for this point

in which more than one developer 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 second strategy is proof by positive demonstration. One obvious place to search is in any archives of explicit communication one does have. 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 non-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.

5 Conclusion

This paper has argued that the literature on coordination in software engineering would be improved by returning to an 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 [32]. We provide illustrations of this concept, reasons why we believe this mechanism has been under-appreciated and strategies for remedying this situation.

References

[1] M. E. Conway, "How do committees invent," *Datamation*, vol. 14, no. 4, pp. 28–31, 1968. [Online]. Available: http://www.melconway.com/research/committees.html

[2] R. E. Grinter, "Supporting articulation work using software configuration management systems," *Computer Supported Cooperative Work*, vol. 5, no. 4, pp. 447–465, 1996.

[3] M. Cataldo and J. D. Herbsleb, "Communication networks in geographically distributed software development," in *Proceedings of the Conference on Computer-supported Cooperative Work (CSCW '08)*. San Diego, CA, USA: ACM, 2008, pp. 579–588.

[4] E. M. Gerson and S. L. Star, "Analyzing due process in the workplace," *ACM Transactions on Office Information Systems*, vol. 4, no. 3, pp. 257–270, 1986.

[5] A. Strauss, "Work and the division of labor," *The Sociological Quarterly*, vol. 26, no. 1, pp. 1–19, 1985.

[6] T. Malone and K. Crowston, "The interdisciplinary theory of coordination," *ACM Computing Surveys*, vol. 26, no. 1, pp. 87–119, 1994.

[7] S. D. Eppinger, D. E. Whitney, R. P. Smith, and D. A. Gebala, "A model-based method for organizing tasks in product development," *Research in Engineering Design*, vol. 6, no. 1, pp. 1–13, 1994.

[8] D. L. Parnas, P. C. Clements, and D. M. Weiss, "The modular structure of complex systems," *IEEE Transactions on Software Engineering*, vol. 11, no. 3, pp. 259–266, 1981.

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

[10] C. R. B. de Souza, D. Redmiles, L.-T. Cheng, D. Millen, and J. Patterson, "Sometimes you need to see through walls—a field study of application programming interfaces," in *Conference on Computer-Supported Cooperative Work*, Chicago, IL,, November 6-10 2004, pp. 63–71.

[11] B. Curtis, H. Krasner, and N. Iscoe, "A field study of the software design process for large systems," *Communications of the ACM*, vol. 31, no. 11, pp. 1268–1287, 1988.

[12] R. E. Kraut and L. A. Streeter, "Coordination in software development," *Communications of the ACM*, vol. 38, no. 3, pp. 69–81, 1995.

[13] R. E. Kraut, C. Steinfield, A. P. Chan, B. Butler, and A. Hoag, "Coordination and virtualization: The role of electronic networks and personal relationships," *Organization Science*, vol. 10, no. 6, pp. 722–740, 1999.

[14] J. D. Herbsleb and D. Moitra, "Global software development," *IEEE Software*, vol. 18, no. 2, pp. 16–20, March/April 2001.

[15] R. Christensen, Lars, "The logic of practices of stigmergy: representational artifacts in architectural design," in *CSCW '08: Proceedings of the ACM 2008 conference on Computer supported cooperative work*.

New York, NY, USA: ACM, 2008, pp. 559–568.

[16] K. Schmidt and C. Simone, "Coordination mechanisms: Towards a conceptual foundation of CSCW systems design," *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, vol. 5, pp. 155–200, 1996.

[17] M. Robinson, "Computer-supported cooperative work: Cases and concepts," in *Proceedings of Groupware '91*, 1991, pp. 59–75.

[18] P. Sørgaard, "Object-oriented programming and computerised shared material," Computer Science Department Aarhus University, Tech. Rep., 1989.

[19] 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*, vol. 6, no. 1, pp. 41–80, 1959.

[20] F. Heylighen, "Why is open access development so successful? Stigmergic organization and the economics of information," in *Open Source Jahrbuch 2007*, B. Lutterbeck, M. Bärwolff, and R. A. Gehring, Eds. Berlin: Lehmanns Media, 2007. [Online]. Available: http://arxiv.org/pdf/cs.CY/0612071

[21] A. Ricci, A. O. M. Viroli, L. Gardelli, and E. Oliva, "Cognitive stigmergy: Towards a framework based on agents and artifacts," in *Environments for Multi-Agent Systems III*, ser. Lecture Notes in Computer Science, vol. 4389. Springer, 2007, p. 124.

[22] G. Robles, J. J. Merelo, and J. M. Gonzalez-Barahona, "Self-organized development in libre software: A model based on the stigmergy concept," in *6th International Workshop on Software Process Simulation and Modeling*, 2005.

[23] C. Goodwin, "Professional vision," *American Anthropologist*, vol. 96, no. 3, pp. 606–633, 1994.

[24] *Palantir: raising awareness among configuration management workspaces*, 2003. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2003.1201222

[25] W. Harrison, "Editorial: Open source and empirical software engineering," *Empirical Software Engineering*, vol. 6, no. 3, pp. 193–194, 2001. [Online]. Available: http://www.springerlink.com/openurl.asp?genre=article&id=doi:10.1023/A:1017379030770

[26] J. D. Herbsleb and R. E. Grinter, "Architectures, coordination, and distance: Conway's law and beyond," *IEEE Software*, vol. 16, no. 5, pp. 63–70, 1999.

[27] K. Alho and R. Sulonen, "Supporting virtual software projects on the Web," in *Workshop on Coordinating Distributed Software Development Projects, 7th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '98)*, Palo Alto, CA, USA, 1998.

[28] J. Tam and S. Greenberg, "A framework for asynchronous change awareness in collaborative documents and workspaces," *International Journal of Human-Computer Studies*, vol. 64, no. 7, pp. 583 – 598, 2006, theoretical and empirical advances in groupware research. [Online]. Available: http://www.sciencedirect.com/science/article/B6WGR-4JKRWH6-1/2/2a3a8faa317d56a8e926b4b6e0625df8

[29] 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 the 20th anniversary conference on Computer-supported cooperative work (CSCW '06)*. Banff, Alberta, Canada: ACM, 2006, pp. 353–362.

[30] D. L. Parnas, P. C. Clements, and D. M. Weiss, "The modular structure of complex systems," *IEEE Transactions on Software Engineering*, vol. 11, no. 3, pp. 259–266, 1985.

[31] C. Y. Baldwin and K. B. Clark, *Design Rules: The Power of Modularity*. Cambridge, MA: MIT Press, 2000, vol. 1.

[32] J. Howison, "Alone together: A socio-technical theory of motivation, coordination and collaboration technologies in organizing for free and open source software development," Ph.D. dissertation, Syracuse University, School of Information Studies, 2009.