'Computing' Requirements in Open Source Software Projects

Completed Research Paper

Xuan Xiao
Harbin Institute of Technology
Harbin, Heilongjiang, China
xiaoxuanhit@gmail.com

Aron Lindberg
Case Western Reserve University
Cleveland, Ohio, USA
aron.lindberg@case.edu

Sean Hansen
Rochester Institute of Technology
Rochester, New York, USA
shansen@saunders.rit.edu

Kalle Lyytinen
Case Western Reserve University
Cleveland, Ohio, USA
kjl13@case.edu

Tienan Wang
Harbin Institute of Technology
Harbin, Heilongjiang, China
wtn@hit.edu.cn

Abstract

Due to high dissimilarity with traditional software development, Requirements Engineering (RE) in Open Source Software (OSS) remains poorly understood, despite the visible success of many OSS projects. In this study, we approach OSS RE as a sociotechnical and distributed cognitive activity where multiple actors deploy heterogeneous artifacts to 'compute' requirements as to reach a collectively-held understanding of what the software is going to do. We conduct a case study of a popular OSS project, Rubinius (a Ruby programming language runtime environment). Specifically, we investigate the ways in which this project exhibits distribution of cognitive efforts along social, structural, and temporal dimensions and how its requirements computation takes place accordingly. In particular, we seek to generalize to a theoretical framework that explains how three temporally-ordered processes of distributed cognition in OSS projects, denoted excavation, instantiation, and testing-in-the-wild, tie together to form a powerful distributed computational structure to manage requirements.

Keywords: Open source software, Requirements engineering, Distributed cognition, Requirements computation

Introduction

> "*The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.*" – Fred Brooks, "No Silver Bullet" (1987)

The determination and management of system requirements remains one of the essential challenges of software development (Cheng and Atlee 2009). In software development, requirements are frequently uncertain (Damian and Zowghi 2003; Mathiassen et al. 2007), inconsistent in that they reflect the interests of disparate stakeholders (Crowston and Kammerer 1998), and volatile because they emerge in turbulent environments (Damian et al. 2013; Mathiassen et al. 2007). As a result, pitfalls in requirements engineering (RE) form a significant source of project stress and ultimately failure (Aurum and Wohlin 2005; Crowston and Kammerer 1998; Hickey and Davis 2003; van Lamsweerde 2000). Furthermore, while the research community has developed a rich and multifaceted literature on traditional RE methods and principles, relatively little research has been conducted on requirements-oriented work in the context of open source software (OSS; Vlas and Vlas 2011).

OSS is developed and released in a public and collaborative manner under open licensing agreements. This enables source code to be freely used and disseminated (Crowston et al. 2007). OSS is produced by a large number of highly-distributed and autonomous developers, who coordinate their work using a multitude of development and communication platforms. On OSS projects, individual participants may come and go at any time and decide where and when to devote their effort and energy. Usually, OSS projects are organized as social movements consisting of a small group of core developers who control the code base and create most of the new functionality, while a larger set of peripheral contributors address narrower problems, contributes minor patches, provides training support, or simply provide suggestions (Mockus et al. 2002). This form of distributed development suggests that requirements knowledge and related skills are diverse and widely disseminated across the social arena.

In contrast to traditional structured development, OSS projects generally eschew formalisms, such as official schedules and standardized documentation (Scacchi 2002). OSS participants identify and settle requirements "on the fly" while being embedded in "OSS webs" (Scacchi 2002) – i.e. families of channels and artifacts – through which requirements knowledge is organized, accessed, and transformed. In general, OSS requirements knowledge is not centralized to 'master documents'; rather, it is distributed across heterogeneous artifacts and actors (Mockus et al. 2002; Scacchi 2002, 2009). Such distribution is often highly volatile, with high rates of developer turnover (Robles and Gonzalez-barahona 2006) driven by varying motivations (Von Krogh et al. 2012; Markus et al. 2000; Shah 2006) and self-assignment (Crowston et al. 2007). Accordingly, the traditional RE assumption that requirements are determined by a small and stable team fails to hold in these environments (Crowston and Kammerer 1998).

Some research has been conducted to distill how OSS RE differs from traditional RE due to the voluntary nature of participation (Crowston et al. 2007; Shah 2006) and widespread use of "informalisms" – informal web-based descriptions of what's happening in an OSS project (Scacchi 2002, 2009). These studies, however, do not provide a cohesive picture of the concurrent interaction of actors and artifacts oriented toward requirements determination in OSS environments. Moreover, they do not offer plausible reasons for why some OSS configurations appear to be surprisingly stable and resilient in determining requirements while others are not. One recent study (Thummadi et al. 2011) has focused on how RE is distributed socially and structurally simultaneously, but it focuses on the impact of distributions on the quality of requirements. Overall, there is a gap in our understanding of how OSS projects successfully determine requirements, given the enormous distribution and volatility encountered. Accordingly, the present study seeks to explore distributed RE processes in OSS environments.

In light of the highly distributed nature of OSS RE, we draw upon the theory of distributed cognition (Hutchins and Klausen 1996; Hutchins 1995) as a sense-making device. The theory advocates for an understanding of cognition that extends beyond the boundary of an individual's mental processes to include distribution across multiple entities – human as well as artificial. This perspective is relevant to our inquiry into OSS RE due to the observed heterogeneity of interrelations between actors and artifacts

in the cognitive effort associated with RE (Hansen et al. 2012). In this research, we apply the principles of distributed cognition to the experiences of an active OSS project.

The theory of distributed cognition maintains "the principle metaphor of cognitive science" (Hutchins 1995, p. 49) – i.e., cognition as computation (Johnson-Laird 1989; Simon and Kaplan 1989). Accordingly, in this study, we focus on how an OSS project achieves *requirements computation* – a collective effort of multiple social actors and heterogeneous artifacts to achieve a feasible requirements closure (Hansen et al. 2012). Through an exploratory case study, we analyze the ways in which an OSS community achieves collaborative 'computation' of a collectively-held requirements set around an evolving design vision. Specifically, we address the following questions: 1) In which ways is OSS RE distributed across actors and artifacts? 2) How do the dynamics between actors and artifacts unfold as they collaboratively 'compute' requirements? 3) What are the specific heuristics and strategies whereby OSS requirements are computed?

Building upon our case study findings, we contend that OSS requirements are embedded within a complex cognitive system of actors and artifacts through which requirements knowledge is generated, stored and disseminated. We also note that actor-artifact configurations are vital in understanding the patterns through which requirements are computed to fulfill RE tasks. These insights provide researchers with a conceptual foundation to understand varying OSS RE practices, and provide guidance to practitioners on how to manage OSS RE efforts.

The remainder of the paper is organized as follows. In the next section, we provide a brief introduction of RE in OSS. This is followed by review of distributed cognition which sets the conceptual foundation for our case study. We then present an overview of the Rubinius case and the detailed findings from our case analysis. We conclude with implications for research and practice, as well as future research directions.

Theoretical Background

To establish a proper context for this study, a brief review of two research domains is warranted. First, we review the extant research on the management of RE knowledge within OSS environments. We then introduce the theory of distributed cognition that grounds our present research.

OSS and RE

The research literature acknowledges significant differences between the context in which traditional RE practices are conducted and that of OSS (Noll and Liu 2010; Scacchi 2002, 2009). These differences are founded on voluntary participation (Crowston et al. 2007; Shah 2006) and use of 'informalisms' (Scacchi 2002, 2009). Given the dearth of formal controls in OSS the question of RE activities and requirements evolution in OSS is ambiguous, but important. To date, relatively little research has been conducted on requirements management in OSS environments (Vlas and Vlas 2011). Specifically, the ways in which distributed actors and artifacts work in concert to process requirements have only been examined in passing. However, based on the extant research, a few key patterns can be discerned.

In his seminal work on the OSS phenomenon, Raymond (1999) employs the metaphor of the bazaar – an open environment of exchange – to characterize the OSS environment. The metaphor portrays an image of a large number of voluntary participants (developers) and customers (users) with distinctive expertise and viewpoints plunging into developing OSS in a seemingly random or haphazard manner. Nevertheless, order does emerge in such environments. For example, we know that distributed work in OSS projects tends to take on a structure which can be illustrated through another metaphor – that of an onion (Crowston and Howison 2005). As an OSS project evolves, a core of developers emerges based on the quality and quantity of participants' contributions (Ye and Kishida 2003) and their technical skills (Scacchi 2005). As one moves further away from the core, the levels of consistent participation and skills tend to decrease (Crowston and Howison 2005; Ye and Kishida 2003). Consequently, the core takes control of the project (Mockus et al. 2002; Valverde and Solé 2007) including its requirements facets, and acts as a gatekeeper of project participation and contribution (Asundi and Jayant 2007).

Importantly, requirements knowledge is not limited to the ideas of the core team, but is constantly exchanged among peripheral developers. As Raymond (1999) observes, "Every good work of software starts by scratching a developer's personal itch" (p. 25). Thus, as "itches" are not constrained to the core team, the requirements knowledge accessible by the project is largely unbounded. While this stream of

literature captures the increased diversity of requirements knowledge sources as more distributed social actors participate in the OSS project, it focuses largely on OSS governance. There is little, if any, consideration of the role of artifacts in this process. In particular, it does not point out how the increased distribution of social actors together with artifacts engenders RE processes, which enable effective channeling of the sourced requirements knowledge to implemented artifacts.

A second stream of research focuses on the use of heterogeneous mechanisms and artifacts through which requirements knowledge is disseminated. Scacchi (2002, 2009) notes that OSS communities do not generate explicit functional statements represented in formal models. Rather, requirements get articulated through a wide range of "informalisms," such as threaded discussion forums, web pages, e-mail communications, and external publications (Scacchi 2002). Recent studies (Ernst and Murphy 2012; Noll and Liu 2010) confirm that OSS RE is less formal but at the same time highly dependent upon online documentation and related communicational tools. They further articulate that requirements are asserted based on developers' experience and domain knowledge (Noll and Liu 2010). The implementation of requirements is typically improvisational and largely consists of post-hoc rationalization of experience (Ernst and Murphy 2012). Though this research stream provides more detailed explanations of how distributed artifacts support RE, it generally reflects a discursive consideration of artifacts and thereby fails to recognize a dynamic flow of requirements computation through the interaction of actors and artifacts. Although Noll and Liu (2010) do consider both social actors and artifacts, none of the studies investigate the dependencies between social actors and artifacts that would enable OSS RE processes.

Finally, work by Thummadi et al. (2011) considers how requirements are distributed across social actors and heterogeneous artifacts. However, they primarily examine the impact of distribution on requirements quality, rather than the distribution *per se*. In particular, they provide limited insight into how requirements knowledge is propagated through varying forms of distribution. Thus, we seek a more holistic framework, employing a sociotechnical perspective (Emery and Trist 1960; Mumford 1985) to identify the relevant modes of distribution and the mechanisms by which they interact to enable OSS RE.

Distributed Cognition

The distributed nature of OSS calls for an approach that recognizes the presence of multiple and varying actors as well as the use of diverse artifacts throughout the course of design and development. Therefore, we suggest an approach that acknowledges RE as a form of sociotechnical computation, which not only identifies the distinct roles of actors and artifacts but also the processes through which they interact in OSS RE to compute the requirements. In this regard, we draw on the theory of distributed cognition (Hutchins 1995) to theoretically ground the mechanisms of requirements computation.

Distributed cognition is a thread of cognitive science theory which has been used extensively to explain how solutions to cognitive tasks emerge from a given cognitive system comprised of distributed actors and artifacts engaged in collaborative work (Hutchins and Klausen 1996; Hutchins 1995). Accordingly, the theory of distributed cognition can offer a fruitful lens for analyzing OSS RE, as it involves a collective effort of multiple social actors and heterogeneous artifacts to achieve a feasible requirements closure (i.e., requirements computation).

Distributed cognition theory argues that cognitive processes are not limited to the mind of an individual (Rogers and Ellis 1994; Zhang and Norman 1994); instead, the unit of cognitive analysis should be expanded to the level of a system – all entities (i.e., human and artificial) involved in a specific task. Cognition can be perceived as a socially and structurally distributed phenomenon with cognitive workloads shared among members of a team and artifacts that the team uses (Hutchins and Klausen 1996; Hutchins 1995). The involvement of artifacts fundamentally alters the nature of cognitive tasks and processes (Norman 1993). In consequence, the principle of "cognition as computation" in cognitive science (Hutchins 1995; Perry 1999) is formulated in a broad sense as "the propagation of representational states across representational media" (Hutchins 1995, p. 118). In this context, a representational state is "a configuration of the elements of a medium that can be interpreted as a representation of something" (Hutchins 1995, p. 117). The configuration may be composed of both internal representations, such as those in an individual's mind, and external representations such as those embodied in artifacts and the physical environment (Perry 1999; Rogers and Ellis 1994). As the representational states change over time, information can be computed and propagated accordingly. In this regard, distributed cognition theory implies at least three modes of cognitive distribution (Hansen et

al. 2012; Hutchins 2001; Thummadi et al. 2011), which are designated as *social distribution* (i.e., the distribution of cognition among actors), *structural distribution* (i.e., the distribution of cognition across artifacts), and *temporal distribution* (i.e., the distribution of cognitive process and tasks over time).

First, *social distribution* refers to distribution of cognitive workload across members of a social group. It reflects the cognitive processing that results from the interaction of multiple actors with diverse skills and expertise. Social distribution has obvious application to the study of OSS RE, as requirements efforts are carried out through diverse volunteers playing specific roles (Crowston and Howison 2005; Ye and Kishida 2003) with significant diversity of knowledge (Noll and Liu 2010; Scacchi 2005). Second, *structural distribution* refers to distribution of cognitive workload achieved through the use of external artifacts. It reflects the information processing roles of artifacts that are mobilized and integrated as extensional elements of the broader collaborative cognitive system. For example, the multiple forms of 'informalisms' and rigorous testing infrastructures, which serve as repositories and archives, help shoulder cognitive load in OSS projects (Scacchi 2002, 2009). Third, *temporal distribution* refers to distribution of cognitive workload with regard to time. This suggests a sense of path dependency: previous cognitive efforts impact ensuing cognition. Temporal distribution reflects how cognitive effort is changed over time through the configuration and reconfiguration of elements in the social and structural dimensions. Temporal distribution can be observed in the OSS RE in reusing ideas and code derived from earlier projects (Raymond 1999). While we can isolate social, structural, and temporal distribution in analytical terms, they are in reality inseparable and interdependent.

We have argued above that OSS RE differs from traditional RE in its structural and social forms; yet it shares the same essential cognitive tasks of discovery, specification, and validation of traditional RE (Hansen et al. 2009). These tasks express how a cohesive requirement set, which explains in a clear and collectively accepted form what the software is going to deliver, is achieved. Discovery concerns the recognition and elicitation of needs and constraints in the form of initial requirements from all relevant stakeholders (Mathiassen et al. 2007). As stakeholders' needs are identified through discovery, specification describes clear and precise articulation of those needs so that the designer will know what the software will do and thus extends discovered needs into functional and technical implications (Hansen et al. 2009). Validation addresses the question of whether or not the established requirements are correct, complete, and consistent and thereby agreed upon by all (Bahill and Henderson 2005). It is important to note that these three tasks are generally not strictly sequential; rather, they often evolve in parallel and through iterative loops as requirements are continuously refined through computation.

Research Design

In presenting our research design, we first briefly describe the case study method that we employ in this research. We then provide an introduction to the Rubinius project at the center of our case analysis.

Method

Given that there is not a solid research base on distributed RE, our study of OSS RE was motivated by the need to identify critical actors and artifacts involved in OSS RE practices, and how their dynamic interaction enables successful computation of requirements. Accordingly, we examine a representative case to achieve a holistic exploration of the phenomenon (Yin 2009). The criteria for selecting a representative case were: 1) the project is of reasonable size to be representative of a larger population of OSS projects; 2) the project has a relatively large developer community, so as to illustrate collective work processes rather than the workflow of an individual; and 3) the project has had at least one official release, thus indicating its viability. We selected the Rubinius project, a relatively popular OSS project sponsored by Engine Yard, Inc., as our representative case. The sponsorship of the project involves paying several developers to work fulltime on the Rubinius project. Rubinius filled all our case selection criteria: it has 2.6 million lines of code, more than 100 contributors and had first successful release at late 2010.

The development community of Rubinius forms the focal unit of analysis. Data was collected over a six-month period from September 2012 to February 2013 and drew on multiple sources of evidence to enable data triangulation (Yin 2009). An interview protocol was developed prior to data collection (Yin 2009) to support a comprehensive and systematic approach for collecting data. An initial interview was conducted with the founder of Rubinius who helped to facilitate access to subsequent respondents. A total of 17

interviews were conducted with 13 members of the Rubinius community, who were identified through a snowball sampling technique. In addition, we conducted an interview with one Senior Vice President (SVP) at Engine Yard who took the lead in Rubinius sponsorship. While the Engine Yard SVP was not an active member of the Rubinius development community, we sought his participation as he represented a key functional stakeholder in the Rubinius project with regard to RE distribution. Since Engine Yard paid several committers to work on the project, we perceive that Engine Yard, as Rubinius' sponsor, plays a significant role in deciding what features Rubinius was going to develop. The interview protocol was e-mailed to all respondents in advance so that the respondents could have a sense of what we were attempting to investigate. All interviews were between 60 minutes and 90 minutes in duration. The interviews were audio recorded with the permission of the respondents. The recordings were subsequently transcribed and follow-up emails were occasionally used to clarify points discussed during the interviews. In addition to the interviews, we collected data from project repositories hosted on GitHub, the project's host platform, and conversations archived in the project's mailing list and Internet Relay Chat (IRC) channel. In part, observations across these secondary source elements helped us to validate, refine, and triangulate the analysis and interpretations that we generated through our interviews. A summary of data collection activities and data types is provided in Table 1.

Table 1. Summary of Data Collection

Data sources		Descriptions
Interviews		Total of 17 interviews conducted, composed of 16 interviews with 13 Rubinius committers within Rubinius community and 1 interview with a Senior VP at Engine Yard Company
Complementary Sources	Project's repository	All publicly available project descriptions, charters, bylaws, issues, pull requests, etc.
	Project's website	All publicly available roadmaps, blogs, documentations, etc.
	Mailing lists	Over 100 emails read, covering the period from late 2010 to late 2012 (data collected in late 2012)
	IRC channel	Over 1000 messages read, covering the period from late 2010 to late 2012 (data collected in late 2012)

In our data analysis, we used the qualitative coding techniques espoused in grounded theory methodology, such as open, axial, and selective coding (Strauss and Corbin 1990). At the same time our analysis was driven and supported by the theoretical framework of distributed cognition, which served as a sensitizing device (Corbin and Strauss 2008). We used Atlas.ti qualitative analysis software (www.atlasti.com) to code the interview transcripts and the secondary data files.

The initial round of coding focused on the identification of themes (i.e., patterns and concepts) and central explanatory categories within the data, concerning the ways in which RE knowledge is processed across actors and artifacts over time. Additional codes and analytical memos were developed as they were identified in the coding process. This initial round of coding was followed by a round of axial coding in which we consolidated some of the codes generated in the open coding phase and began to identify key relationships between, and higher-level categorization of, the preliminary codes. The technique of constant comparison was employed in the development of these higher level categories (Corbin and Strauss 2008). Finally, we conducted a round of selective coding to validate the codes and relationships developed in the axial coding phase, to determine consistent patterns of interaction between social, structural, and temporal modes of cognitive distribution, and to formulate a computational framework that reflected the RE processes at play in the Rubinius community.

Categories were refined until a systematic explanation of RE distribution patterns was formulated. As a result, the final selective coding structure emerged during iterative coding of all interviews and secondary data sources. Code generation and refinement proceeded until the researchers deemed that theoretical saturation was achieved (Eisenhardt 1989; Glaser and Strauss 1967). The research findings were presented to the informants and other members of our research team with in-depth discussions feeding back into the analysis process to validate the theoretical scheme (Corbin and Strauss 2008).

Overview of the Rubinius Project

Rubinius is an implementation of the Ruby programming language involving the development of a virtual machine (VM) and related compiler for Ruby. The project is hosted on GitHub, a web-based hosting platform for developing OSS projects based on the Git version control system. As a partially sponsored OSS project, Rubinius has been relatively successful in a highly specialized domain. The project was initiated in 2005 as a hobby by Evan Phoenix who intended to write Ruby *in* Ruby, making it analogous to C and Java whose major functionalities available to programmers are written in the language itself. Subsequently, Rubinius ceased to be written purely in Ruby when the VM was rewritten in C++ to improve efficiency. The project has a large base of committers built around a core committer team, primarily due to its open commit policy (i.e., if one pull request has been accepted, the developer who submitted the pull request will become a committer and be entitled to commit directly to the project).

In late 2007, Engine Yard Company – one of the biggest privately-held companies focused on Ruby on Rails and PHP development and management – began to sponsor several committers of Rubinius to work fulltime on the project. In late 2010, the first released version of Rubinius 1.0 (Fabius) was launched. Our data collection focused primarily on the period after that initial release, because it indicated a new round of efforts to prepare for a Rubinius 2.0 release. The new release will introduce significant changes to the codebase and involve substantial updates to the base functions of the implementation with additional features, bug fixes, and significant performance improvements. Interestingly, in the middle of 2012, after seven years of enthusiastic involvement, the initiator elected not to slot himself in the core committer team, citing a change in his professional work and the intention to seek new challenges. Although it impacted the emotional commitment of the core development team somewhat, the leadership transition has not substantively impeded the efforts toward raising the functionality of Rubinius 2.0.

The requirements of the new release were determined through diverse efforts by a large number of participants. While there was no formalized approach in place to manage requirements, the informal resources generated by collaborative engagements together with robust coding, testing and coordination infrastructures (Github) and other artifacts employed play a significant role in shouldering RE practices.

Findings

We report our findings by focusing on the three restated research questions. RQ1: What are the social, structural, and temporal modes of distribution observed? RQ2: How does a computational structure emerge through the interplay of distribution elements to enable RE? RQ3: What are certain heuristics and strategies derived from the computational structure?

Social Distribution

As we have noted, social distribution refers to the distribution of cognitive workload across members of a social group. Table 2 shows the social distribution for the Rubinius case with distinct analytical categories and actors. The categories are classified according to the relative position within the Rubinius project, including 1) the focal community (i.e., social actors internal to the project), 2) sponsors (i.e., social actors neither entirely internal nor entirely external, but somewhere in between) and 3) external communities (i.e., social actors external to the project)[1]. Each category and key actors which comprise them contributed requirements knowledge to Rubinius and reflect the distributed cognitive workload of computing requirements across multiple organizational boundaries. We find that the focal community devoted their knowledge and technical expertise to RE in large part with the help of sponsors, while external communities provided an important impetus for the identification of new requirements.

The cognitive workload was significantly distributed among multiple actors within the focal community. Primarily, Rubinius relied heavily upon the vision of core committers due to their knowledge in developing Rubinius 1.0. This knowledge consisted of domain knowledge (e.g., knowledge about the Ruby environment) as well as content specific knowledge (e.g., knowledge about certain components within the

[1] Due to the commit policy described previously, we distinguish between internal and external participation based on whether social actors have commit access to the Rubinius project. In this sense, the position of sponsors cannot be deemed to be entirely internal or external, since some actors may directly contribute to the Rubinius codebase while others may not.

Rubinius context, such as virtual machine, benchmarks, and specification documents). Apart from a small number of core committers, hundreds of peripheral committers afforded cognitive support to RE computation. Their content specific knowledge focused on discovering and settling specific requirements drawing on their diverse expertise and interests. As one respondent observed:

> *"[M]ost people did have some kind of narrower focus … There were some people who worked just on the VM or even just on the code generation, I think, but a lot of the contributors worked solely on Ruby, solely on the specs or just implementing some standard functionality, like for arrays or numbers or whatever."*

Table 2 Summary of Social Distribution Mechanisms

Categories	Actors	Descriptions	Bases of Distribution/ Knowledge
Focal Community	Core Committers	The people who engage in the daily management of the project and are responsible for guiding the overall direction of the project and coordinating the development of the project. Usually, they have been involved in the project for quite a long time and contributed new functionalities or features regularly.	Domain knowledge, Content-specific knowledge
	Peripheral Committers	The people who primarily contribute to discovering, reporting or fixing bugs for the project. They contribute new functionalities or features occasionally and their involvement is sporadic, periodic or seasonal.	Content-specific knowledge
Sponsor	Engine Yard Company	A Platform-as-a-Service (PaaS) company focused on Ruby on Rails and PHP development and management, which provides financial support to several Rubinius committers.	Domain knowledge
External Communit-ies	Ruby developers/ users	The people who work on or use Ruby programming language.	Technical skills, Experience with other Ruby-related projects
	Ruby on Rails developers/users	The people who work on or use Rails, a web application framework for the Ruby programming language.	
	Engine Yard customers	The people who are clients of Engine Yard.	
	Puma users	The people who are the users of Puma, a concurrent HTTP 1.1 server for Ruby web applications.	
	Travis users	The people who are the users of Travis, a hosted continuous integration service for open source projects.	

Several peripheral committers identified themselves as 'cheerleaders.' They not only offered their content-specific knowledge to discover new features and report/fix bugs, but also persuaded others to collect and disperse requirements knowledge (e.g., what Rubinius was and what Rubinius was going to do).

The distribution of requirements was not limited to the focal community. It was complemented with the help of sponsors within the Engine Yard Company, who had an emotional association with Ruby as they "wanted Ruby to win." During the period we studied (i.e., late 2010 to late 2012), Engine Yard representatives did not propose any *formal* functional requirements for Rubinius; rather, they respected the will of the Rubinius' focal community and provided useful advice to help the community understand and settle requirements. In addition, Engine Yard sponsors facilitated the exposure of Rubinius to external communities by providing Rubinius as an option in the Engine Yard platform for their clients, so as to encourage external communities to contribute their requirements to Rubinius.

External communities refer to a variety of Ruby-related users and developers other than Rubinius committers, who contributed requirements knowledge to Rubinius based on their technical skills. Indeed, many technical requirements flowed from external communities through heterogeneous artifacts, including GitHub, the Engine Yard platform, Puma, and Travis. Thus, structural mechanisms enabled this heterogeneous social distribution. A respondent's comment provides a useful illustration of this dynamic:

> *"People will try out Rubinius and something will break and they submit a bug report, so they use the GitHub issues… There are some people that don't want to deal with Rubinius. Like they don't really care, essentially, but Travis still gives them an opportunity to test on Rubinius and maybe allow for the failure so that the Rubinius team can see what is, you know what's happening and maybe diagnose or create an issue or you know make an improvement. "*

Though Rubinius as a community is in a constant state of flux due to the voluntary nature of OSS participation, the roles related to the requirements computation and flow remained relatively stable. Thus, the skills of these actors (see Table 2) largely determined how the requirements knowledge was socially distributed.

Structural Distribution

Structural distribution refers to distribution of cognitive workload through the use of artifacts. Table 3 shows a summary of structural distribution mechanisms observed in Rubinius. Three key categories of structural artifacts – i.e., web resources, system artifacts, and communicational channels – were identified based on characteristics of the media and the cognitive functions they enabled (Metzler and Shea 2011). Importantly, these structural forms of distribution do not simply reflect the state of the requirements knowledge at a certain point of the project; rather, they were iteratively mobilized and constantly modified while requirements knowledge was located, mobilized, and transformed over time.

Web resources consisted of a set of related project web pages that were regarded as part of either input or output related to requirements computation, where requirements knowledge was rendered in both textual and graphical forms. These resources were primarily intended to support identification of related requirements knowledge for Rubinius 2.0. Specifically, they helped different actors perceive salient issues, because they *de*scribe, *pre*scribe, and *pro*scribe behaviors of the Ruby programming language in general and Rubinius in particular. Further, they set the basis for subsequent problem solving and testing associated with requirement knowledge or could trigger new requirements. The role of RubySpec – a collection of specification documents available on the web – provides a useful illustration:

> *"[RubySpec is] an executable specification part of Ruby language libraries, and Rubinius, that's what Rubinius tests itself against and it maintains a set of tags which are basically a little file that says, 'This particular RubySpec fails on Rubinius.' So this is a point that Rubinius needs to be fixed…So it was fairly easy to go and find either where the problem was or in a lot of cases it was just something hasn't implemented yet, so I could just go and start writing the method from scratch based on the behavior that it need to have according to the RubySpec."*

System artifacts are software-intensive tools incorporating automation of computational processes, which primarily supported the main cognitive functions of requirements computation. In particular, the Github platform served as Rubinius' organizational memory as it recorded what, how, and when each piece of code was created, while Matz's Ruby Interpreter (MRI), the Engine Yard platform, Puma webserver, Travis CI, and other Ruby-related projects became significant sources for new requirements generated from external communities. Therefore, much of the central activity around requirements discovery, specification, and validation centers on the Github platform, which integrates and establishes relationships to and from other system artifacts.

Communicational channels refer to media through which project related information is communicated between teams and individuals carrying out the design and implementation work. They play a pivotal role in enabling requirements knowledge to flow across individual boundaries, revealing hidden or latent requirements, and rendering them exchangeable with other people. For example, constant use of IRC serves several important cognitive functions in the creation and transformation of requirements knowledge. First, it provides a virtual interactive environment that enables group communications to elicit new requirements. Second, IRC serves as a pathway for perceiving problems, promoting problem solving, and supporting fast decision making to ensure swift action. Third, it acts as a web repository where part of the history of requirements elicitation and validation is documented and can be traced for contextualization and understanding. The following quote illustrates these roles:

> *"We did coordinate pretty much exclusively on the IRC channel… [T]he actual discussions mostly took place on IRC. I think we had a couple of conference calls maybe, but mostly IRC, and just told people what we were doing, gonna be doing."*

Table 3 Summary of Structural Distribution Mechanisms			
Categories	**Artifacts**	**Descriptions**	**Cognitive Functions[2]**
Web resources	Rubinius website	A website for introducing, documenting, and blogging the Rubinius project	Perception, Memory
	RSpec	A specification for describing the expected behavior of Ruby programming language	Perception, Problem solving
	RDocs	Documentation generated for the Ruby Programming language	Perception, Problem solving
	RubySpec	An executable specification for the Ruby programming language that is syntax-compatible with RSpec	Perception, Problem solving
System artifacts	GitHub	A web-based hosting service for software development projects	Computation, Perception, Problem solving, Decision making, Memory
	Engine Yard platform	A cloud Platform-as-a-Service (PaaS) for Ruby on Rails, PHP and Node.js applications	Computation
	Puma webserver	A concurrent HTTP 1.1 server for Ruby web applications	Computation
	Travis CI	A hosted continuous integration service for open source projects	Computation
	MRI	A standard Ruby interpreter	Computation, Perception, Problem solving, Decision making
	Tools	Git, Adebugger, Gdb, Insiter, to name a few (Different developers intended to use different tools with respect to personal preferences.)	Computation, Problem solving
	Ruby related projects	Projects which test on Rubinius	Computation
Communica-tional channels	Mailing list		Communication, Perception, Problem solving, Decision making, Memory
	Skype		Communication, Decision making, Problem Solving
	IRC channel		Communication, Perception, Problem solving, Decision making, Memory
	Phone call		Communication, Decision making, Problem solving

Temporal Distribution

Temporal distribution refers to distribution of cognitive workload over time. It is achieved through the configuration of socially and structurally distributed elements. Table 4 shows the temporal analytical categories as well as configured social and structural components for each temporal category to understand how requirements evolved.

[2] Cognitive functions are used here to describe the basic functions that a given artifact provides to shoulder the cognitive system. We adopted the taxonomy of cognitive functions from Metzler and Shea (2011).

Table 4 Summary of Temporal Distribution Mechanisms			
Categories	**Descriptions**	**Social Elements**	**Structural Elements**
Embedded requirements in root artifacts	Requirements embodied within existing the artifacts upon which the project is based	Focal community Ruby developers and users	MRI
Embedded requirements in distal artifacts	Requirements embodied within existing artifacts which the project intends to be compatible with	External communities	Engine Yard platform Puma webserver Travis CI Ruby related projects
Early-stage translation	Requirements are translated from distributed abstractions into tangible midrange artifacts that are incorporated into the structural distribution	Focal community	Rubinius website Rubinius code and specs on Github
Late-stage translation	Requirements are translated from particularized experiments into a wholly functional baseline artifact, which becomes a part of the structural distribution	Core committers	Rubinius side branches on Github Rubinius baseline implementation on Github

Embedded requirements in root artifacts refer to requirements embodied within existing system artifacts that offer guidelines for the ongoing implementation. These requirements are expressed in the standard implementation guidelines of the Ruby programming language (i.e., MRI). They reflect prior cognitive efforts of past and present Ruby developers and users and lay a material basis upon which Rubinius is built. Each instantiation of a runtime environment must be able to replicate this standard more or less perfectly. Indeed, a large number of requirements of Rubinius have been established *a priori,* based on the capabilities and properties of MRI with the consensus agreement that MRI is an essential reference in functionalities. This kind of embedded requirements was also informed by what Pollock and Williams found in their analysis of generic requirements that transcend the place of production across many contexts (Pollock and Williams 2009). As one respondent stated:

> *"MRI is the de facto standard. You know there is no language standard other than what MRI does…A lot of Rubinius work is actually basically reverse engineering what MRI does, figuring out its behavior."*

Embedded requirements in distal artifacts are requirements that are "inherited" by Rubinius from multiple Ruby-related projects in which external communities run Rubinius to test functionality. These external communities either actively engage in testing their projects on Rubinius, or merely test their projects on Engine Yard platform/Puma webserver/Travis CI, which provide Rubinius as one of their runtime environments. The continuous flow of requirements from test failures broadens the scope of cognitive distribution to external communities and artifacts with which Rubinius is compatible. This enables exogenous requirements embedded in distal artifacts to be uncovered which need to be accounted for within the Rubinius codebase itself. Additionally, these embedded requirements provide insights into how requirements knowledge can be transferred across social and structural boundaries:

> *"Travis provides the ability for people who are writing libraries and applications to easily test across multiple Ruby organizations… Travis users basically will go in and say 'I want to build on the nightly builder on Rubinius. I want to build a weekly release. I want to build a most recent release candidate.' They can easily specify their level of engaging you know Rubinius changes and then just watch and see whether their project passes or failed…So it's obviously you can very easily see if something fails, instead of someone saying 'Hey, I was trying to run my test'. They just link us to their results on Travis and say 'Here, this is what I get. It's having this error.' and we're like 'Oh, okay… [W]e can just figure it out directly from the error output, the back trace or whatever on Travis."*

Early-stage translation transforms successive waves of representations from distributed and relatively abstract forms (e.g., informal verbal, graphical and literal forms) to more explicit and tangible forms. It involves a process that the focal community converts previously abstracted representations into code and specs, so that certain cognitive frames get inscribed into a set of artifacts (Yoo et al. 2008). Hence, the transformation of requirements knowledge towards new forms draws on the past configuration of social and structural elements.

Late-stage translation involves the continuously integration of experiments (i.e., trials and errors through which requirements knowledge are conveyed into executable forms) into a baseline implementation. Thus, the baseline implementation merged by the core of the focal community is superimposed upon existing forms of structural distribution. In this way, cognitive frames residing in individual experiments become constructed into wholly functional software that works. At this point, requirements no longer are unsystematic or partial, reflective of limited experiments; rather, they have been situated in a relatively consistent and 'stabilized' implementation.

An Emergent Computational Framework

The overall computational framework of OSS RE as distributed cognition is illustrated in Figure 1 which depicts how a requirement 'flows' through the system by touching different artifacts and actors. In other words, the figure depicts how the three-part task structure of requirements discovery, specification, and validation is enacted by configuring three separate subsystems entailing distinct social and structural distribution patterns and the computational flows between them. Moreover, the framework reveals how distinct representations (i.e., the totality of socially and structurally distributed representations in the system) are recursively mobilized and related to one another from discovery through validation.

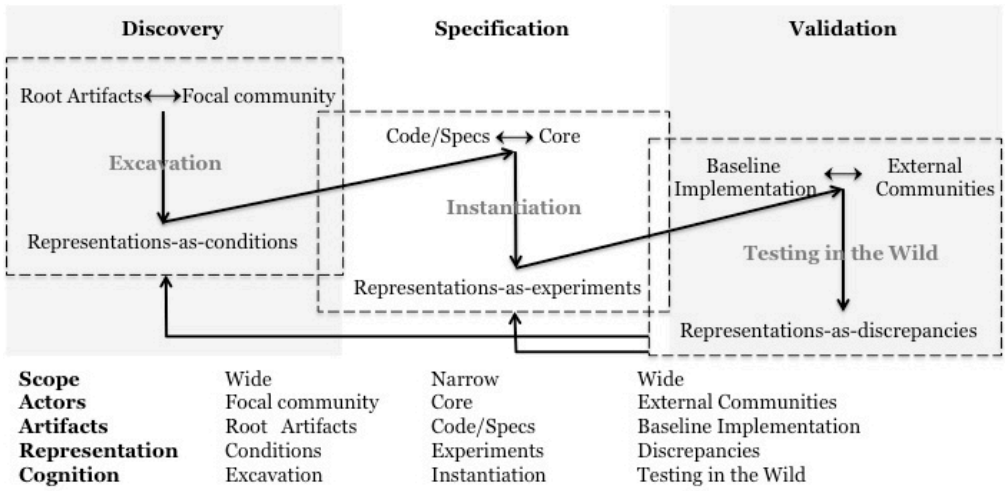

Figure 1. Computational Framework of RE in OSS

Before elaborating each computational mechanism and system in great detail, we provide an illustrative vignette explaining how an important feature, implementation of concurrency (a meta requirement about parallel processing) within Rubinius, traversed the systems. This brief sketch is emblematic of the way in which requirements knowledge is discovered, specified, and validated in the Rubinius project. One day, in the midst of regular IRC channel conversations among committers, one of the Rubinius committers proposed the development of concurrency functionality that MRI did not yet support. Given the increasing prominence of multiple processor cores in the design of computers and even mobile devices, concurrency was acknowledged as a clear technology trend worthy of the community's attention. The core committers therefore endeavored to give it a try so as to improve Rubinius's performance. Thus, the idea which had surfaced in the IRC conversations was translated into a series of informal specs and corresponding pieces of code. After several days of exploratory programming effort, the core committers

concluded that concurrency might, indeed, be feasible, so they established a side branch (Hydra) hosted on the Github platform to instantiate the concurrency functionality through detailed coding and debugging. When the side branch code proved to be relatively stable, it was integrated into the master branch to form a baseline implementation, which could be executed as fully functional software. The baseline implementation was then ready for a wider community to download and play with. Various individuals and communities (e.g., Engine Yard customers, Puma and Travis users, and other Ruby or Ruby on Rails developers and users) began testing their personal projects on Rubinius to determine, if it fit their local needs. In some cases, they tested Rubinius against RubySpec and MRI to see whether or not it worked as purported. If any issues were identified, the "testers" would submit feature requests or report bugs either via IRC/mailing lists or via Github/Engine Yard/Puma IO/Travis CI platform. This feedback to the Rubinius community served as a signal of additional requirements to be (re)computed.

Next, we delineate the computational processes and the dynamic flow of representations illustrated in the vignette so as to discern the emerging computational structure of the OSS community and articulate certain heuristics and strategies for requirements computation.

Discovery – In this phase, requirements are discovered through a sort of archeological *excavation* (Luckham 2001) of the rich social and structural forms upon which the project is founded. Standing 'root' artifacts – artifacts such as MRI, establish basic structural constraints (creating meta-requirements) to which each system instantiation (with its requirements) must conform. By drawing widely on such root artifacts, specific actors continuously excavate requirements that are embedded in root artifacts or non-fulfilled by root artifacts, but corresponding to the larger technical environment. Excavation involves actions by the focal community to extract knowledge embedded in root artifacts, to integrate the extracted knowledge within the minds of the human actors, and to express prospects of the project in informal representations that can be exchanged with others. Representations of such excavation activities include suggestions and challenges voiced in IRC or descriptions of an envisioned future for the project in a series of Skype dialogues. We refer to them as *representations-as-conditions* (i.e., propagated representations of the verbal, graphical, and literal formations of requirements knowledge). Throughout the excavation process, the focal community is likely to marshal the largest amounts of resources towards computing the set of requirements that it deems to be important based on a constant matching of competencies and opportunities. As latent requirements have been excavated and transformed into tangible conditions, they can be carried forward to specification. This movement from discovery to specification reflects the process we have labeled *early-stage translation* (see the subsection of "Temporal Distribution," above).

Specification – In this phase, requirements are specified through the interplay of the core of the focal community as code and specs are derived from former representations-as-conditions. Due to task complexity, specification involves a narrower scope of actors and artifacts (a smaller scale of social and structural distribution). During this task, the core draws on programming expertise to arrange code and specs in ways that convert the requirements knowledge into readable and executable forms. We refer to this as *instantiation* where social and structural distribution interact to articulate specific requirements discovered in the previous stage. The instantiation process involves trial and error (i.e., *representation-as-experiments*) through which actors seek to extend discovered requirements into functional and technical implications. Different requirements may often be instantiated in parallel. When experiments composed of working code and feasible specs are produced, the newly computed representations-as-experiments are ready to be transmitted to the last phase – validation. The movement from the specification phase to that of validation is denoted *late-stage translation* (see "Temporal Distribution" subsection).

Validation – In the last phase, a baseline implementation becomes part of the standard structural form called code baseline which is transmitted to the wider community, stressing test under various circumstances. Cognitive effort in this task reflects *testing-in-the-wild* through the scaffolding of various artifacts (Clark 1998) – MRI, RubySpec, Engine Yard platform, Puma IO, Travis CI, etc. This process involves multiple developers and users attempting to integrate Rubinius into their local technical portfolios, as a means through which to achieve their own goals (i.e. embedded requirements in distal artifacts). Therefore, it is the difficulty in using Rubinius for a large set of heterogeneous goals in concert with a wide range of other technical artifacts that generates *representations-as-discrepancies* in forms of bug report, feature requests. Those representations-as-discrepancies refer to the incompatibility between

performance expectations of Rubinius and its actual performance. Accordingly, the cognitive outcome creates a recursive loop that feeds back to discovery and specification tasks.

Recursion – The feedback mechanism from validation back to discovery and specification entails both social and structural distribution mechanisms. It invokes new combinations of actors and artifacts to re-compute successive instantiations of requirements by excavating additional requirements so as to maintain integrity across emerging latent set of requirements, tangible conditions, testable experiments, and (ultimately) codebase that is free of discrepancies. Although the figure shows the computational workflow as a sequence, it does not mean that RE in OSS follows a waterfall principle; rather, the three tasks evolve through iterative loops for each requirement as new requirements enter and old ones get continuously refined until a relatively stable requirements set is achieved as expressed in a release.

Contributions and Discussion

Our inquiry offers a number of significant contributions and important insights for both systems development practitioners and IS scholars. By attending to the sociotechnical distribution of cognitive effort within OSS projects, IS practitioners can more effectively leverage the resources needed for different facets of RE computation. In particular, this research provides valuable insights into how an OSS community can benefit from reverse engineering embedded requirements in root artifacts as well as acquiring a continuous flow of embedded requirements in distal artifacts. The research also underscores the role that artifacts play as dynamic vehicles for requirements dissemination. At the same time, the integration of peripheral committers and other external communities serves as a channel to enable new requirements to emerge continuously as more established requirements are addressed and refined. From a practice perspective, the framing of requirements-oriented activities as a distributed cognitive process can alert OSS project leaders to the critical role played by different stakeholders as well as artifacts within the system. In light of the very high rates of OSS project failure, an enhanced understanding of the elements and processes that contribute to effective identification of system requirements within "OSS ecosystems" (Thomas and Hunt 2004) is extremely valuable. For example, our study highlights the idea that integration of relatively peripheral participants into the process of supplying requirements enhances the robustness of an OSS project. Furthermore, the research underscores the critical role that structural artifacts play in enabling such engagement. From this perspective, project managers and core developers need to attend to community maintenance and the expansion of both social and structural distribution mechanisms. Such attention can be fostered by the discovery of embedded requirements and that mitigate the need for "greenfield" discovery. Finally, the framework that we develop reveals how different elements within the distributed cognitive system can be configured at various stages of requirements computation to enhance the flow of requirements knowledge within the system.

For the IS research community, this paper contributes to our understanding of OSS projects by taking both actors and artifacts into account and integrating them in a single theoretical perspective. The application of the theory of distributed cognition to OSS RE is novel in that it offers a strong theoretical (cognitive) foundation for approaching RE in a flexible way and helps to unpack how RE is conducted in OSS settings. We identify for the first time multiple forms of social and structural cognitive distribution present in OSS efforts and how they become configured differently across various stages of a requirements computation. These mechanisms, and the different forms of distribution that they entail, reveal the importance of differentiating between the alternative RE processes that unfold at different stages of RE computation. In addition, these observations invite scholars to focus on differences in the social and structural distributions in each stage. In addition, the study illustrates the value of analyzing RE as a sociotechnical distributed cognitive process, providing a new route for scholars to explore the dynamic intertwining of diverse actors and artifacts. We hope that our findings encourage other researchers to delve more deeply into the varying RE practices of OSS and other software development environments. In particular, scholars can use the framework to isolate specific mechanisms through which representations become transformed into requirements and subsequently usable artifacts.

The research suggests several avenues for additional research. Our application of distributed cognition to develop a model of OSS requirements computation can be applied to other software development environments, including traditional structured development, COTS-based development, and agile software development. The computational structures employed in these different environments are likely to vary widely with respect to social, structural, and temporal forms of cognitive distribution, the

configurations in which the elements are arranged, and the general computational structure and heuristics. Such application to multiple environments could foster inductive theory-driven comparison and identification of computational configurations that influence project success and failure rates, developer and user satisfaction, or perceived innovativeness of solutions.

The generalizability of these findings is limited by the fact that we analyzed a single representative case. While the size of OSS communities and the degree to which such communities utilize external artifacts varies to a large degree, we contend that the sociotechnical cognitive processes we identify will remain largely consistent with other OSS environments due to high similarity of social organization principles and deployed artifacts in many OSS projects. Nevertheless, we do acknowledge that Rubinius shares a number of idiosyncratic characteristics such as an open commit policy, highly technical nature, and a high level of interconnectedness with multiple external artifacts. These unqiue characeteristics may limit the generalizability of our model to all types of OSS projects. To address this issue, we intend to expand the study in future research to sample other types of OSS projects.

Conclusion

In this research, we approached RE in OSS as a sociotechnical, distributed cognitive process. Through an empirical analysis of the Rubinius OSS project, we illustrate the ways in which OSS RE processes are distributed along social, structural, and temporal dimensions (RQ1). In addition, we provide insight into how the social, structural, and temporal mechanisms interact to enable the cognitive system to successfully compute requirements through opportunity-driven matching (RQ2). During the execution of three primary RE tasks (i.e., discovery, specification, and validation), the cognitive system is configured to address three temporally-ordered and interconnected cognitive pursuits – i.e., excavation, instantiation, and testing-in-the-wild. These cognitive processes transform representations of requirements through a succession of forms (representation-as-conditions, representation-as-experiments, and representation-as-discrepancies) into a collectively accepted understanding of the design requirements for a release (RQ3).

The study offers important insights for both research and practice in OSS environments. From a research perspective, the study underscores the importance of attending to the interplay of human actors and structural elements in software development environments. In addition, we present a novel and theoretically-grounded approach to the study of requirements efforts in OSS projects. From a practice perspective, the study highlights a range of distributed cognitive dynamics which can inform OSS project leaders as they seek to support and maintain vibrant development communities.

Acknowledgements

This study benefited from financial support from National Science Foundation (Grant # 1217345). We are also grateful for the interviewees for their time, insight, and enthusiasm around the research topic. We further thank Amol Kharabe and Veeresh Thummadi for their constructive comments on early drafts of this paper.

References

Asundi, J., and Jayant, R. 2007. "Patch Review Processes in Open Source Software Development Communities: A Comparative Case Study," In *Proceedings of the 40th Annual Hawaii International Conference on System Science*.

Aurum, A., and Wohlin, C. 2005. "Requirements Engineering: Setting the Context," In *Engineering and Managing Software Requirements*, A. Aurum and C. Wohlin (eds.), Springer, pp. 1–15.

Bahill, T. A., and Henderson, S. J. 2005. "Requirements Development, Verification, and Validation Exhibited in Famous Failures," *Systems Engineering* (8:1), pp. 1–14.

Brooks, F. P. 1987. "No silver bullet: Essence and accidents of software engineering," *IEEE Computer* (20:4), pp. 10–19.

Cheng, B. H., and Atlee, J. M. 2009. "Current and Future Research Directions in Requirements Engineering," In *Design Requirements Engineering: A Ten-Year Perspective*, K. J. Lyytinen, P. Loucopoulos, J. Mylopoulos, and W. N. Robinson (eds.), Berlin, Germany: Springer, pp. 11–43.

Clark, A. 1998. *Being There: Putting Brain, Body and World Together Again*, Cambridge, MA: MIT Press.
Corbin, J., and Strauss, A. 2008. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*, Thousand Oaks, CA: SAGE.
Crowston, K., and Howison, J. 2005. "The Social Structure of Free and Open Source Software Development," *First Monday* (10:2).
Crowston, K., and Kammerer, E. 1998. "Coordination and Collective Mind in Software Requirements Development," *IBM Systems Journal* (37:2), pp. 227–245.
Crowston, K., Li, Q., Wei, K., Eseryel, Y. U., and Howison, J. 2007. "Self-Organization of Teams for Free/Libre Open Source Software Development," *Information and Software Technology* (49:6), pp. 564–575.
Damian, D. E., and Zowghi, D. 2003. "Requirements Engineering Challenges in Multi-Site Software Development Organizations," *Requirements Engineering Journal* (8:3), pp. 149–160.
Damian, D., Helms, R., Kwan, I., Marczak, S., and Koelewijn, B. 2013. "The Role of Domain Knowledge and Cross-Functional Communication in Socio-Technical Coordination," In *Proceedings of the 2013 International Conference on Software Engineering*, IEEE press, pp. 442–451.
Emery, F., and Trist, E. 1960. "Socio-Technical Systems," In *Management Science Models and Techniques*, C. W. Churchman and M. Verhulst (eds.), London.
Ernst, N. A., and Murphy, G. C. 2012. "Case Studies in Just-In-Time Requirements Analysis," In *IEEE 2nd International Workshop on Empirical Requirements Engineering*, pp. 25–32.
Hansen, S., Berente, N., and Lyytinen, K. 2009. "Requirements in the 21st Century: Current Practice and Emerging Trends," In *Design Requirements Engineering: A Ten-Year Perspective*, K. Lyytinen, P. Loucopoulos, J. Mylopoulos, and B. Robinson (eds.), Berlin, Germany: Springer, pp. 44–87.
Hansen, S. W., Robinson, W. N., and Lyytinen, K. J. 2012. "Computing Requirements: Cognitive Approaches to Distributed Requirements Engineering," In *2012 45th Hawaii International Conference on System Sciences*, pp. 5224–5233.
Hickey, A. M., and Davis, A. M. 2003. "Elicitation Technique Selection: How Do Experts Do It?," In *Proceedings of the 11th IEEE International Requirements Engineering Conference*, pp. 169–178.
Hutchins, E. 1995. *Cognition in the Wild*, Cambridge, MA: MIT Press.
Hutchins, E. 2001. "Distributed Cognition," In *International Encyclopedia of the Social and Behavior Sciences*, Elsevier Ltd, pp. 2068–2072.
Hutchins, E., and Klausen, T. 1996. "Distributed Cognition in an Airline Cockpit," In *Cognition and Communication at Work*, Y. Engeström and D. Middleton (eds.), Cambridge University Press, pp. 15–34.
Johnson-Laird, P. 1989. *The Computer and the Mind: An introduction to Cognitive Science*, Harvard University Press, pp. 448.
Von Krogh, G., Haefliger, S., Spaeth, S., and Wallin, M. W. 2012. "Carrots and Rainbows: Motivation and Social Practice in Open Source Software Development," *MIS Quarterly* (36:2), pp. 649–676.
Van Lamsweerde, A. 2000. "Requirements Engineering in the Year 00: A Research Perspective," In *Proceedings of the 22nd International Conference on Software Engineering*, Limerick: ACM Press, pp. 5–19.
Luckham, D. C. 2001. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*, Boston, MA: Addison-Wesley.
Markus, M. L., Manville, B., and Agres, C. E. 2000. "What Makes a Virtual Organization Work?," *Sloan Managment Review* (42:1), pp. 13–26.
Mathiassen, L., Tuunanen, T., Saarinen, T., and Rossi, M. 2007. "A Contingency Model for Requirements Development," *Journal of the Association for Information Systems* (8:11), pp. 569–597.
Metzler, T., and Shea, K. 2011. "Taxonomy of Cognitive Functions," In *Proceedings of the 18th International Conference on Engineering Design*, pp. 330–341.
Mockus, A., Fielding, R. T., and Herbsleb, J. D. 2002. "Two Case Studies of Open Source Software Development: Apache and Mozilla," *ACM Transactions on Software Engineering and Methodology* (11:3), pp. 309–346.
Mumford, E. 1985. *Sociotechnical Systems Design: Evolving Theory and Practice*, Manchester Business School and Centre for Business Research.
Noll, J., and Liu, W.-M. 2010. "Requirements Elicitation in Open Source Software Development," In *Proceedings of the 3rd International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, ACM Press, pp. 35–40.

Norman, D. A. 1993. *Things That Make us Smart: Defending Human Attributes in the Age of the Machine*, Cambridge, MA: Perseus Books.
Perry, M. 1999. "The Application of Individually and Socially Distributed Cognition in Workplace Studies: Two Peas in a Pod?," In *Proceedings of European Conference on Cognitive Science*, Siena, Italy, pp. 87–92.
Pollock, N., and Williams, R. 2009. "Global Software and Its Provenance: Generification Work in the Production of Organisational Software Packages," In *Configuring User-Designer Relations: Interdisciplinary Perspective*, V. Alex, M. Hartswood, R. Procter, M. Rouncefield, R. Slack, and M. Büscher (eds.), London: Springer, pp. 193–218.
Raymond, E. 1999. "The Cathedral and the Bazaar," *Knowledge, Technology & Policy* (12:3), pp. 23–49.
Robles, G., and Gonzalez-barahona, J. M. 2006. "Contributor Turnover in Libre Software Projects," In *Open Source Systems*, E. Damian, B. Fitzgerald, W. Scacchi, M. Scotto, and G. Succi (eds.), Boston: Springer, pp. 273–286.
Rogers, Y., and Ellis, J. 1994. "Distributed Cognition: An Alternative Framework for Analysing and Explaining Collaborative Working," *Journal of Information Technology* (9:2), pp. 119–128.
Scacchi, W. 2002. "Understanding the Requirements for Developing Open Source Software Systems," *IEE Proceedings Software* (149:1), pp. 24–39.
Scacchi, W. 2005. "Socio-Technical Interaction Networks in Free/Open Source Software Development Processes," In *Software Process Modeling*, S. Acuña and N. Juristo (eds.), Springer, pp. 1–27.
Scacchi, W. 2009. "Understanding Requirements for Open Source Software," In *Design Requirements Engineering: A Ten-Year Perspective*, K. Lyytinen, P. Loucopoulos, J. Mylopoulos, and B. Robinson (eds.), Berlin, Germany: Springer, pp. 467–494.
Shah, S. K. 2006. "Motivation, Governance, and the Viability of Hybrid Forms in Open Source Software Development," *Management Science* (52:7), pp. 1000–1014.
Simon, H., and Kaplan, C. 1989. "Foundations of Cognitive Science," In *The Foundations of Cognitive Science*, M. I. Posner (ed.), Cambridge, MA: MIT Press.
Strauss, A., and Corbin, J. 1990. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*, Newbury Park, CA: SAGE.
Thomas, D., and Hunt, A. 2004. "Open Source Ecosystems," *IEEE Software* (21:4), pp. 89–91.
Thummadi, B. V., Lyytinen, K., and Hansen, S. 2011. "Quality in Requirements Engineering (RE) Explained Using Distributed Cognition: A Case of Open Source Development," *Sprouts: Working Papers on Information Systems* (11).
Valverde, S., and Solé, R. V. 2007. "Self-Organization versus Hierarchy in Open-Source Social Networks," *Physical Review E* (76:4).
Vlas, R., and Vlas, C. 2011. "A Requirements-Based Analysis of Success in Open-Source Software Development Projects," In *Proceedings of the 17th Americas Conference on Information Systems*, Detroit, Michigan.
Ye, Y., and Kishida, K. 2003. "Toward an Understanding of the Motivation of Open Source Software Developers," In *Proceedings of the 25th International Conference on Software Engineering*, pp. 419–429.
Yin, R. K. 2009. *Case Study Research: Design and Methods*, Thousand Oaks, CA: SAGE.
Yoo, Y., Lyytinen, K., and Boland, R. J. 2008. "Distributed Innovation in Classes of Networks," In *Proceedings of the 41st Annual Hawaii International Conference on System Sciences*, Waikoloa, HI.
Zhang, J., and Norman, D. 1994. "Representations in Distributed Cognitive Tasks," *Cognitive Science* (18:1), pp. 87–122.