Journal of the Association for Information Systems (2018) **19**(12), 1217-1252
doi: 10.17705/1jais.00525

RESEARCH PAPER

ISSN 1536-9323

"Computing" Requirements for Open Source Software: A Distributed Cognitive Approach

Xuan Xiao[1], Aron Lindberg[2], Sean Hansen[3], Kalle Lyytinen[4]

[1]Guangzhou University, China, xiaoxuan@gzhu.edu.cn
[2]Stevens Institute of Technology, U.S.A., aron.lindberg@stevens.edu
[3]Rochester Institute of Technology, U.S.A., shansen@saunders.rit.edu
[4]Case Western Reserve University, U.S.A., kalle@case.edu

Abstract

Most requirements engineering (RE) research has been conducted in the context of structured and agile software development. Software, however, is increasingly developed in open source software (OSS) forms which have several unique characteristics. In this study, we approach OSS RE as a sociotechnical, distributed cognitive process where distributed actors "compute" requirements—i.e., transform requirements-related knowledge into forms that foster a shared understanding of what the software is going to do and how it can be implemented. Such computation takes place through social sharing of knowledge and the use of heterogeneous artifacts. To illustrate the value of this approach, we conduct a case study of a popular OSS project, Rubinius—a runtime environment for the Ruby programming language—and identify ways in which cognitive workload associated with RE becomes distributed socially, structurally, and temporally across actors and artifacts. We generalize our observations into an analytic framework of OSS RE, which delineates three stages of requirements computation: excavation, instantiation, and testing-in-the-wild. We show how the distributed, dynamic, and heterogeneous computational structure underlying OSS development builds an effective mechanism for managing requirements. Our study contributes to sorely needed theorizing of appropriate RE processes within highly distributed environments as it identifies and articulates several novel mechanisms that undergird cognitive processes associated with distributed forms of RE.

Keywords: Open Source Software Development, Requirements Engineering, Distributed Cognition, Case Study, Heuristics, Ruby Programming Language

Sandeep Purao was the accepting senior editor. This research article was submitted on November 18, 2015 and went through two revisions.

1 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 (Brooks, 1995, p. 199).*

Since the 1970s, *requirements engineering* (RE)—a cohesive set of tasks that focus on discovering, specifying, and validating what software should do in its use context—has constituted an essential challenge in software development (Brooks, 1995; Cheng & Atlee, 2009). Software requirements are known to be uncertain, inconsistent, and temporally volatile (Damian, Helms, Kwan, Marczak, & Koelewijn, 2013; Mathiassen, Tuunanen, Saarinen, & Rossi, 2007), and weaknesses in dealing with these

requirements' properties create a constant source of project stress and outright failure (Aurum & Wohlin, 2005; Hickey & Davis, 2003). Over the last decade, software development has grown increasingly distributed and dynamic, making the RE challenge even more formidable. A prominent example of this change is the emergence of open source software (OSS) development where volunteers, who work under open licensing agreements, deliver software in a collaborative and distributed manner through public development platforms such as SourceForge and GitHub (Crowston, Li, Wei, Eseryel, & Howison, 2007). These arrangements allow source code and related design information to be freely used, modified, and disseminated among large numbers of geographically distributed, autonomous developers. A consequence of the distributed structure of OSS projects is that they have ambiguous "customer" roles and few "stages" or deadlines. Rather, the software evolves organically to meet the needs and aspirations of its attendant community. The success of several OSS projects and communities implies that somehow "the right requirements" get decided upon, addressed, and eventually implemented "correctly." The ways in which this outcome is achieved are the primary focus of the present study.

Traditional structured software development follows well-defined RE principles such as time-bounded discovery and documentation of needs conducted by a stable team of analysts in relation to a clearly identified set of users. This arrangement provides clarity around social and technological arrangements that identify and manage requirements. Recently, with the rising popularity of agile methods, some traditional RE activities (e.g., formal requirements documentation and modeling) have been replaced by dynamic social practices such as face-to-face interaction and iterative prototyping. Because of the distributed and voluntary nature of OSS, it follows neither structured nor agile forms of RE (Crowston & Kammerer, 1998; Hansen, Berente, & Lyytinen, 2009). OSS projects eschew RE formalisms, such as detailed project plans and formal specifications (Scacchi, 2002) that form the foundation for structured approaches to RE, but they also rely on few and intermittent face-to-face interactions due to their voluntary and distributed natures. In contrast, requirements knowledge in OSS is socially embedded in various artifacts and conversations, highly diverse, and widely distributed (Mockus, Fielding, & Herbsleb, 2002). Due to high turnover rates (Robles & Gonzalez-Barahona, 2006), diverse motivations (Shah, 2006), and self-assignment (Crowston et al., 2007), the configuration of actors and artifacts in OSS projects is also inherently volatile. How then do OSS projects successfully carry out RE given their dynamic and distributed natures?

Our knowledge of OSS RE remains limited. The small number of studies conducted suggest that OSS projects settle requirements "on the fly" as participants resolve and negotiate requirements by relying on multiple networks of communication (Scacchi, 2002, 2009). Past research also sheds light on how OSS processes differ from traditional RE (Crowston et al., 2007; Scacchi, 2002, 2009; Shah, 2006). Yet, extant studies do not provide a comprehensive account of how RE tasks are actually accomplished and how the interweaving of participants and artifacts ultimately makes requirements discovery, specification, and validation possible. In particular, past studies do not explain why OSS RE activities appear to be stable, resilient, and effective despite the dynamic and heterogeneous nature of the requirements knowledge, participants, and related processes. Simply put, there is a substantial gap in the understanding of how and why OSS projects successfully manage their requirements. This study seeks to address this gap by focusing on how the social, structural, and temporal organization of OSS activities enables effective sharing and coordination of requirements knowledge.

In addressing this research question, we approach RE as a cognitive task—a knowledge-oriented effort in which participants seek to make sense of what software should do and why. Given the heterogeneity and dynamism of knowledge and its distribution across actors and artifacts during OSS (Hansen, Robinson, & Lyytinen, 2012), our analysis draws upon the theory of distributed cognition (DCog) (Hutchins, 1995; Hutchins & Klausen, 1996). This theory expands traditional models of cognition centered on the mental processes of individuals to include an analysis of distributed, knowledge-related activities that involve multiple heterogeneous entities—both human and artificial. At the same time, DCog theory extends cognitive science's fundamental view of "cognition as computation"—i.e., the idea that cognition is about the creation and manipulation of symbol systems (Simon, 1980)—into broader sociotechnical settings. Accordingly, we treat OSS RE as a stream of distributed cognitive activities focused on what we refer to as the process of *requirements computation*—the transformation of vague requirements knowledge embedded and discovered within a broader OSS environment, through a set of artifacts and social mappings, into implementable code that realizes those requirements. Such computation is both enabled and constrained by the dynamic reconfiguration of actors and artifacts, often framed by simple heuristic rules that guide the organization and execution of the process.

We probe the essential characteristics of RE as sociotechnical "computation" through an exploratory case study of a typical midsized OSS project—

Rubinius (a Ruby programming language runtime environment: https://rubinius.com/). Through the case study, we demonstrate that computing requirements in an OSS project are made possible by a complex and dynamic cognitive system consisting of constellations of actors, artifacts, and temporal structuring mechanisms. Requirements knowledge is discovered through excavating neighboring artifacts and technological environments; it is specified through code change units identified in ongoing discussions and expressed in code segments; and it is validated "in the wild" by sharing the code segments across a broader OSS community. Specific temporal structuring mechanisms such as design heuristics focus the cognitive effort on particular elements and computational goals at different stages of the RE process. These insights provide us with a rich foundation for theorizing the mechanisms that make highly distributed RE successful within an OSS context. The remainder of the paper is organized as follows. In the next section, we provide a brief introduction to how RE is changing due to new forms of software development such as agile methods and OSS. This is followed by a review of DCog theory which frames the conceptual foundation of our case study. Thereafter, we present an overview of the case study and detail its key findings. We conclude by formulating a model of RE computation in OSS environments and discuss implications of our findings for RE research and practice.

2 Theoretical Background

2.1 Requirements Engineering and the Changing Face of Development

RE refers to processes carried out by software developers and other stakeholders to achieve a cohesive set of requirements for software. These requirements express, in a clear and collectively accepted form, what the software is going to deliver and why. In general, RE processes address three necessary facets associated with requirements knowledge: discovery, specification, and validation (Hansen et al., 2009). *Discovery* concerns the identification of functional needs associated with a given system as well as constraints to which it must conform (Mathiassen et al., 2007)—i.e., it addresses the question: What do we need to build? *Specification* focuses on the explicit articulation of discovered needs, including deriving in detail their functional and technical consequences (Hansen et al., 2009)—i.e., it addresses the question: How can we articulate, express, and share those needs in the most effective way? Finally, *validation* ensures that the requirements are correct, complete, and consistent, and therefore implementable, commonly understood, and accepted (Bahill & Henderson, 2005)—i.e., it address the question: How can we be sure that we are working in the right direction with the correct requirements?

In structured development—commonly referred to as the *waterfall model*—these three RE facets are executed in a sequential manner (Larman & Basili, 2003). In addition, waterfall development is largely oriented toward the specification component and lays a heavy emphasis on formal (often voluminous) documentation of requirements, which incorporates both natural language and modeling elements (Hansen et al., 2009). While the rigid waterfall structure was developed as a response to the chaotic approaches to RE/software development that preceded it (Fitzgerald & Avison, 2003), the slow pace and inflexibility of the approach has engendered severe criticism (Jarke, Loucopoulos, Lyytinen, Mylopoulos, & Robinson, 2011).

To cope with the increasing rate of change in the contemporary business environment, a less rigid, iterative form of development has become widely popular—agile development. Agile development downplays the importance of formal documentation, because its advocates argue that the elimination of formal documentation can be compensated for through interpersonal communication, rapid prototyping, and continual replanning (Beck & Andres, 2004; Cao & Ramesh, 2008). Therefore, agile development emphasizes direct and frequent face-to-face interactions between design stakeholders, including development team members and clients, as a means of discovering and validatimg requirements (Conboy, Coyle, Wang, & Pikkarainen, 2011; Highsmith & Cockburn, 2001). Being prototype-driven and iterative, it also aids in discovering and specifying requirements, because prototypes help developers focus their discussions in ways that integrate requirements discovery, specification, and validation (Cao & Ramesh, 2008; Ramesh, Mohan, & Cao, 2012). Finally, continual replanning and reprioritization introduces flexible attention to various requirements questions (Port & Bui, 2009; Vidgen & Wang, 2009).

OSS development shares some characteristics with the agile approach, such as the commitment to iterative development. However, from an RE perspective, the OSS model introduces several distinct challenges, which originate from the highly distributed nature of the development team (a marked contrast to agile methods' ideal of face-to-face communications) and the absence of control in OSS due to the reliance on a voluntary workforce. As a result, the management of requirements knowledge in OSS projects has several idiosyncratic characteristics. The early studies characterized OSS development as a chaotic, bazaar-like engagement (Raymond, 1999) where the idea of coordinated RE process would largely disappear and become fully an individual concern. Yet, recent evidence suggests that OSS

communities maintain a relatively stable social order through constant mobilization and integration of expertise and related community building by using free-flowing electronic communications and "community events." Crowston and Howison (2005) observed that a typical OSS development community organizes itself in an "onion" structure, with a relatively small core of dedicated contributors surrounded by a larger peripheral community marked by lower levels of participation and technical skill. The core acts as the gatekeepers of project participation, mission, and contribution (Asundi & Jayant, 2007). It also controls and coordinates knowledge assets necessary to develop and implement the code and determines the directions of the project (Mockus et al., 2002; Valverde & Solé, 2007). This community structure implies that the discovery, specification, and validation facets of RE occur at multiple layers and degrees of intensity across the community and that specific arenas of knowledge exchange, involving specific arrangements of social and electronic communications, are likely to serve specific RE tasks within each facet. However, past OSS studies have primarily focused on static features of communities and have not investigated salient social processes and the deployment of artifacts that enable knowledge exchanges and transform requirements knowledge between different layers and functions of the community.

Recent popular models of software development downplay the earlier emphasis on the formal aspects of RE. However, because of the nature of design, the fundamental goals of discovery, specification, and validation need to be honored regardless of the development approach adopted (Cao & Ramesh, 2008). As we argue below, the resolution of these questions is conditioned by how the selected approach frames and solves a set of cognitive challenges related to RE as a collaborative design effort. For example, structured requirements approaches rely on specification documents and formal models—i.e., explicit and relatively closed forms of externalized RE knowledge. The use of these artifacts, however, has been rendered increasingly problematic in development environments which are highly complex, dynamic, and heterogeneous, while also demanding increasingly dynamic ways of expressing and sharing diverse requirements knowledge (Jarke et al. 2011). Therefore, both agile and OSS approaches tend to disavow formal documentation and deny the presence of a clearly demarcated requirements phase. In contrast, these approaches see requirements discovery and validation as dynamic and interwoven social and technical practices and rely heavily on social mechanisms in expressing and sharing RE knowledge (Petersen & Wohlin, 2010). In the OSS context, the process grows still more complicated due to the lack of control over volunteer developers, who are physically and temporally distributed. This renders face-to-face communications infrequent and shallow (see Appendix A for a comparison of RE principles in the three methodologies) and raises the question of how RE knowledge is expressed and shared in such settings. Because of these differences with OSS, extant RE research provides only limited explanations regarding how requirements knowledge is effectively managed during OSS development (Jarke et al., 2011).

One fruitful avenue for investigating OSS RE is to explore the role and function of heterogeneous artifacts and related mechanisms through which requirements knowledge is created, maintained and disseminated within the OSS community for discovery, specification, and validation. Scacchi (2002, 2009) observed that OSS communities eschew explicit requirements statements in favor of "informalisms" buried in threaded discussion forums, web pages, email communications, and external publications. Recent studies confirm that OSS RE relies on evolving heterogeneous online documentation scattered across diverse representational forms split across multiple channels which record intensive exchanges about varying aspects of the software (Ernst & Murphy, 2012; Noll & Liu, 2010). These studies also show that OSS designs and implementations often result from local improvisation which is rationalized in discovery and specification logic only post hoc, when requirements need to be written down (Ernst & Murphy, 2012). Though this research stream describes in detail how distributed artifacts are used during OSS development, it mainly concentrates on reporting discursive uses of artifacts and their roles in capturing RE knowledge while largely ignoring how the uses of these artifacts trigger, influence, and transform RE knowledge so as to address the three principal RE goals. To put it another way, we do not know how cognitive processes necessary for RE and related to the use of various artifacts by diverse participants unfold within OSS settings. To understand better these forms of RE, we will next formulate a cognitive approach that helps us analyze such processes.

2.2 Cognitive Foundations

Broadly conceived, *cognition* is the human ability to acquire, transform, store, and apply information (knowledge) gathered from the environment (Shettleworth, 2009). Cognition serves as an umbrella term for activities traditionally captured under the concept of general mental functioning: "a moniker for practically all the interesting functions the brain performs to facilitate behavioral adaptations and survival" (Cromwell and Panksepp 2011, p. 2027). Accordingly, cognitive activity includes such diverse functions as attention, evaluation, memory,

communication, decision-making, and problem-solving (Anderson, 2005; Metzler & Shea, 2011). Within the field of cognitive science, cognition has been specifically framed through the analogy of *computation*—the creation and manipulation of symbol systems (Simon, 1980)—to the extent that computation has been identified as "the principle metaphor of cognitive science" (Hutchins, 1995, p. 49).

The distributed and sociotechnical nature of OSS calls for an analysis of cognition in the presence of heterogeneous actors and artifacts. Accordingly, we conceive of RE as a form of *sociotechnical computation*. This lens identifies not only the distinct and mutually supportive roles of participating social actors during cognition, but also their reliance upon artifacts during cognition. The idea of RE as sociotechnical computation does not, however, imply that the processes through which the design environment moves from problem formulation to solution generation are algorithmic and deterministic; rather, the use of the term accentuates the interplay between actors and artifacts that jointly promote cognition around requirements.

The rich literature around sociotechnical cognition presents several candidates for a cognitive approach to OSS RE, including activity theory (Leont'ev, 1981; Vygotsky, 1987), transactive memory systems (Wegner, 1995), actor-network theory (Callon, 1986; Latour, 1987), situated action (Lave, 1988; Suchman, 1987), and DCog theory (Hollan, Hutchins, & Kirsh, 2000; Hutchins, 1995). For our purposes, DCog offers the most appropriate foundation, because it simultaneously considers the social (actor-related), structural (artifact-related), and temporal (dynamism) aspects of cognitive activities, all of which are critical in analyzing how each of the RE facets are "computed." Other theories focus primarily on one side of the sociotechnical divide or on comparatively narrow aspects of cognition. For example, activity theory focuses on a single actor as the unit of analysis with less consideration of interplay between diverse actors and the degree to which artifacts actually bear a cognitive load in information processing. Similarly, while transactive memory systems theory attends to the distribution of cognitive effort among the members of a team, it provides limited consideration of the impacts of artifacts on cognition. In addition, transactive memory systems theory centers on a single cognitive function, namely memory. Actor-network theory explains why sociotechnical knowledge-based networks emerge and achieve stability, particularly in the domains of scientific or technological innovation. However, actor-network theorists are less focused on how such systems process information and execute cognitive functions on a day-to-day basis; rather their focus is on how knowledge is grounded in sociotechnical arrangements (Fomin & De Vaujany, 2008). Finally, while situated action (Lave, 1988; Suchman, 1987) places a significant emphasis on the influence of the external world on human action, the approach downplays the importance of distinct cognitive objectives or goals in favor of contingent and improvisatory responses to changing situations (Nardi, 1996). A summary comparison of reviewed sociotechnical theoretical approaches is provided in Table 1.

Table 1. Summary Comparison of Multiple Sociotechnical Theoretical Approaches

Theory/ framework	Areas of focus	Selected citations	Relative to distributed cognition	
			Points of commonality	Points of contrast
Activity theory	Understanding human activity as a by-product of sociotechnical systems	Leont'ev, 1981; Vygotsky, 1987	Emphasizes the role of artifacts in human activity; focuses on the fusion of external and internal influences	Artifacts merely *mediate* human cognition; focus on cognitive efforts of individuals
Actor-network theory	Explaining formation and maintenance of innovation networks, esp. in the sciences	Callon, 1986; Latour, 1987	Significant concern for nonhuman elements (actants) in a system; knowledge as a product of sociotechnical operations	Limited concern for day-to-day cognition among actors; less focus on how knowledge is used and deployed
Situated action	Understanding the influence of social and material contexts on human actions	Lave, 1988; Suchman, 1987	Emphasis on the interaction of human beings with the social and material environment	Greater concern for contingent response to the external word; less focus on distinct cognitive goals
Transactive memory systems	Exploring how groups capture and store knowledge collectively	Wegner, 1995	Focuses on social distribution of cognitive load	Little concern for the specific role of artifacts in cognition; primarily concerned with memory functions

2.3 Distributed Cognition

DCog theory considers how cognitive tasks are executed through the interaction of distributed actors and artifacts (Hutchins, 1995; Hutchins & Klausen, 1996). Accordingly, it offers a fruitful lens for analyzing OSS RE as it accommodates the roles of heterogeneous actors and artifacts in pursuing discovery, specification, and validation of requirements. While maintaining the metaphorical framing of cognition as computation, DCog theory breaks with traditional cognitive science by asserting that cognitive processes are not bounded by an individual's mind (Rogers & Ellis, 1994). Not only does cognition extend beyond the brain to other parts of the body (e.g., our hands as we write), but also to other humans and the physical environment (Hutchins, 1995). Hence, during cognition, relevant information gets distributed and transformed across actors and physical artifacts over time and space.

Given the shift in scope, DCog reformulates cognition as "the propagation of representational states across representational media" (Hutchins 1995, p. 118), where 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). In the context of OSS RE, representational states would cover diverse ways in which requirements knowledge is perceived, communicated, refined, and instantiated in a software artifact. This framing embraces both internal (e.g., an individual's mental states) and external representations (e.g., embodied artifacts) (Rogers & Ellis, 1994). As these representational states change over time and across media, information propagates through the cognitive system. In the context of OSS, the cognitive system is all actors and artifacts that perceive, store, transform, and transfer requirements knowledge.

Specifically, DCog theory identifies three intertwined modes of cognitive distribution (Hansen et al. 2012; Hollan et al. 2000): (1) social distribution, (2) structural distribution, and (3) temporal distribution. *Social distribution* spreads cognitive workload across members of a group and reflects the cognitive processing resulting from the interactions of multiple actors with diverse skills and expertise. *Structural distribution* allocates cognitive workload through the use of external artifacts—representational forms which store, display, and process information (Hollan et al., 2000). Such artifacts afford specific ways to shoulder cognitive activities such as memory, visualization, or inference. OSS projects use artifacts for RE such as document repositories or recording histories of design discussion (memory), layout, and wireframes (visualization), etc. Finally, *temporal distribution* spreads cognitive workload over time,

where preceding cognitive efforts establish a foundation for the subsequent ones. Several OSS practices, such as code reuse and patterns, manifest temporal distribution (Raymond, 1999). These modes of distribution provide a valuable lens for comparing development models (Hansen, Lyytinen, & Kharabe, 2015). For example, while comparing waterfall and agile approaches, one observes substantial differences in the mechanisms of social distribution (e.g., strict specialization of work vs. close interpersonal interaction), structural distribution (e.g., emphasis on formal documentation vs. informal mock-ups and rapid prototyping), and temporal distribution (e.g., sequential ordering of states vs. intense iteration between) of cognitive effort. In general, the waterfall approach places priority on structural and sequential temporal forms of distributions, while the agile approach emphasizes social and recursive temporal forms of distribution.

Because DCog approaches cognition as a form of computation, it also places significant emphasis on the impact of heuristics—"rules of thumb"—that convey how (and under what conditions) the propagation between representational states happens. Heuristics embody rough "algorithms" that represent the content and nature of computation. Heuristics often combine multiple cognitive functions, including evaluation (e.g., establishing thresholds/acceptance criteria), memory (e.g., reducing the elements to be maintained in working memory), decision-making (e.g., providing tacit decision rules), and problem-solving (e.g., conducting trouble-shooting through causal attribution). Overall, heuristics embody higher-order knowledge within a given domain—"discipline-appropriate problem-solving strategies and patterns of justification, explanation, and inquiry" (Perkins 1993, p. 101). Ultimately, they encapsulate available inferential knowledge regarding the "what, when, and how" of computation. Akin to design patterns (Mangalaraj, Nerur, Mahapatra, & Price, 2014), heuristics can be combined into larger cognitive *patterns* (Vidgen & Wang, 2009). Building upon these foundations the present study will pursue the following research questions:

RQ1: In what ways is requirements-oriented cognition distributed across actors and artifacts in OSS development?

RQ2: What are the specific mechanisms through which OSS requirements are dynamically computed?

RQ3: How do the system-level dynamics between actors and artifacts unfold during requirements "computation"?

3 Research Design

3.1 Project Selection

Given the novelty of using a cognitive approach to study OSS RE our study is exploratory and motivated by the need to identify critical actors and artifacts and the constellations involved in OSS RE "computation." Accordingly, we wanted to sample a representative case to achieve a holistic and generalizable understanding of the phenomenon (Eisenhardt, 1989; Yin, 2009). The criteria for selecting a representative case embodied Lee, Kim, & Gupta's (2009) criteria of OSS success which encompasses both software and community service quality: (1) the project should be of reasonable size (i.e., greater than one million lines of code) to be representative of a population of successful OSS projects; (2) the project should have a relatively large developer community (i.e., greater than 100 developers), so as to be representative of more widely distributed RE processes; and (3) the project should have had at least one official release, indicating initial success in deriving and managing requirements. Based on these criteria, we selected the Rubinius project, which has over 2.6 million lines of code, more than 100 contributors, and had its first release in late 2010 followed by multiple successive releases. Our data collection focused on the period between spring 2010 (when Rubinius 1.0 was launched) and fall 2012 (when the Rubinius 2.0 preview was launched). This period involved a new round of efforts to prepare for the Rubinius 2.0 release and involved significant extension in software functionality. A detailed description of the Rubinius project is provided in Appendix B.

3.2 Data Collection

We drew on multiple sources of data which enabled data triangulation and use of complementary evidence (Yin, 2009). Our primary data collection method was interviewing aimed at investigating how project participants understand and work with design requirements. During interviews, we followed a critical incident approach by asking respondents to describe specific cases where a requirement was successfully implemented and cases where it was not. We then sought to "follow" requirements computation by tracing the movement of knowledge through the system in all cases. To support a comprehensive and systematic data collection, we developed an open-ended interview protocol prior to data collection (Eisenhardt, 1989; Yin, 2009). The protocol was refined and augmented as the study progressed. The final protocol is provided in Appendix C.

In our sampling, we followed theoretical and snowballing techniques (Corbin & Strauss, 2008). To

help in sampling salient respondents, an initial interview was conducted with the founder of Rubinius. He helped us identify other relevant respondents and facilitated access to them. Our criterion in sampling was to cover all key stakeholder groups in the project, including core team members, peripheral developers, participants with different technical skills or roles (e.g., encoding), and representatives of the sponsor. A total of 17 interviews were conducted over a six-month period from September 2012 to February 2013, and 13 were with members of the Rubinius community. We interviewed all three core members, who contributed to 2.2 million lines of code, accounting for approximately 85% of the total lines of code by the end of 2012. These three subjects were interviewed several times to validate and deepen our understanding of the process. In addition, we conducted an interview with a senior vice president at Engine Yard, the primary organizational sponsor of Rubinius. We stopped seeking more interviewees when theoretical and empirical saturation was achieved (i.e., we ceased to gain additional insights from the interviews and the empirical cases could be accounted for by the emerging theory).

The interview protocol was emailed to all respondents prior to the interview so that they could prepare for the interview by recalling critical incidents. Interviews lasted between 60 and 90 minutes and covered the key elements of the protocol, with additional probing when salient issues were identified. Because of the distributed nature of the community, most interviews were conducted via videoconferencing. Due to one respondent's linguistic limitations (e.g., lack of comfort with speaking English), one interview was conducted through instant messaging over Skype. All other interviews were audio recorded with the permission of the respondents. The recordings were transcribed and follow-up emails were occasionally used for clarification. Overall the interview corpus covered 393 pages of transcribed text, containing more than 140,000 words.

In addition to interviews, we collected archival data from the project's GitHub repository. This data represented a heterogeneous mix of project documentation, including project descriptions, 26 blog posts, 1082 instances of stated issues, and 604 instances of pull requests. We also collected direct conversations between project stakeholders archived in the project's mailing list (35 email threads). To manage the overwhelming size of this dataset, a set of emic (Headland, Pike, & Harris, 1990) keywords were elicited from the interviews (see Appendix D for details on this procedure). We then searched the archival data sources for these keywords, and selected all entries that matched. This helped us pare down the archival data to a manageable slice of relevant data. Internet relay chat (IRC) messages, numbered in the hundreds of thousands, were selected when an interview or archival item explicitly pointed us toward them. Analysis of the archival data sources helped validate, refine, and triangulate the data and insights generated through interviews. A summary of data collection activities and data types is provided in Table 2.

Table 2. Summary of Data Collection

Data sources	Descriptions
Interviews	A total of 17 interviews conducted with 393 pages of transcribed data, composed of 16 interviews with 13 committers within the Rubinius community (including multiple interviews with the three core developers) and one interview with a senior vice president at Engine Yard Company
Archival data	1082 issues and 604 pull requests, 26 blog posts, and 35 threads from the developer mailing list

3.3 Data Analysis

In line with grounded theory methodology (Strauss and Corbin 1990), we began data analysis while collecting data. We analyzed the interview transcripts using open, axial, and selective coding (Strauss & Corbin, 1990). As the coding process evolved, we triangulated our initial findings in light of a number of extant theories that focused on the interplay of individuals and artifacts within the cognitive effort of teams. These included the theories summarized in Section 2.2 above. While we were careful not to force the theoretical structures onto our data set, the principles of DCog theory emerged as a critical sensitizing device (Corbin & Strauss, 2008; Eisenhardt, 1989), suggesting a strong fit with our observations in the Rubinius project. In all phases of data analysis, we used Dedoose analysis software (http://www.dedoose.com) to code the relevant data sets. The detailed steps of the coding are presented in Appendix E.

4 Findings

We delineate our key findings in four steps. First, we provide three illustrative vignettes of how several features implemented in various Rubinius releases emerged through RE processes. We will detail the implementation of: concurrency (a metarequirement with regard to parallel processing), flip-flops (a sophisticated logic operator), and

profiler/benchmarking functionalities (tools for identifying performance gaps). These vignettes are intended to offer a sense of the dynamism involved in requirements computation. Second, we explore how requirements computation is socially and structurally distributed in Rubinius across all observed RE activities. Third, we show how computation is temporally distributed through trends, patterns, and heuristics across all observed implemented features. Fourth, we synthesize these analyses into a generalized OSS RE model which distinguishes three broad classes of recursively organized cognitive processes—excavation, instantiation, and testing-in-the-wild—which we put forward as distinct stages of requirements computation within OSS.

4.1 Three Vignettes Illustrating Requirement Engineering Computation

The vignettes that follow are emblematic of the ways in which requirements knowledge is discovered, specified, and validated in Rubinius. We include these here to provide the reader with a firsthand sense of how RE processes unfolded within the Rubinius community. When we consider these vignettes, several salient features about the distribution of cognitive load during RE become visible including distinct dimensions of distribution (i.e., social, structural, and temporal), reliance upon several heuristics to guide actions, and what patterns of propagation across representational states emerged.

Table 3. An Example of RE Computation in Rubinius—Concurrency

Given the increasing prominence of multicore processors, concurrency was acknowledged as a technological change worthy of the Rubinius community's attention. One day, in the midst of regular IRC channel exchanges, a Rubinius committer called attention to the lack of support for multicore programming in Rubinius. This functionality was missing because Matz's Ruby Interpreter (MRI, a standard Ruby interpreter upon which Rubinius is based) did not yet support runtime concurrency even though concurrent I/O had been implemented.

This observation elicited a conversation on IRC about the feasibility of implementing concurrency in Rubinius. Examples of the concerns voiced at that time included "Running concurrency forces Rubinius to suck down GIL [Global Interpreter Lock] timeslices." Based on this exchange, the core committers decided to "give it a try" by attempting to develop this new feature. The main objective was to improve Rubinius's runtime performance. One of the core committers tentatively began to develop the feature. During the process, the general idea (runtime concurrency requirements), which initially had surfaced during IRC conversations, was translated into a series of informal specs and corresponding code. After several days of exploratory development, the core committers concluded that runtime concurrency might indeed be a feasible feature.

Accordingly, they established a new branch on GitHub, dubbed Hydra, to pursue this work. This decision kicked off another round of trial-and-error experimentation recorded under the new branch and characterized as "experiments with concurrent allocation and full stop GC [Garbage Collection]." After several months of experimentation, one of the core committers published a post on the Rubinius website (http://rubini.us/2011/02/17/rubinius-what-s-next/) announcing that, "I'd like to introduce the work we are doing on the Hydra branch and the features you can expect to see in Rubinius soon." By then, the Hydra code had proven to be relatively stable so that runtime concurrency could be effectively demonstrated.

Next, a baseline implementation was formed where the core committers merged the then-current master branch with Hydra. This indicated that the new feature was ready for the wider community to "play with." Several individuals and related communities began to test their own software projects that would need such a feature to determine if the newly implemented functionality would serve their needs. The developers also ran the baseline implementation against MRI to see whether it worked as expected. During this process, the stated requirement of runtime concurrency was validated with the support of multiple artifacts such as MRI, GitHub, Puma (a web server), and Travis (a continuous integration server). When tests failed or the baseline implementation crashed, the developers were encouraged to identify new bugs and report them via GitHub in the form of issues and pull requests. For example, one developer raised an issue concerning "Randomizer in Hydra segfaults" when running the Hydra branch under high load (https://github.com/rubinius/rubinius/issues/726). One of the core committers addressed this issue by "adding spinlock around state in Randomizer." However, resolving the issue triggered another problem (e.g., fault in Inline Cache). Therefore an issue (https://github.com/rubinius/rubinius/issues/729) was later opened signaling the creation of new requirements to be considered.

The first vignette focuses on the emergence and evolution of a requirement around concurrency of the running code. Socially, we can see the activity of distinct sets of social actors, such as core committers, peripheral developers, and external communities, who play different roles with respect to the discovery, specification, and validation of the concurrency requirement. For example, the idea originally emerged from a peripheral committer and was then discussed extensively within the core through the IRC channel. The idea was subsequently taken up by a core committer who proposed a tentative solution which was then shared with a broader group of core committers, resulting in a spec. Structurally, several artifacts are prominent in the computation process, including the GitHub platform, IRC channels, informal specs, and related software platforms. These were critical in storing, displaying, and distributing knowledge or generating new knowledge through experimenting around focal requirements. As with the social actors, the prominence of these artifacts varied widely at different stages of the process. For example, specs were important in the early stages but not relied upon in the later stages. Finally, temporal distribution shows how efforts build upon one another. For example, the processing of the idea moved into an IRC channel, then into tentatively implemented code, then into an informal spec, and so on. These steps show how the whole Rubinius ecosystem moves from initial problem identification toward resolution and ultimate feature use. Interestingly, the community employed fairly simple behavioral norms (i.e., heuristics) to foster the evolution of the desired functionality over time. For example, one core committer volunteers to "give it a try," reports his results in an informal spec, which is then expanded to a fuller, potentially promising implementation, providing the justification of a new branch in the

GitHub repository. Overall, we see a trace of mappings from one representational state to another where the concurrency feature was expressed as an idea, as a set of recorded notes in the IRC channel, as a set of tentative software solutions, as informal specs, and so on.

Table 4. An Example of RE Computation in Rubinius—Flip-Flops

A Rubinius committer, who wanted to "try something new and crazy," opened an issue in GitHub related to implementing a flip-flops feature, which already existed in MRI. However, this issue was closed by one of the core committers because flip-flops were deemed "esoteric and unnecessary." The core committer refused to regard flip-flops implementation as a new requirement until the committer could show him real-world code that used it. The committer therefore examined the MRI source code and asked the larger community of Ruby developers whether anyone was using flip-flops. Fortunately, a Ruby developer came up with "the weirdest code example" that illustrated a valuable use case. Given that flip-flops behavior could be found in MRI and a real-world use-case existed, the core committer conceded and agreed to include the flip-flops feature in Rubinius. Accordingly, the committer began to work on the flip-flops feature as a trial-and-error experiment under a new branch in GitHub, eponymously named flip-flop. Due to insufficient tests for flip-flops, the committer "searched through RSpec [a specification for describing the behavior of MRI]" and communicated with core committers from time to time "on IRC mainly" to write and adjust specs and code until they passed on MRI and Rubinius. The committer "went into wherever the compiler was throwing errors and just added the code necessary." Although the core committers did not take charge of the actual coding process, they always guided the committer and were involved in writing some specs and code. For instance, a core committer commented regarding the issue "sexp_key should be sexp_name to be consistent with existing code" and another committer commented "This isn't thread safe at all . . . Flipflops should just be implemented using stack locals . . ." (https://github.com/rubinius/rubinius/pull/1257). Once all specs and code passed, the related commits were merged into the master branch, which signaled to the wider community that the feature was ready to be validated.

In the second vignette, we can again see social, structural, and temporal distribution. Socially, core committers, peripheral committers, and external communities are all involved in the computation of the flip-flops requirement. Flip-flops—sophisticated logic operators allowing for multiple truth conditions—already existed in MRI. Therefore, a peripheral committer proposed their implementation in Rubinius, but the proposal was turned down by a core committer on the grounds that it was "esoteric and unnecessary" from a cost-benefit perspective. Later on, the functionality was supported by a real-world use-case submitted by a Ruby developer in an external community, which convinced the core committer of the usefulness of the functionality; therefore, the functionality was eventually accepted for inclusion in Rubinius release. Structurally, we can see the importance of GitHub, MRI, Ruby-related projects, RSpec, and the IRC channel in perceiving and evaluating knowledge and generating a solution. The existence of flip-flops in MRI triggered the peripheral committer to raise the issue of flip-flops in GitHub's Rubinius repository, enabling the core committer to evaluate its feasibility. As real-world code from Ruby-related projects emerged, the functionality was reevaluated by the core committer, and the implementation could be moved forward by the peripheral committer. Afterwards, the knowledge of the flip-flops requirement flowed from RSpec, which directed the peripheral committer to write specs describing the desired flip-flops behavior of Rubinius. The peripheral committer first turned to MRI which provided a baseline for the functionality's intended behavior and then to the IRC channel which facilitated problem solving. Temporally, we can observe how subsequent efforts rely on preceding efforts, from functional discovery to validation. For example, the processing of the function was translated into a bug report on GitHub, which was then transformed into specs and code, and eventually transformed into changes to the master branch.

Table 5. An Example of RE Computation in Rubinius—Profiler/Benchmarking

As an interpreted (as opposed to compiled) programming language, Ruby has often been accused of being slow. Rubinius, being written largely in Ruby, as opposed to C (in which MRI is written), has also received its fair share of accusations in this regard, but the community largely considered it to be its mission to overcome such performance constraints ("the hope that Rubinius actually would be faster"). To realize this hope, Rubinius core developers launched an initiative for "improving the JIT compiler." A particular feature requested to help realize this vision was to implement benchmarks for specific methods (e.g. "Added a benchmark to allow performance comparison of Array#permutation against other Ruby implementations"), as well as an improved profiler which allows for very detailed profiling, i.e. "type profiling," which in turn enables developers to understand performance in relation to specific object types and methods.

The overall process of identifying performance gaps relies on the involvement of the community, leading the core developers to essentially make a call for contributions: "Rubinius is calling on the über programmers of the world to implement solutions in Ruby to help us identify performance challenges and address them." Through this call, Rubinius could harness the capacity of "oceans of regulars" to look into the various nooks and crannies of the codebase, using the profiler and the associated benchmarks, so as to find places where the performance of Rubinius could be improved.

In the last vignette, we see how a persistent need for speed generates the implementation of a set of tools—a profiler and associated benchmarks for improving the just-in-time (JIT) compiler. In this vignette, significant social distribution helps gather a large number of benchmarks. These benchmarks then constitute a widened structural distribution of artifacts that allows developers in multiple situations to compare their code to the benchmarks. Temporally, the core initiates the development of a tool (i.e., the profiler), so as to activate the broader social distribution, which in turn generates and activates a broader benchmark-oriented structural distribution, thus sequencing the various forms of distribution across time, so that the work performed through structural distribution builds upon work performed first by the core and later a broader community of actors.

In summary, these vignettes illustrate the rich dynamics across the social, structural, and temporal forms of cognitive distribution. Next, we consider generic social and structural elements present in Rubinius RE as well as discuss how they are tied together by temporal distribution mechanisms.

4.2 Social and Structural Distribution Mechanisms

4.2.1 Social Distribution

Overall, social distribution is concerned with allocating cognitive workload across members of a group of actors involved in RE computation. Table 6 shows the social distribution mechanisms in Rubinius, with an emphasis on the distinct analytical categories and actor types observed. These categories classify actors according to their relative position within the Rubinius project and the cognitive activities in which they participated. These include: (1) the focal community (i.e., actors internal to the project), (2) sponsors (i.e., actors neither entirely internal nor external to the project), and (3) external communities (i.e., actors external to the project). Because of Rubinius's open commit policy (i.e., anyone who gets a pull request accepted is automatically granted commit rights to the repository), we can distinguish between internal and external participation based on whether or not actors have a commit access to the project. 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.

Each category supports different cognitive functions in provisioning and manipulating requirements knowledge. These functions reflect the necessary distribution of the cognitive workload across social and organizational boundaries. Based on these functions, the focal community can devote their software knowledge and technical expertise to RE activities with the benefit of sponsor input. External communities, in contrast, provide mainly an impetus for the discovery of new requirements. Though the Rubinius community experienced high member turnover, the roles related to requirements computation and related knowledge flow remained relatively stable across the study period. Thus, the skills of the actors in specific social positions (see Table 6) largely determined how the requirements knowledge became socially sourced and distributed.

Table 6. Summary of Social Distribution Mechanisms

Actors	Descriptions	Functions in cognitive system
Focal community		
Core committers	Actors engaging in the daily management of the project; responsible for guiding the overall project direction and coordinating development activities; generally reflecting extended involvement on the project and regular contribution of new features	Providing domain and content-specific knowledge Problem solving and decision-making
Peripheral committers	Actors primarily contributing to discovering, reporting, and/or fixing bugs; occasionally contributing new features, with sporadic, periodic, or seasonal involvement	Providing content-specific knowledge Perception and communications of issues/problems
Sponsor		
Engine Yard	Platform-as-a-Service (PaaS) company focusing on Ruby on Rails and PHP development and management; providing financial support to several Rubinius committers	Providing domain knowledge Perception and communications of needs
External communities		
Ruby developers/ users	Actors working on or using the Ruby programming language	Providing use-cases and technical skills Perception of needs Evaluation of functionality
Ruby on Rails developers/ users	Actors working on or using Rails, a web application framework for the Ruby programming language	
Engine Yard customers	Users of Engine Yard services	
Puma users	Users of Puma, a concurrent HTTP 1.1 server for Ruby web applications	
Travis users	Users of Travis, a hosted continuous integration service for open source projects	

Within the *focal community*, RE relied heavily on the emerging vision of core committers. This vision was strongly rooted in the knowledge gained in developing Rubinius 1.0, including both domain knowledge (e.g., regarding the Ruby environment) and content-specific knowledge (e.g., regarding certain Rubinius components such as virtual machines, benchmarks, and standard specifications). Apart from a small number of core committers, hundreds of peripheral committers carried out specific aspects of requirements discovery. For example, a peripheral committer, who previously used an external "*gem*" (i.e., a package) for a specific purpose, submitted a pull request to make said feature part of Rubinius: "I use FFI::Pointer.size in a gem to check if I'm on a 32-bit or 64-bit platform, and I imagine I'm not the only one doing that . . . I hope this is a good addition to Rubinius." Their RE activities focused on discovering and settling requirements questions by drawing on their diverse expertise and interests. The following two statements illustrate the social dynamic of the focal community:

> *[One of the core committers] has like most of the interknowledge because he wrote most of Rubinius. His hunches of what [a problem] could be usually come quicker. That's the biggest difference, as you probably get to know the whole code base and how it works in these cases.*

> *Most people did have some kind of narrower focus . . . There were some people who worked just on the VM or the code generation, but a lot of the contributors worked solely on Ruby, solely on the specs, or just implementing standard functionality.*

The distribution of requirements knowledge was not limited to the focal community. It was complemented

with the support of Engine Yard, which had a strong emotional tie to OSS in general and the Ruby programming language, in particular. They were culturally embedded within the larger OSS community and "wanted Ruby to win," i.e., become an established programming language. During the study period (i.e., 2010 to 2012), Engine Yard representatives did not propose any formal functional requirements for Rubinius; they respected the will of the Rubinius community. Rather, they provided useful advice to the community on how to specify and implement requirements. In addition, Engine Yard facilitated Rubinius's exposure to other external communities by providing Rubinius as an option for their clients on the Engine Yard platform. In this regard, Engine Yard's sponsorship helped Rubinius build relationships with external communities and thereby increase the credibility of the RE process and validity of the requirements.

Indeed, many of the salient requirements flowed from external communities including GitHub, the Engine Yard platform, Puma, and Travis CI (Continuous Integration). *External communities* refer here to a variety of Ruby-related users and developers, who contributed requirements knowledge to Rubinius based on their domain or technical skills. These related communities provided use-cases which helped make certain requirements explicit, public, and highly specific. A committer's comment provides a useful illustration of this role:

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

4.2.2 Structural Distribution

Structural distribution refers to the distribution of cognitive workload through the presence and use of external (representational) artifacts. In the case of Rubinius, nearly all notable artifacts were digital in nature due to the significant geographic distribution of the work and extensive use of software development platforms (GitHub)[1]. Table 7 provides a summary of structural distribution mechanisms in Rubinius. Four categories of structural artifacts—i.e., web resources, system artifacts, communication channels, and environments—were identified based on characteristics of the medium and the functions that they served in propagating requirements knowledge forward. Importantly, this suggests that these observed forms of distribution do not simply reflect the current state of the requirements knowledge at a certain point; rather, these artifacts are iteratively and recursively mobilized and modified as requirements knowledge evolves.

[1] While analog artifacts (e.g., notepad sketches, whiteboards) may be used by individual Rubinius developers, these could not be "verified" empirically, given the nature of the study.

Table 7. Summary of Structural Distribution Categories and Artifacts

Artifacts	Descriptions	Functions in cognitive system
Web Resources		
Rubinius website	A website for introducing, documenting, and blogging about the Rubinius project	Directing perception of focal community members Establishing parameters or constraints on the solution space Creating external memory functions
Rspec	A specification for describing the expected behavior of the Ruby programming language	
RDocs	Documentation generated for the Ruby programming language	
RubySpec	An executable specification for the Ruby programming language that is syntax-compatible with RSpec	
System artifacts		
GitHub	A web-based hosting service for software development projects	Automation of computational processes Creating external memory function Enabling evaluation of proposed or instantiated functionality Directing perception of focal and external communities
Engine Yard platform	A cloud Platform-as-a-Service (PaaS) for Ruby on Rails, PHP and Node.js applications	
Puma webserver	A concurrent HTTP 1.1 server for Ruby web applications	
Travis CI	A hosted continuous integration service for OSS projects	
MRI	A standard Ruby interpreter	
Tools	Git, A debugger, Gdb, Insiter, etc. (different developers used different tools with respect to personal preferences)	
Ruby-related projects	Projects testing on/against Rubinius	
Communication channels		
Mailing list	Email lists for community discussions	Enabling communication between social actors Facilitating decision-making Supporting perception of hidden or latent requirements Creating external memory functions
Skype	Internet telephony and video teleconference	
IRC channel	Internet relay chat discussion forums	
Phone call	Traditional telephony	
Environments		
Hardware	The environment of contemporary hardware architectures (e.g., multicore processors) to which Rubinius must relate	Enabling perception of broader technological capabilities Facilitating decision-making
Software	The environment of contemporary software architectures (e.g., real-time web interfaces) to which Rubinius must relate	

Web resources consists of the project's web pages. These served as sources of input or output to requirements computation, by rendering requirements knowledge in explicit (textual and graphical) form. Primarily, these resources supported discovery of relevant requirements knowledge in that they helped

respective actors produce and then perceive and evaluate salient issues in the expected functioning of Rubinius and the Ruby programming language. Further, they set the baseline goals and criteria for the subsequent testing of identified requirements. The role of RubySpec[2]—a key web resource—provides a useful illustration:

> *RubySpec is an executable specification part of Ruby language libraries. It is what Rubinius tests itself against and it maintains a set of tags which basically say, "This particular RubySpec fails on Rubinius." So this is a point that Rubinius needs to fix—to find out either where the problem is or [determine] if it is something that hasn't been implemented yet.*

System artifacts are software tools that automate certain computational processes related to requirements. In particular, the GitHub platform serves as Rubinius's organizational memory by recording what, how, and when each piece of code was created and by whom. Accordingly, much of the activity around requirements discovery, specification, and validation takes place on GitHub by tracing the ongoing commentaries and related knowledge provisioning associated with pull requests and issues reports. The platform also offers a way of recording and maintaining "who knows what" knowledge. As a committer noted: "GitHub just made it easy for people to jump in and contribute, and for us to handle the code." MRI, the Engine Yard platform, Puma webserver, Travis CI, and other Ruby-related projects also served as significant sources for requirements discovery and management from external communities. These system artifacts played a critical computational role in requirements identification, as their functionality (e.g., continuous integration, automated testing) simplified the information processing demanded of the developers. For example, with the use of continuous integration, the burden of assessing the status of the build was transformed from a cognitively intensive task (e.g., continuous review, manual testing) to a comparatively simple task of observing a status indicator. For the pull request "Running spec/ruby/core for 1.8 fails," Travisbot (an automated "bot" used by the Travis CI system) would leave a comment indicating: "This pull request passes," while in the pull request "Completely fix deadlocks of Thread#raise," Travisbot would leave a comment among discussions indicating: "This pull request fails."

2 RubySpec is a collection of executable specification documents available on the web for the Ruby programming language. It describes Ruby language syntax and standard library classes. A detailed description of RubySpec is provided in Appendix B.

Communication channels refer to various (mostly electronic) media through which project-related information gets communicated between developers and other stakeholders. They played a pivotal role in enabling requirements knowledge to flow across boundaries and revealing hidden or latent requirements, thereby making them exchangeable with other members of the community. The most prominent communication channel in Rubinius was IRC ("Have you been to the IRC channel? Things are actually discussed there"), which served several functions in the cognitive system including facilitating problem perception, enabling solution generation, promoting decision making, and capturing the history of requirements-oriented discussions (i.e., memory). The following two quotes illustrate these roles:

> *We were talking about [the concurrency stuff] on the IRC Channel one day and it got me thinking and I was like "I'll just do it as a spike and see how far I can get."*

> *We did coordinate pretty much exclusively on the IRC channel. . . . The actual discussions mostly took place on IRC, and just told people what we were doing or going to be doing. If somebody needed help or assistance in implementing something, then they notified there and we would come and help.*

Environments represent broader sociotechnical contexts in which the project exists. Specifically, they consist of software and hardware that are relevant in relation to new features being considered for implementation. For example, the emergence of new hardware architectures with multicores established the importance of supporting such hardware features. This external development offloads some of the cognitive burden from the team in making decisions and helping delineate some features as being more relevant than others. As a result, real-time web interface functionality, which could capitalize on the concurrent runtime capabilities of multicore processors, became a promising avenue to pursue. In order to stay abreast of the technical state-of-the-art, Rubinius needs to support such functionalities. Thus, implicit requirements were often embedded within hardware and software architectures that influenced Rubinius's implementation:

> *The concurrency stuff is because of industry . . . It's just the technology. People are building other core CPUs and putting them in everything, so we knew we had to have those features.*

4.3 Temporal Structuring Mechanisms

We observed three temporal structuring mechanisms, through which requirements knowledge evolved over

time: (1) trends, (2) patterns, and (3) heuristics. These mechanisms are hierarchically organized and recursive (i.e., trends incorporate multiple patterns, patterns incorporate multiple heuristics). At the highest level, *trends* represent broad environmental expectations of technology such as the evolution in hardware or software environments or the emergence of nonfunctional requirements (e.g., security standards). By establishing a context for technological advancement, trends influence an OSS project's relationship with its external environment and especially the temporal pacing of requirements discovery, prioritization, and implementation. *Patterns* aggregate salient heuristics to facilitate requirements computation in an orderly manner in order to ensure that all three RE facets are addressed consistently (e.g., creating a shifting focus of work at different stages of release development). [3] Patterns typically operate within the timescale of a release cycle and are influenced by the extent of scoping of requirements within a release. Such patterns in Rubinius are primarily concerned with extracting requirements from root artifacts—which delineate the de facto standard that Rubinius must meet and depend on—and also from distal artifacts, which alert developers to take note of new technical requirements that Rubinius needs to meet. At the lowest level, *heuristics* refer to the "rules of thumb" that simplify cognitively intensive tasks (e.g., "If a behavior can be found in MRI, then replicate it in Rubinius"). They operate on the timescale of computing a specific aspect of an individual requirement. While the role of heuristics is well-established in cognitive theory, the former two temporal structuring mechanisms can be viewed as higher-order structuring mechanisms called "patterns" above. Overall, they provide goal oriented "narrative structures," which enable social actors to identify what sort of "story" or "cognitive play" the current activity represents and which types of heuristics might be relevant in a given situation. Table 8 provides a summary and examples of each temporal mechanism present in the Rubinius project. We also show how each structuring mechanism is embedded in specific elements of the social and structural distributions.

[3] Our concept of "pattern" is distinct from the way the term is used in programming. However, it is consistent with Alexander's (1964) original idea in that patterns reflect generic approaches to problem framing and solution generation which are developed and refined over time while being applied.

Table 8. Summary of Temporal Distribution Mechanisms

Pacing	Mechanism	Descriptions	Social elements	Structural elements
Trends	Scanning for high-level requirements in the technological environment	Identifying broader requirements capitalizing on general technological changes and trajectories	External communities	Environments
Patterns	Uncovering embedded requirements in root and distal artifacts	Requirements embodied within artifacts which are external, but related to Rubinius	Focal community External communities	Web resources System/software artifacts
Heuristics	Detailed guidelines for computing an individual requirement	Rules of thumb guiding individual requirement to be discovered, refined and implemented	Focal community External communities	Web resources System/software artifacts Communication channels

4.3.1 Trends

The degree to which OSS projects follow changes in the broader technological landscape (e.g., multicore processors or the real-time web) influences the search for a new set of features (i.e., requirements) to be integrated into the software. *Scanning for high level requirements in the technological environment* refers to the identification of broad, often implicit, requirements established by changes in the sociotechnical environment (such as security standards). These are envisioned technological capabilities that the project seeks to capitalize upon so as to keep the software "hot." The project team therefore has to transform vague, contested, and weakly articulated trends into explicit requirements that can be instantiated in the software. This process depends on the cognitive function of perception, as it helps frame which new features in the technology environment are most relevant. Recalling the

multicore processing example, the emergence of the multicore hardware architectures and supporting operating systems necessitated adding new features to the Ruby language to provide support of multiple run time threads. Indeed, this was one of the original motivations for releasing Rubinius 2.0—to make Ruby multithreaded. Hence, a new hardware capability triggered the initiation of the development process through the discovery of new requirements:

> *Every CPU going into almost anything these days has more than one core, so being able to utilize resources efficiently requires being sensitive to memory pressure. If you're using five instances of a process instead of one process using five threads, you're not gonna be efficient. So it was sort of a no-brainer in terms of concurrency as something we always intended to do. All those steps up to now—improving the architecture in the C++ and VM—helped lay the groundwork for doing the concurrency work.*

4.3.2 Patterns

Patterns provide a consistent framing for how the distributed cognitive system will solve a set of similar problems typically associated with an RE facet. Patterns bundle together a set of heuristics that guide the use of social and structural resources and artifacts at hand. Overall, patterns help organize activities to coordinate requirements computation across and between RE facets. Even though all sorts of RE activities occur regularly throughout a release cycle, OSS developers show a marked tendency toward bundling some of them to occur more frequently in the early stages of the cycle (e.g., discovery), whereas other activities become more prominent in the later stages (e.g., validation). One reason for this is the high interdependency between discovered requirements and the need to use scarce skills and knowledge effectively. Table 9 depicts three patterns associated with early and late stages in the Rubinius release cycle that integrate embedded requirements in root artifacts, replicating MRI behavior, and embedded requirements in distal artifacts.

Table 9. Summary of Patterns

Patterns	Descriptions	Social elements	Structural elements	Illustrative statements
Early Stage				
Uncovering embedded requirements in root artifacts	Requirements embodied within existing artifacts which offer implementation guidelines for the project (i.e. mainly discovery)	Focal community Ruby developers and users	MRI	"MRI is the de facto standard . . . A lot of Rubinius work is actually to reverse engineering what MRI does, figuring out its behavior." Pull request: "are you just working on failures or porting code from MRI? Since we just import this library from MRI, if these bugs are fixed in MRI, we should just update to the 1.8.7 stable version. If they are not fixed, we should be submitting fixes to MRI."
Late Stage				
Replicating MRI behavior	Writing code so that the Rubinius artifacts deliver behaviors encapsulated in specs (i.e. mainly specification)	Core committers	RubySpec	"When you're working on new features, you look at how MRI behaves or read how MRI is supposed to behave, write some specs, check them against MRI to make sure that the documentation is correct. Then you just sort of stare at that and say 'Okay, this is ready to implement,' and you go do it."
Uncovering embedded requirements in distal artifacts	Requirements embodied within existing artifacts which the project intends to be compatible with (i.e. mainly discovery and validation)	External communities	Engine Yard platform Puma webserver Travis CI Ruby-related projects	"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 build of Rubinius. I want to build a weekly release.' They can easily specify their level of engaging Rubinius changes and then just watch and see whether their project passes or fails. . . . They link us to their results on Travis and say 'Here, this is what I get. It's having this error.' And we can figure it out directly from the error output."

Uncovering embedded requirements in root artifacts refers to the process of discovering requirements embodied within existing artifacts. They offer guidelines for the subsequent implementation of new features. In Rubinius, the implementation guidelines of the Ruby programming language (i.e., MRI) were a critical source of embedded requirements. These guidelines reflect prior cognitive effort and learning and thus present developers and users with a shared cognitive foundation and constraints upon which to build their RE process. The norm is that each instantiation of a Ruby runtime environment has to adhere to this standard. Indeed, a large number of Rubinius requirements are established a priori in MRI, with the understanding that MRI forms an essential reference for desired functionality. Requirements from MRI are uncovered early in a release cycle, as they represent the "low-hanging fruit" around which there is a strong consensus. Further, identifying embedded requirements and translating them into testable specs provides valuable knowledge for subsequent development by establishing common points of reference in the community. This embedding of requirements is similar to "best practices" transcending their place of production as they are instantiated in commercial software packages (Pollock & Williams, 2009).

Replicating MRI behavior. As embedded requirements are uncovered and translated into testable specs, it becomes increasingly feasible for the

developers—especially core committers—to shift to writing code (i.e., engaging in design) that addresses those requirements. Hence, the cognitive load is again redistributed, as fewer developers can engage with the increasing technical difficulty (as one of our respondents put it, "*brain melting*" work) of configuring the internal workings of Rubinius to deliver the behaviors as described in specs.

Uncovering embedded requirements in distal artifacts identifies requirements that are "inherited" from other Ruby-related projects, which run Rubinius and test its functionality. These external communities either test their projects directly on Rubinius or on other platforms (e.g., Engine Yard, Puma webserver, Travis CI), which include Rubinius as the runtime environment. The continuous backward flows from test failures highlight new elements that need to be accounted for. Additionally, these embedded requirements provide insights into how requirements knowledge can be transferred across social and structural boundaries.

4.3.3 Heuristics

In Rubinius, heuristics take the form of guidelines for how developers should discover, consolidate, refine, negotiate, and implement a certain requirement. The heuristics combine social and structural resources to simplify specific cognitive tasks associated with requirements discovery, specification, and validation, as well as the overall recursive process which connects the three RE facets in an iterative loop. Heuristics guide developers to coordinate knowledge with specific groups of fellow developers and to manipulate certain artifacts in order to sequence their activities. For example, if the focal community finds a behavior in MRI that is not addressed in Rubinius, it would extract associated knowledge embedded in MRI and transform it into verbal, graphical, and literal representations in Rubinius (i.e., often a "spec"). Apart from MRI, the other Ruby-related projects provide alternative sources for developers to identify missing behaviors that need to be implemented in Rubinius. A set of representative heuristics observed in Rubinius requirements computation are summarized in Table 10. The table also indicates the primary RE facet that the heuristic supports.

Table 10. Summary of Heuristics per RE Facet

Heuristics	Social elements	Structural elements	Interaction of social and structural elements	Illustrative statements
Discovery				
1. If a behavior can be found in MRI, then replicate it in Rubinius	Focal community	MRI	Focal community extracts requirements knowledge embedded in MRI	"For every method [in the MRI source code] I would take the textual description and break it down into as many distinct facets of behavior as I could, and then write a specification for each of those." Pull request: "I've implemented an initial run of building Rubinius into static and shared libraries . . . I was mainly following what MRI does in that case."
2. If a use-case exists in defining MRI behavior, then replicate it in Rubinius	Focal community Sponsor External community	MRI Ruby-related projects	Focal community, sponsor, and external community identifies use-case from Ruby- related projects	"We provide feedback [from real-world use cases] to the Rubinius project and we think it's a big value for any open source project to have real-world use cases that it uses just to figure out if it's actually delivering what it's trying to deliver."
Specification				
3. If a feature is experimental, separate it as a side branch	Core committers	GitHub	Core committers utilize Github as documentary foundation of requirements knowledge	"If [a feature] takes longer than using the local branch or if it's something that we want other people to review . . . then we sometimes use a feature branch."

Table 10. Summary of Heuristics per RE Facet

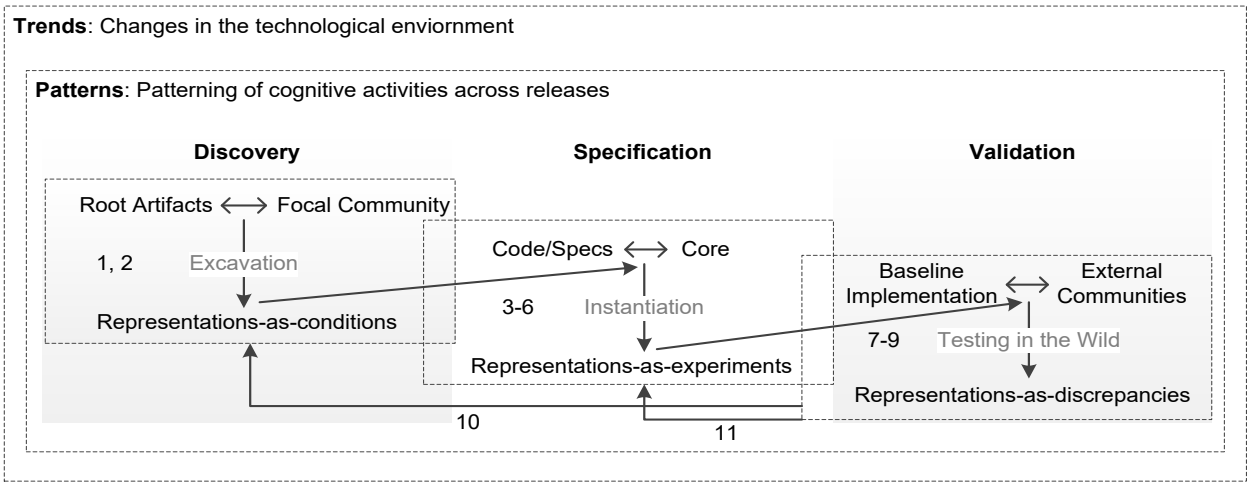
4. If a developer is trying to understand an established feature, then check documentation	Focal community	Web resources (esp. Ruby Spec)	Focal community referred to certain web resources to explore problems and identify solutions	IRC channel: "xxx look in spec/ruby/optional/ffi/string_spec.rb and see if the code you're looking at is in there." Pull request: "In the future, please separate spec/ruby patches from the rest. See http://rubini.us/doc/en/specs/ for the reason why."
5. If a developer is trying to understand an in-development feature, then go to IRC and talk to Core	Focal community External community	Communication channels (esp. IRC)	Focal community and external community discuss in-development requirements via communication channels	IRC channel: yyy: "hey, I want to work on the 1.9 compatibility. What should I attack?" zzz: "depends what you want to work on. small request, do small bits and push frequently. work on the hydra branch."
6. If a spec has been coded and passes the test, mark it as completed	Core committers	RubySpec MRI GitHub	Core committers test Rubinius specs against RubySpec and MRI	"You write the specs. You get them to pass on Ruby 1.9 and then you get them to fail on Rubinius and then you go about implementing them on Rubinius."
Validation				
7. If code passes tests, then distribute the code through the master branch	Core committers	MRI GitHub	Core committers disseminate code baseline of Rubinius, which had been tested against MRI, in Github platform	"Master is something that we really keep very stable, so if you want the best Rubinius version out there, you just grab today's Master."
8. If tests fail or Rubinius crashes, then identify the bug	Focal community External community	Engine Yard platform Related projects GitHub	Focal and external communities use various system artifacts to identify bugs	"When you have an actual user bug, one of the big processes is just trying to isolate it and refine the code into something much smaller—this tiny bit of code that shows the bug."
9. If a bug has been identified, then report it in the bug tracker	Focal community External community	GitHub Travis CI	Focal and external communities identify bugs and convert knowledge into bug tracker held in Github	"If [it] appeared to be a bug or some conflict with the documentation, then we would file a bug report and ask for clarification." Pull request: "I tried to compile rubinius on my archlinux machine and got a warning/error regarding an implicit case of (unsigned int) to (const int32_t). . . . Here is my setup and the exact error."

Table 10. Summary of Heuristics per RE Facet

Recursion				
10. If a bug is related to faulty/missing specs, then go to "Discovery"	Focal community	MRI RubySpec GitHub	Focal community referred to MRI and/or RubySpec to identify faulty/missing specs	"When we hit a bug that we don't have a RubySpec for, we look at the specs, figure out where we're missing all those things." Pull request: "This is a patch to correct the behavior of Class#dup in Rubinius. The patch ensures the proper ancestry for the duplicated singleton class. Included are specs for the correct behavior (met by MRI)."
11. If a bug is related to problematic code, then go to "Specification"	Focal community	MRI RubySpec GitHub	Focal community referred to MRI and/or RubySpec to identify problematic code	"I thought there would be a bug in Rubinius . . . so I went through RubySpec and then go back and fix Rubinius."

4.4 Generalized Computational Structure

Based on our analysis, we can synthesize a generalized framework of OSS RE "computation." The framework is graphically depicted in Figure 1, with an emphasis on how requirements knowledge "flows" through the cognitive system and how the cognitive system as a whole "reconfigures" itself as the computation unfolds. Specifically, the model illustrates how the tripartite functions of RE (discovery, specification, and validation) become enacted in Rubinius through the ongoing reconfiguration of three forms of distribution. This results in different temporary configurations of social and structural distribution of actors, artifacts, and heuristics to perform the cognitive work necessary to satisfy the goals of each RE facet. Notably, although the figure shows the computational structure as a sequential process, it does not imply a "linear" flow of RE activity; rather, the three patterns evolve and interact through threaded iterative loops as different requirements emerge and evolve toward closure, enabling a new release.



Note: The numbers in the figure refer to the application of the 11 heuristics detailed in Table 10.

Figure 1. Computational Framework of RE in OSS

The framework also reveals how requirements, understood as representational states, become propagated through the distributed cognitive system from discovery through validation. It also highlights where the flow is likely to "break," such as when someone drops a requirement by failing to note it in an IRC channel or the system "forgets" a requirement by failing to record it. It also shows where requirements typically "hang up" in the system due to inadequate feedback mechanisms to earlier stages of

each RE process. Such propagation "failures" or "lags" emerge especially when the requirements knowledge needs to travel from one subsystem to another. Next, we delineate in detail the computational processes and the dynamic flow of representational states captured in Figure 1.

4.4.1 Trends and Patterns

The long-term temporal structuring mechanisms reflected in RE computation (i.e., trends and patterns) have an effect on the overall computational system. *Trends* drive the characteristics and directions of the overall RE process. For example, the specific focus on concurrency in Rubinius during the study period provided a cognitive framing through which individual requirements were interpreted and indicated what sort of expertise needed to be mobilized. Those requirements necessary to support runtime concurrency or benefiting from it became favored, while those which did not were pushed to the background. *Patterns* provide a consistent temporal structure for a set of distributed cognitive efforts within the release cycle. For example, identification of requirements from root artifacts had to be pursued early on, whereas the evaluation of possible solutions was shifted to later stages. Similarly, excavating requirements from distal artifacts was shunted toward the end of the release cycle. Such sequencing principles enabled the overall cognitive system to attend to separate tasks at different times throughout a release cycle and thus economized the use of the cognitive resources. We next discuss specific classes of requirements computation along with the specific heuristics that drive these broader patterns.

4.4.2 Discovery

In this facet, requirements are discovered through a process of *excavation* (Luckham, 2001) enabled by the rich social and structural interconnections within the project. "Root" artifacts, such as MRI, establish constraints to which proposed solutions must conform. By drawing on such root artifacts, developers excavate requirements (using Heuristics 1 and 2 in Table 10), which are then integrated into the focal software. In excavation, representations include suggestions and challenges expressed in IRC or future-state vision descriptions in Skype dialogues. We refer to these as *representations-as-conditions*, because they operationalize requirements knowledge in the form of assumptions, logic, and rules about the existing environment. Throughout excavation, the focal community focuses on identifying a set of requirements that it deems important and feasible based on the current matching of competencies and opportunities emerging from trends.

As embedded requirements are excavated and transformed into tangible conditions, they are carried forward to a specification. This movement relies on the use of heuristics for mapping elements of root artifacts to Rubinius specs. This is the first step toward instantiating representations-as-conditions into a representational form (i.e., specs) for which code can be developed and tested. This is a process of looking for "low-hanging fruit"—i.e., less-demanding cognitive tasks that are appropriate for inexperienced developers. Such tasks often include either writing specs (tests) or fixing bugs so as to make specs pass. For example, in one pull request, a developer stated:

> *This patch makes the String#squessze spec pass on 1.9. The spec only specified what error should be thrown, so I copied the message that gets thrown from MRI 1.9.2. . . This is my first pull request here . . . I couldn't tell if there were any other tests I should run to make sure I didn't break anything else, so that is where I stopped.*

This process guides developers to seek voluntary contributions that match their level of experience.

4.4.3 Specification

In this facet, requirements become specified through the interplay of the core committers where specific heuristics (3-6) guide developers to produce code and specs derived from *representations-as-conditions*. Interestingly, specification involves a smaller and more limited group of developers and artifacts. During this task, more experienced core developers work collaboratively, drawing upon their collective technical expertise to convert the requirements knowledge into readable and executable forms. We refer to this as *instantiation*, a process of detailing the technical implications of discovered requirements. Importantly, this process involves substantial trial-and-error learning, as developers expand or narrow requirements while they explore their technical consequences. In light of the extensive experimentation employed, we can say that this core instantiation process results in *representations-as-experiments*. In the case of concurrency, the core committers conducted a "first experiment to remove GIL," followed by "experiments with concurrent allocation and full stop GC." During this phase, several requirements may be instantiated in parallel. When experiments composed of working code and feasible specs have been produced, the newly computed representations-as-experiments are transmitted to the next phase—validation (Heuristic 7).

4.4.4 Validation

In this facet, a baseline implementation is transmitted to the wider community for testing. Cognitive effort in this task centers on *testing-in-the-wild* through the "scaffolding" (Clark, 1998) of multiple external

artifacts—e.g., MRI, RubySpec, Engine Yard platform, Puma IO, Travis CI. These artifacts are used to expose a multitude of bugs and additional necessary requirements. In contrast to the narrowing of the social distribution in specification, *testing-in-the-wild* expands the social distribution, with multiple developers and users integrating Rubinius into their local tasks (i.e., requirements embedded in distal artifacts). Any challenges that emerge in these tests give rise to *representations-as-discrepancies,* in the form of bug reports and feature requests. These representations-as-discrepancies reflect disconnects between the expectations and actual performance. Hence, Heuristic 8 guides developers to revise code based on bug reports so as to harmonize different parts of the code base, and Heuristic 9 guides the handling of bugs. In contrast to feature development, bug fixing is viewed as being less demanding cognitively and is therefore often conducted by peripheral developers. This effectively distributes the system's cognitive resources across a large number of bugs as they emerge. Testing-in-the-wild uncovers previously unexcavated new requirements and creates a recursive loop that feeds back to discovery and specification.

4.4.5 Recursion

By excavating additional requirements that maintain the software's integrity, the feedback mechanisms from validation to discovery (Heuristic 10) and specification (Heuristic 11) can invoke additional combinations of developers and artifacts to recompute successive instantiations of the same set of requirements. Based on our observations, however, this recursion sometimes fails, due to inconsistencies across the structural distribution, ineffective configurations of developers, or failure on part on developers to apply heuristics appropriately. One breakdown of the cognitive system illustrates several of these failures:

> *I actually broke Rubinius really badly at one point because I merged in a pull request from someone else. At the time I didn't spend too much time on Rubinius . . . So at the time you would merge the pull request locally and then run all the tests and then maybe do the automated testing locally and do some manual playing with it.*

Here the peripheral committer misunderstood the social distribution mechanism. The external community might devote technical skills and experience with other Ruby-related projects to Rubinius, but they were not as familiar with Rubinius-specific RE practices. Therefore, the focal community generally took charge of the workflow and acted as "gatekeepers," helping to convert external insights into internalized Rubinius requirements. When a peripheral committer did not follow the proper heuristics (i.e., 10 or 11), this resulted in a faulty merge. The feedback mechanisms thus call for reconfiguration of social and structural elements and recomputing of requirements discovery/specification shouldered by, for example, MRI/RubySpec. Overall, the above incident was ultimately a failure to follow an established computational structure within the distributed cognitive system.

5 Discussion

In this study, we have considered how RE is made possible in OSS development despite highly distributed teams and lack of formal governance. Accordingly, we approach OSS RE as a sociotechnical, distributed cognitive task whereby RE knowledge is maintained and computed within the sociotechnical system. We use this lens to analyze RE in a midsize successful OSS project called Rubinius. Through our analysis, we identify and illustrate the dynamic relationships between actors and artifacts within the Rubinius community (RQ1). In addition, we articulate three-layered temporal structuring mechanisms (i.e., trends, patterns, and heuristics), through which requirements knowledge is propagated in Rubinius (RQ2). By identifying the temporally ordered cognitive processes of excavation, instantiation, and testing-in-the-wild, we show how the Rubinius project transformed discovered requirements through a series of mappings between specific representational states that change requirements knowledge from representations-as-conditions, and representation-as-experiments, to representations-as-discrepancies. These mappings resulted over time in a collectively accepted understanding of the design requirements for a release (RQ3). Our inquiry offers a number of important insights for both IS scholars and OSS developers. With respect to research, our study fosters a broader understanding of requirements-oriented activity and the nature of software requirements. With respect to developers, our research suggests ways in which practitioners can effectively leverage resources and enhance project sustainability by attending to the sociotechnical distribution of cognitive effort in OSS projects.

5.1 Implications for Research

This study offers two primary research contributions: (1) providing an integrated and theoretically grounded model on how OSS communities, despite the lack of formal governance mechanisms and the presence of high distribution, can garner a robust and shared set of requirements through the interplay of actors, artifacts, and temporal structuring mechanisms; (2) an articulation of how OSS RE is different from RE in waterfall and agile contexts.

To the best of our knowledge, this study provides the first empirically based, theoretically grounded view of the computational mechanisms through which requirements are effectively managed in OSS development. Our exploratory analysis of the Rubinius project's computational structure outlines a generic set of generative cognitive mechanisms (Anderson et al., 2006)—a system of actors and artifacts which interact over time to compute requirements. Since OSS represents a relatively novel form of organizing (Puranam, Alexy, & Reitzig, 2014) in which collective action is possible despite the challenges of voluntary contribution, emergent coordination, and asynchronous work (Howison & Crowston, 2014), understanding how complex knowledge coordination is achieved in such a context is an important endeavor (Tuertscher, Garud, & Kumaraswamy, 2014). By integrating consideration of the actors, artifacts, and temporal mechanisms shaping the evolution of OSS systems, our model helps discern how OSS projects foster a shared understanding of requirements, despite their highly distributed nature. Specifically, the study suggests several key factors which contribute to the robustness of RE in OSS projects—including the leveraging of embedded requirements in distal artifacts, the integration of knowledge flows from external communities, and policies that foster broader engagement of community members (e.g., an open commit policy). Future research can build upon these insights to operationalize formal comparisons of distinct OSS communities with respect to their outcomes and sustainability.

The study also contributes to the broader research on RE. Our analysis illustrates the value of analyzing RE as a sociotechnical cognitive process, with an eye toward the dynamic relationships between actors and artifacts. Indeed, the distributed cognitive framing enables us to highlight several ways in which the OSS context differs from either waterfall or agile approaches with respect to the nature of social interactions, the role of artifacts, and the temporal sequencing of cognitive effort. These distinctions hold regardless of whether waterfall or agile approaches are distributed geographically (Espinosa, Slaughter, Kraut, & Herbsleb, 2007). OSS development eschews both the formal documentation of the waterfall model and the intense face-to-face interaction of agile development, and thereby moves away from both the strict phasing approach typical of waterfall and the locally constrained social iterations preferred in the agile approach. This has important implications for how we see the role of artifacts, social interactions, and temporal structuring.

First, our study indicates the curious role that digital artifacts (Kallinikos, Aaltonen, & Marton, 2013) play in mediating the OSS RE process. Rather than serving as centralized deposits of knowledge, which is common in waterfall and agile, artifacts become distributed and dynamically engaged in the process of discovering and implementing requirements. Here, the DCog perspective allows us to show the ways in which artifacts participate in the *process* of computing requirements, rather than simply being passive information repositories assumed in other approaches. When such artifacts are engaged in the process of requirements computation, they are continuously updated, and often serve as the original source for specific requirements, rather than just being used as ledgers where requirements emanating from the customer get recorded. When artifacts are viewed in this way, they become central to the ability of OSS projects to dynamically pick up requirements across a set of disparate subcommunities distributed in both space and time.

Second, while the lack of face-to-face interaction crucial to agile practices weakens the ability of a core group to communicate in an intensive manner, the use of distributed artifacts with related dialogues radically expands the diversity and socialization of individuals who may participate in the RE process (Dabbish, Stuart, Tsay, & Herbsleb, 2012). In a sense, the distributed artifacts engaged in the requirements computation process allow for "quasi-dialogues" involving "invisible others" who lurk behind the digital artifacts (Baralou & Tsoukas, 2015). This is crucially different from the forms of geographical distribution that we might observe in waterfall and agile processes, because the artifacts, through their dynamic participation, substitute for the formal working relationships within software development processes where all participants are paid either as employees or consultants. In sum, the participants can be spatially and temporally distributed, and may use the software being developed for varying purposes. This may lead to unexpected degrees of robustness, as the software can effectively be "stress-tested" from a multitude of angles. Third, while the temporal structuring of waterfall and agile is established beforehand (e.g., either as phased or iterative), the temporal structuring of OSS is fluid and contingent. Requirements move forward in the RE process if and when they activate various heuristics by meeting specific conditions. Each cycle of code revisions called "superpositioning" (Howison & Crowston, 2014) is triggered by specific conditions being met. This provides valuable knowledge with regard to when work (1) gets deferred, and (2) is rendered easy enough to be accomplished "with only a single programmer working on any one task . . . rather than being undertaken through structured team work" (Howison & Crowston, 2014, p. 29).

To a certain extent, our framing challenges the RE research community to reassess the fundamental

understanding of the "requirements" concept. This context necessitates a departure from the idea of explicit requirements as a strict articulation of desired functionality using formalized notation (Scacchi, 2002) to something which is improvised and emergent. Such a shift would enable us to embrace the complex sociotechnical mechanisms through which knowledge about user needs or desired features emerges, crystallizes, and evolves in a localized and distributed manner. With the rise of less-structured development approaches such as OSS and agile methods new theoretically grounded approaches to analyze RE processes become critical.

5.2 Implications for Practice

From a practice perspective, the framing of RE activities as a distributed cognitive process can alert OSS leaders to heed the critical roles played by different stakeholders and artifacts. Given the high rates of OSS project failure (Lee et al., 2009), a better understanding of the elements that contribute to effective requirements discovery is desirable. Most notably, the research underscores the importance of focusing on "OSS ecosystems" (Scacchi, Feller, Fitzgerald, Hissam, & Lakhani, 2006; Thomas & Hunt, 2004)—interrelated webs of developers, technologies, and projects—rather than distinct projects or tools. For example, our study highlights the robustness gained by engaging external stakeholders in the process of supplying requirements. By fostering linkages with an extended network of interested entities, a project can draw upon a richer set of perceived needs for their focal platform. In particular, the Rubinius case highlights the ways in which strong channels of exchange within external communities enhance discovery and validation.

The significant role of peripheral developers that we observe also highlights the importance of distributed requirements discovery. The integration of peripheral committers enables new requirements to emerge continuously as more established requirements are addressed. The open commit policy adopted by Rubinius, for example, ensures that requirements discovery and distribution is not limited to a small core; rather, requirements and possible solutions can originate from a broader community of developers. While the core team still controls the direction of the software evolution, the broader engagement calls into question the conventional framing of OSS core developers as "benevolent dictators" (Shah, 2006).

The research also underscores the critical role of artifacts in supporting or simplifying cognitive processes within the cognitive system. At the most fundamental level, artifacts serve an external memory function by capturing requirements knowledge at various points within a community. Given the transitory nature of OSS participation, this externalization remains a central consideration for a community's continuity and resiliency. In addition, as we observed in Rubinius, artifacts are essential in directing attention to new needs as they emerge. Such attention shifts can be fostered by the discovery of embedded requirements that mitigate the need for "greenfield" discovery. Third, system artifacts provide a material foundation for requirements evolution. For example, during instantiation, developers clarify requirements through experimentation by using diverse representations and exploring multiple solution possibilities—what Latour (1986) calls "thinking with eyes and hands."

Finally, the importance of heuristics in propagating representational states in the cognitive system suggests the value of conscious attention to the maintenance and dissemination of heuristics throughout a community. While heuristics provide support for rapid cognitive processing in complex environments (Kleinmuntz, 1985), inappropriate filtering or a mismatch between the situation and the heuristics can result in "severe and systematic errors" (Tversky & Kahneman, 1974). Fortunately, heuristics are also subject to conscious design and evaluation (Gigerenzer, 2008). Therefore, OSS project leaders may benefit from directed and constant evaluation of heuristics for excavation, instantiation, and testing. Such analysis could include identification of factors that contribute to user satisfaction, search strategies, and errors from past heuristics. In addition, OSS communities could consider redesigning communications media, processes, and tools that reinforce effective heuristic use.

5.3 Limitations and Future Research

Generalizability of our findings is naturally limited by the fact that we analyzed a single representative case. Furthermore, we acknowledge that Rubinius has some idiosyncratic characteristics, such as an open commit policy, highly technical nature, and significant interconnectedness with other OSS artifacts. Though the forms of social, structural, and temporal distribution within OSS communities may vary widely, we contend that the sociotechnical cognitive processes we identified are likely to remain consistent in other OSS environments due to the similarity in tasks, social organization, and deployed artifacts. In this sense, we are claiming that our proposed framework provides strong theoretical generalizability (Lee & Baskerville, 2003), even when specific facts of our case may not generalize to all other OSS projects. Therefore, we expect the theoretical mechanisms that we have proposed to remain quite stable.

Several avenues are open for additional research. Our DCog model can be applied to other software environments, including structured development,

"commercial off-the-shelf"-based development, and agile development. The computational structures employed in these environments are likely to vary with respect to social, structural, and temporal forms of cognitive activity distribution, as well as the heuristics used. Consistent application of the perspective to multiple environments should foster inductive theory-driven identification of appropriate computational configurations that influence software project success, developer or user satisfaction, and innovativeness.

6 Conclusion

We inquired into how DCog can account for RE management in OSS. We find that establishing requirements is a "computational" process, whereby requirements knowledge is transformed across a system that is distributed socially, structurally, and temporally. Distinct components and mechanisms within each form of distribution carry out distinct roles within the cognitive system, which collectively computes requirements in the absence of formal planning or hierarchical authority. The study offers important insights for both research and practice in OSS. From a research perspective, the study underscores the importance of attending to the temporal unfolding of interactions between human and structural elements in software development. From a practice perspective, the study highlights a range of cognitive dynamics, which can inform OSS leaders as they seek to support and maintain vibrant development communities.

Acknowledgements

The authors sincerely thank the senior editor Sandeep Purao and the two anonymous reviewers for their valuable feedback throughout the review process. The authors also thank all participants of the Rubinius project who gave their time and support voluntarily and with excitement. This research was partially supported by a grant from the CISE division of the U.S. National Science Foundation [1217345].

References

Alexander, C. (1964). *Notes on the synthesis of form*. Cambridge, MA: Harvard University Press.

Anderson, J. R. (2005). *Cognitive psychology and its implications* (6th ed.). New York, NY: Worth.

Anderson, P. J. J., Blatt, R., Christianson, M. K., Grant, A. M., Marquis, C., Neuman, E. J., Sonenshein, S., & Sutcliffe, K. M. (2006). Understanding mechanisms in organizational research: Reflections from a collective journey. *Journal of Management Inquiry*, *15*(2), 102–113.

Asundi, J., & 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*. AIS.

Aurum, A., & Wohlin, C. (2005). Requirements engineering: Setting the context. In A. Aurum & C. Wohlin (Eds.), *Engineering and Managing Software Requirements* (pp. 1–15). Berlin: Springer.

Bahill, T. A., & Henderson, S. J. (2005). Requirements development, verification, and validation exhibited in famous failures. *Systems Engineering*, *8*(1), 1–14.

Baralou, E., & Tsoukas, H. (2015). How is new organizational knowledge created in a virtual context? An ethnographic study. *Organization Studies*, *36*(5), 593–620.

Beck, K., & Andres, C. (2004). *Extreme programming explained: Embrace change* (2nd ed.). Upper Saddle River, NJ: Addison-Wesley.

Bell, T. E., & Thayer, T. A. (1976). Software requirements: Are they really a problem? In *Proceedings of 2nd International Conference on Software Engineering* (pp. 61–68). IEEE.

Brooks, F. P. (1995). *The mythical man-month: essays on software engineering*. Upper Saddle River, NJ: Addison-Wesley Professional.

Callon, M. (1986). Some elements of a sociology of translation: Domestication of the scallops and the fishermen of St. Brieuc Bay. In J. Law (Ed.), *Power, Action, and Belief: A New Sociology of Knowledge?* (pp. 196–223). London: Routledge.

Cao, L., & Ramesh, B. (2008). Agile requirements engineering practices: An empirical study. *IEEE Software*, *25*(1), 60–67.

Cheng, B. H., & Atlee, J. M. (2009). Current and future research directions in requirements engineering. In K. J. Lyytinen, P. Loucopoulos, J. Mylopoulos, & W. N. Robinson (Eds.), *Design requirements engineering: A ten-year perspective* (pp. 11–43). Berlin: Springer.

Clark, A. (1998). *Being there: Putting brain, body and world together again*. Cambridge, MA: Massachusetts Institute of Technology Press.

Conboy, K., Coyle, S., Wang, X., & Pikkarainen, M. (2011). People over process: Key challenges in agile development. *IEEE Software*, *28*(4), 48–57.

Corbin, J., & Strauss, A. (2008). *Basics of Qualitative research: Techniques and procedures for developing grounded theory* (3rd ed.). Thousand Oaks, CA: SAGE.

Cromwell, H. C., & Panksepp, J. (2011). Rethinking the cognitive revolution from a neural perspective: How overuse/misuse of the term "cognition" and the neglect of affective controls in behavioral neuroscience could be delaying progress in understanding the brainmind. *Neuroscience and Biobehavioral Reviews*, *35*(9), 2026–2035.

Crowston, K., & Howison, J. (2005). The social structure of free and open source software development. *First Monday*, *10*(2). Rerieved from https://journals.uic.edu/ojs/index.php/fm/article/view/1478/1393

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

Crowston, K., Li, Q., Wei, K., Eseryel, Y. U., & Howison, J. (2007). Self-Organization of teams for free/libre open source software development. *Information and Software Technology*, *49*(6), 564–575.

Dabbish, L., Stuart, C., Tsay, J., & Herbsleb, J. (2012). Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work* (pp. 1277–1286). ACM

Damian, D., Helms, R., Kwan, I., Marczak, S., & 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* (pp. 442–451). IEEE.

Eisenhardt, K. M. (1989). Building theories from case study research. *The Academy of Management Review*, *14*(4), 532–550.

Ernst, N. A., & Murphy, G. C. (2012). Case studies in just-in-time requirements analysis. In *IEEE 2nd International Workshop on Empirical Requirements Engineering* (pp. 25–32). IEEE.

Espinosa, J. A., Slaughter, S. A., Kraut, R. E., & Herbsleb, J. D. (2007). Team knowledge and coordination in geographically distributed software development. *Journal of Management Information Systems*, *24*(1), 135–169.

Fitzgerald, G., & Avison, D. E. (2003). Where now for development methodologies? *Communications of the ACM*, *46*(1), 79–82.

Fomin, V., & De Vaujany, F. X. (2008). Theories of ICT design: Where social studies of technology meet the distributed cognitive perspective. In *Proceedings of the 29th International Conference on Information Systems*. AIS.

Fowler, M., & Highsmith, J. (2001). The agile manifesto. *Software Development*, *9*(8), 28–35.

Gigerenzer, G. (2008). Why heuristics work. *Perspectives on Psychological Science*, *3*(20), 20–29.

Glaser, B. G., & Strauss, A. L. (1967). The discovery of grounded theory. *International Journal of Qualitative Methods*, *5*(1), 1–10.

Hansen, S., Berente, N., & Lyytinen, K. (2009). Requirements in the 21st century: Current practice and emerging trends. In K. Lyytinen, P. Loucopoulos, J. Mylopoulos, & B. Robinson (Eds.), *Design Requirements Engineering: A Ten-Year Perspective* (pp. 44–87). Berlin: Springer.

Hansen, S. W., Lyytinen, K., & Kharabe, A. (2015). a tale of requirements computation in two projects : A distributed cognition view. In *Proceedings of the 2015 International Conference on Information Systems*. AIS.

Hansen, S. W., Robinson, W. N., & Lyytinen, K. J. (2012). Computing requirements: Cognitive approaches to distributed requirements engineering. In *2012 45th Hawaii International Conference on System Sciences* (pp. 5224–5233). AIS.

Headland, T. N., Pike, K. L., & Harris, M. (1990). *Emics and etics: The insider/outsider debate.* Frontiers of anthropology vol. 7. Losa Angeles, CA: SAGE.

Hickey, A. M., & 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). IEEE.

Highsmith, J., & Cockburn, A. (2001). Agile software development: The business of innovation. *Computer*, *34*(9), 120–127.

Hollan, J., Hutchins, E., & Kirsh, D. (2000). Distributed cognition: Toward a new foundation for human-computer interaction research. *ACM Transactions on Computer-Human Interaction*, *7*(2), 174–196.

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

Hutchins, E. (1995). *Cognition in the wild*. Cambridge, MA: Massachusetts Institute of Technology Press.

Hutchins, E., & Klausen, T. (1996). Distributed cognition in an airline cockpit. In Y. Engeström & D. Middleton (Eds.), *Cognition and Communication at Work* (pp. 15–34). Cambridge, U.K.: Cambridge University Press.

Jarke, M., Loucopoulos, P., Lyytinen, K., Mylopoulos, J., & Robinson, W. (2011). The brave new world of design requirements. *Information Systems*, *36*(7), 992–1008.

Kallinikos, J., Aaltonen, A., & Marton, A. (2013). The ambivalent ontology of digital artifacts. *MIS Quarterly*, *37*(2), 357–370.

Kleinmuntz, D. N. (1985). Cognitive heuristics and feedback in a dynamic decision environment. *Management Science*, *31*(6), 680–702.

Larman, C., & Basili, V. R. (2003). Iterative and incremental development: A brief history. *Computer*, *36*(6), 47–56.

Latour, B. (1986). Visualization and cognition: Thinking with eyes and hands. In H. Kuklick (Ed.), *Knowledge and society studies in the sociology of culture past and present* (pp. 1–40). Greenwich, CT: Jai.

Latour, B. (1987). *Science in action: How to follow scientists and engineers through society*. Cambridge, MA: Harvard University Press.

Lave, J. (1988). *Cognition in practice*. Cambridge, U.K.: Cambridge University Press.

Lee, A. S., & Baskerville, R. L. (2003). Generalizing generalizability in information systems research. *Information Systems Research*, *14*(3), 221–243.

Lee, S. Y. T., Kim, H. W., & Gupta, S. (2009). Measuring open source software success. *Omega*, *37*(2), 426–438.

Leont'ev, A. N. (1981). *Problems of the development of mind*. Moscow: Progress.

Luckham, D. C. (2001). *The Power of events: An introduction to complex event processing in distributed enterprise systems*. Boston, MA: Addison-Wesley.

Mangalaraj, G., Nerur, S., Mahapatra, R., & Price, K. H. (2014). Distributed cognition in software design: An experimental investigation of the role of design patterns and collaboration. *MIS Quarterly*, *38*(1), 249–274.

Mathiassen, L., Tuunanen, T., Saarinen, T., & Rossi, M. (2007). A contingency model for requirements development. *Journal of the Association for Information Systems*, *8*(11), 569–597.

Metzler, T., & Shea, K. (2011). Taxonomy of cognitive functions. In *Proceedings of the 18th International Conference on Engineering Design* (pp. 330–341). Design Society.

Mockus, A., Fielding, R. T., & Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, *11*(3), 309–346.

Nardi, B. (1996). Studying Context: A comparison of activity theory, situated action models, and distributed cognition. In B. Nardi (Ed.), *Context and consciousness: Activity theory and human-computer interaction*. Cambridge, MA: Massachusetts Institute of Technology Press.

Noll, J., & 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* (pp. 35–40). ACM.

Perkins, D. N. (1993). Person-plus: A Distributed view of thinking and learning. In S. Gavriel (Ed.), *Distributed cognitions: Psychological and educational considerations* (pp. 88–110). Cambridge, U.K.: Cambridge University Press.

Petersen, K., & Wohlin, C. (2010). The effect of moving from a plan-driven to an incremental software development approach with agile practices: An industrial case study. *Empirical Software Engineering*, *15*(6), 654–693.

Pollock, N., & Williams, R. (2009). Global software and its provenance: generification work in the production of organisational software packages. In V. Alex, M. Hartswood, R. Procter, M. Rouncefield, R. Slack, & M. Büscher (Eds.), *Configuring user-designer relations: interdisciplinary perspective* (pp. 193–218). London: Springer.

Port, D., & Bui, T. (2009). Simulating mixed agile and plan-based requirements prioritization strategies: Proof-of-concept and practical implications. *European Journal of Information Systems*, *18*(4), 317–331.

Puranam, P., Alexy, O., & Reitzig, M. (2014). What's "new" about new forms of organizing? *Academy of Management Review*, *39*(2), 162–180.

Ramesh, B., Mohan, K., & Cao, L. (2012). Ambidexterity in agile distributed development : An ambidexterity in agile distributed development : An empirical Investigation. *Information Systems Journal*, *23*(2), 323–339.

Raymond, E. (1999). The Cathedral and the bazaar. *Knowledge, Technology & Policy*, *12*(3), 23–49.

Robles, G., & Gonzalez-Barahona, J. M. (2006). Contributor turnover in libre software projects. In E. Damian, B. Fitzgerald, W. Scacchi, M. Scotto, & G. Succi (Eds.), *Open source systems* (pp. 273–286). Boston: Springer.

Rogers, Y., & Ellis, J. (1994). Distributed cognition: An alternative framework for analysing and explaining collaborative working. *Journal of Information Technology*, *9*(2), 119–128.

Royce, W. (1970). Managing the development of large software systems. In *Proceedings of IEEE WESCON* (pp. 1–9). IEEE.

Scacchi, W. (2002). Understanding the requirements for developing open source software systems. *iEE Proceedings Software*, *149*(1), 24–39.

Scacchi, W. (2009). Understanding requirements for open source software. In K. Lyytinen, P. Loucopoulos, J. Mylopoulos, & B. Robinson (Eds.), *Design requirements engineering: A ten-year perspective* (pp. 467–494). Berlin: Springer.

Scacchi, W., Feller, J., Fitzgerald, B., Hissam, S., & Lakhani, K. (2006). Understanding free/open source software development processes. *Software Process: Improvement and Practice*, *11*(2), 95–105.

Shah, S. K. (2006). Motivation, governance, and the viability of hybrid forms in open source

software development. *Management Science*, *52*(7), 1000–1014.

Shettleworth, S. J. (2009). *Cognition, evolution, and behavior*. Oxford, U.K.: Oxford University Press.

Simon, H. A. (1980). Cognitive science: The newest science of the artificial. *Cognitive Science*, *4*, 33–46.

Strauss, A., & Corbin, J. (1990). *Basics of qualitative research: Grounded theory procedures and techniques*. Newbury Park, CA: SAGE.

Suchman, L. (1987). *Plans and situated actions*. Cambridge, U.K.: Cambridge University Press.

Thomas, D., & Hunt, A. (2004). Open source ecosystems. *IEEE Software*, *21*(4), 89–91.

Tuertscher, P., Garud, R., & Kumaraswamy, A. (2014). Justification and interlaced knowledge at ATLAS, CERN. *Organization Science*, *25*(6), 1579–1608.

Tversky, A., & Kahneman, D. (1974). Judgment under uncertainty: Heuristics and biases. *Science*, *185*(4157), 1124–1131.

Valverde, S., & Solé, R. V. (2007). Self-organization versus hierarchy in open-source social networks. *Physical Review E*, *76*(4).

Vidgen, R., & Wang, X. (2009). Coevolving systems and the organization of agile software development. *Information Systems Research*, *20*(3), 355–376.

Vygotsky, L. S. (1987). *Thinking and speech*. New York, NY: Plenum.

Wegner, D. M. (1995). A computer network model of human transactive memory. *Social Cognition*, *13*(3), 319–339.

Yin, R. K. (2009). *Case study research: Design and methods* (4th ed.). Thousand Oaks, CA: SAGE.

Appendix A. Software Development Methodologies

In Table 11 below we provide an overview of the main characteristics of waterfall, agile, and open source.

Table 11. Comparison of Three RE Approaches

Qualities	**Waterfall** (Bell & Thayer, 1976; Royce, 1970)	**Agile development** (Fowler & Highsmith, 2001; Vidgen & Wang, 2009)	**Open source** (Howison & Crowston, 2014; Scacchi, 2009)
Methodological Characteristics			
Team structure	Stable	Stable	Fluid
Location	Mainly colocated	Mainly colocated	Distributed
Role of developer	Employee or hired for work	Employee or hired for work	Volunteer (secondary employee)
Coordination mechanisms	Formal planning & documentation, scripting	Face-to-face exchange, "big & visible" displays of project knowledge	Emergent and largely computer-mediated (e.g., via GitHub)
Requirements Facets			
Discovery	Formal planning; requirements set prior to design and implementation	Close collaboration in a core team; cocreation with customers	Developer-driven exploration features and "scratching itches"
Specification	Formal documentation; standardized requirements specification documents	List of flexible backlogs; mock-ups/prototypes	"Informalisms" (e.g., discussion forums, email messages, test results)
Validation	Document review and sign-off	Iterative review and test-driven development	Informal ex ante evaluation Post hoc evaluation and use; testing platforms

Appendix B. The Rubinius Project

Rubinius is an implementation of the Ruby programming language through a virtual machine (VM) that provides a runtime environment and a dynamic compiler for Ruby. As a partially sponsored OSS project, Rubinius has been relatively successful in this 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 highly 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 full-time on the project. On May 14, 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 system and involves 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 changed the dynamic within the core development team somewhat, the leadership transition has not substantively impeded the overall development effort.

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 played a significant role in shouldering RE practices. In the context of Rubinius, one important artifact is RubySpec which is highly relevant in RE management. RubySpec consists of an executable specification for the Ruby programming language and describes Ruby language syntax and its standard library classes. It captures behaviors of Matz's Ruby Interpreter (MRI, i.e., the reference implementation of the Ruby programming language setting the de facto standard for any Ruby interpreter) and tells how Rubinius should execute the same behavior so as to evaluate the correctness of its implementation. In contrast to the traditional specification documents which are usually written in a formal, literal way upfront, RubySpec is a living open source project repository on GitHub composed of executable specs and test cases. For example, a spec named array_spec.rb can be represented in the RubySpec as follows:

```
"require File.dirname(__FILE__) + '/../../spec_helper'
describe "Array" do
 it "includes Enumerable" do
   Array.ancestors.include?(Enumerable).should == true
 end
end"
```

Appendix C. Interview Protocol

This interview intended to explore evolution of coding and requirements engineering practices. Specifically, the interviews focused on interviewee's personal experiences/views of these practices.

Background

- Could you tell us about your background, and how you came to work on this project?
- Could you describe your process of entering the community of this particular project?
- When do you usually spend your energy on the project? How many hours do you usually spend?

Coding/Requirements engineering practices

- Could you provide an overview of events since you joined the project? Were there any notable events of importance during this period?
- Were there any conversations or chats discussing what the project was going to do before the event? Where did the conversations or chats take place? Did they happen on a regular basis? Did anyone within the community take primary responsibility for documenting them? Where were they documented?
- Which tasks do you take on within the project? Why did you choose those tasks?
- Please walk us through a typical/recent issue or commit you submitted.
    - Where did your idea come from? Why did you think the problem was necessary to address?
    - What techniques did you employ to identify the problem you were addressing? What development tools or resources were used in identifying the problem?
    - Were there any people that you turned to for identifying the problem? Could you tell us about your interactions with other project members, particularly as it pertains to the identification of the problem? How did you initiate contact, with whom did you make contact, and why was the process done this way?
    - Could you describe how you solved the problem? What technology platforms, modeling techniques and tools were used? What were specific reasons for choosing them?
    - Were there any project members or any resources that you turned to for solving the problem? How did they help you in this regard? Which communication channels did you use?

- Did you and other project members share the same understanding of what needed to be done during the process? Were there any disagreements? If there were conflicts, how did you negotiate them and achieve consensus?

Next steps

- We would like to have a continuing relationship. Would it be possible for us to talk again in the future?
- What would be interesting for you to find out? What would you like to have visibility into?
- Who else can we talk to?

Appendix D. Selection of Archival Entries Based on Keywords

Based on the coding of the interviews we identified a set of emic (Headland et al., 1990) keywords that capture salient topics within the Rubinius project. We identified 20 such keywords:

1. MRI / Matz Ruby Interpreter
2. Flip Flop / FlipFlop / Flip-Flop
3. Concurrency / concurrent
4. GLI / Global Interpreter Lock
5. IO / I/O
6. Hydra
7. GC / Garbage Collection
8. Puma
9. Travis
10. CI / Continuous Integration
11. Engine Yard / EngineYard
12. Rspec
13. RDocs
14. RubySpec
15. GitHub
16. Git
17. Gdb
18. Insiter
19. Real-time web / realtime web
20. Multicore / multicore / processors / CPUs

These keywords were then used to search the GitHub repository, project website, developer mailing list, as well as IRC, for the time period May 14, 2010 (when Rubinius 1.0 was launched) to November 2, 2012 (when the Rubinius 2.0 preview was launched). All items in these archival data sources that matched any of these search terms were then selected for coding. The selected archival items were then loaded into Dedoose, and were then coded using the theoretical codes extracted from the interview data. Overall, the data funnel can be seen in Table 12 below.

Table 12. Archival Analysis Data Funnel

	Issues	**Pull requests**	**Blog posts**	**Mailing List threads**[a]
Total	1082	604	26	35
Selected	816	233	19	25

[a] Each thread contains multiple emails

The data were collected from these hyperlinks, corresponding to the columns in Table 12:

- https://github.com/rubinius/rubinius/issues
- https://github.com/rubinius/rubinius/pulls
- https://github.com/rubinius/rubinius-archive/tree/master/_posts
- https://groups.google.com/forum/#!forum/rubinius-dev

Appendix E. Coding Protocol

The initial round of coding focused on the identification of themes (i.e., themes and concepts) and central explanatory categories within the interview data, concerning the ways in which requirements knowledge is processed across actors and artifacts over time. Additional codes and analytical memos were developed during the coding process. This initial round of coding was followed by a round of axial coding in which we consolidated some of the codes from the open coding phase. During axial coding, we identified 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 & Strauss, 2008). Finally, we conducted a round of selective coding 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.

At the point where substantive categories had emerged and we had gained a certain level of stability and saturation, we approached the archival data sources. Due to the overwhelming amounts of archival data (thousands of email conversations, workflows, and public documents), the themes that emerged from the interviews were used as guideposts. By searching archives using keywords associated with prominent themes, relevant passages were identified and coded to support comparisons as well as to increase the richness and integration of the emerging theory.

Comparing across both interviews and archival data, we refined the categories 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 archival data sources. Code generation and refinement proceeded until the researchers deemed that theoretical saturation was achieved (Eisenhardt, 1989; Glaser & 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 emerging theoretical scheme (Corbin & Strauss, 2008). In the end, 611 codable moments were recognized along with 13 analytical memos and 87 distinctive codes.

About the Authors

Xuan Xiao is an assistant professor at the School of Management, Guangzhou University. She received her PhD in Business Administration from Harbin Institute of Technology, China. Her research interests include social networking service and open source software. She has published in a number of peer-reviewed journals and conferences including the International Conference on Information Systems, Hawaii International Conference on System Sciences, and the Academy of Management meeting.

Aron Lindberg is an assistant professor of information systems at the School of Business, Stevens Institute of Technology. He received his PhD at Case Western Reserve University, and primarily studies complex design processes, often using a combination of qualitative and computational methods. His research has been published or is forthcoming in major journals such as *Information Systems Research, Journal of the Association for Information Systems, Communications of the Association for Information Systems,* and *IEEE Computer.*

Sean Hansen Sean Hansen is an associate professor of management information systems and chair of the Department of MIS, Marketing, & Digital Business at Rochester Institute of Technology's Saunders College of Business. He earned his PhD and MBA from the Weatherhead School of Management at Case Western Reserve University. His research interests include information systems development, health IT, requirements engineering, and distributed cognition

Kalle Lyytinen Kalle Lyytinen (PhD, Computer Science, University of Jyväskylä; Dr. h.c. Umeå University, Copenhagen Business School, Lappeenranta University of Technology) is Distinguished University Professor and Iris S. Wolstein Professor of Management Design at Case Western Reserve University, and a distinguished visiting professor at Aalto University, Finland. Between 1992 and 2012 he was the third most productive scholar in the IS field when measured by the AIS Basket of 8 journals; he is among the top five IS scholars in terms of his h-index (83); he is a LEO Award recipient (2013), Association for Information Systems fellow (2004), and the former chair of IFIP WG 8.2 "Information systems and organizations." His Erdos number is 3 and he has the highest network centrality among all published IS scholars. He has published over 350 refereed articles and edited or written over 30 books or special issues. He conducts research that explores digital innovation, especially in relation to nature and organization of digital innovation, design work, requirements in large scale systems, diffusion and assimilation of digital innovations, and emergence digital infrastructures.