IEE Proceedings -- Software
Paper number 29840, Accepted for publication with revisions, December 2001.

Understanding the Requirements for Developing Open Source Software Systems

Walt Scacchi
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425 USA
http://www.ics.uci.edu/~wscacchi
wscacchi@ics.uci.edu

Abstract

This study presents an initial set of findings from an empirical study of social processes, technical system configurations, organizational contexts, and interrelationships that give rise to open software. The focus is directed at understanding the requirements for open software development efforts, and how the development of these requirements differs from those traditional to software engineering and requirements engineering. Four open software development communities are described, examined, and compared to help discover what these differences may be. Eight kinds of software informalisms are found to play a critical role in the elicitation, analysis, specification, validation, and management of requirements for developing open software systems. Subsequently, understanding the roles these software informalisms take in a new formulation of the requirements development process for open source software is the focus of this study. This focus enables considering a reformulation of the requirements engineering process and its associated artifacts or (in)formalisms to better account for the requirements for developing open source software systems.

Revised version appears in:

***IEE Proceedings--Software*, 149(1), 24-39, February 2002.**

1. Overview

The focus in this paper is directed at understanding the requirements for open software development efforts, and how the development of these requirements differs from those traditional to software engineering and requirements engineering [10, 17, 22, 28]. It is not about hypothesis testing or testing the viability of a prescriptive software engineering methodology or notational form. Instead, this study is about discovery, description, and abstraction of open software development practices and artifacts in different settings in different communities. It is about expanding our notions of what requirements engineering processes and process models need to address to account for open source software development. But to set the stage for such an analysis, we first need to characterize the research methods and principles employed in this study. Subsequently, these are used to understand what open software communities are being examined, and what characteristics distinguish one community from another. This is followed by the model of the processes and artifacts that give rise to the requirements for developing open source software systems. The model and artifacts are the focus of the analysis and basis of the concluding discussion. This includes a discussion of what is new or different in the findings presented in this report, as well as some of their implications for what can or should be formalized when developing different kinds of open software systems.

2. Research methodology: comparative methods and principles

This study reports on findings and results from an ongoing investigation of the socio-technical processes, work practices, and community forms found in open source software development. The purpose of this multi-year investigation is to develop narrative, semi-structured (i.e., hypertextual), and formal computational models of these processes, practices, and community forms. This report presents a systematic narrative model that characterizes the processes through which the requirements for open source software systems are developed. The model compares in form, and presents a contrasting account of, how software requirements differ between traditional software engineering and open source approaches. The model is descriptive and empirically grounded. The model is also comparative in that it attempts to characterize an open source requirements engineering process that transcends the practice in a particular project, or within a particular community. This comparative dimension is necessary to avoid premature generalizations about processes or practices associated with a particular open software system or those that receive substantial attention in the news media (e.g., the GNU/Linux operating system). Such comparison also allows for system projects that may follow a different form or version of open source software development (e.g., those in the Astrophysics research community or networked computer game arena). Subsequently, the model is neither prescriptive nor proscriptive in that it does not characterize what should be or what might be done in order to develop open source software requirements, except in the concluding discussion, where such remarks are bracketed and qualified.

Comparative case studies are also important in that they can serve as foundation for the formalization of our findings and process models as a *process meta-model* [24]. Such a meta-model can be used to construct a predictive, testable, and incrementally refined theory of open software development processes within or across communities or projects. A process meta-model can also be used to configure, generate, or instantiate Web-based process modeling, prototyping, and enactment environments that enable modeled processes to be globally deployed and computationally supported [e.g., 26, 27]. This may be of value to other academic research or commercial development organizations that seek to adopt "best practices" for open software development processes well suited to their needs and situation. Therefore, the study and results presented in this report denote a new foundation on which computational models of open software requirements processes may be developed, as well as their subsequent analysis, simulation, or redesign [34].

The study reported here entails the use of empirical field study methods [40] that follow conform to the principles for conducting and evaluating interpretive research design [19] as identified here. Seven principles are used in this study in the following manner.

The first principle is that of the *hermeneutic circle,* here focusing attention on the analysis of the whole open source requirements process, how it emerges from consideration of its parts (i.e., requirements sub-processes and associated artifacts), and how the whole interacts with and shapes these parts.

The second principle is that of *contextualization,* which draws attention to the need to identify a web of situations, conditions, or events that characterize the social and historical background of requirements engineering practices found in open source development projects or communities [20]. This begins in Sections 3 and 4, then continues throughout the presentation of the descriptive model and informal artifacts of open software requirements processes.

The next principle is that of *revealing the interaction of the researcher and the subjects/artifacts*. This is a basic concern that must be addressed and disclosed whenever the research involves an ethnographic field study, particularly those requiring virtual ethnography [16], as is the situation here. In this study, the researcher acted as a participant observer who seeks to understand how open source software requirements for a set of specific open software systems are developed, and to what ends. In the virtual worlds of open software development projects, there is generally no corporate workplace or single location where software development work occurs. Thus, traditional face-to-face ethnography and visible participant observation cannot readily occur. What occurs, and where it occurs, is generally online (i.e., hosted on a Web site or interactively accessed via Internet mechanisms), open to public access[1], and dispersed across geographic space and multiple time zones. Informal hallway conversations, as well as organized and scheduled meetings (rare though they may be), generally take place in an electronic and publicly visible online manner, though the requirements development work itself may be implied, hidden, or otherwise invisible. Subsequently, as a Web-based participant, the researcher could "sit in" or lurk on a group chat among core developers when it was a pre-announced event, as casual developers or other reviewers regularly do.

Alternatively, like many others potential or active participants, one could simply browse email, community bulletin boards (bboards), and related Web site postings that signal the occurrence of events or situations of interest (e.g., software release announcements or problem reports). These modes of participation are not uncommon, and are cited as one way how project newcomers can join in and learn the domain language and issues, with minimal bother or distraction of those doing the core development effort [11, 29, 32]. Social interaction among open software project participants may rarely, if ever, be face-to-face or co-located in most open source development efforts. However, real-world events like professional conferences may enable distant collaborators to meet, interact with, and learn about one another, but such events may occur only once a year, or be effectively inaccessible to project participants due to its distant location.

In open software projects, social and technical interaction primarily occurs in a networked mediated computing environment populated with Web browsers, email/bboards (and sometimes instant messaging) utilities, source text editors, and other software development tools (e.g., compilers, debuggers, and version control systems [14]). Each program/tool runs asynchronously in different end-user processes (application windows) appearing on a graphic user interface, as well as appearing as artifacts stored and distributed across one or more repositories [26, 27]. The workplace of open source software development is on the screen together with the furniture and surroundings that house it. This workplace is experienced, navigated, and interacted through keystrokes and mouse/cursor movement gestures in concert with what is seen, read, or written (typed). Thus, to observe, participate, and comprehend what's going on in an open source development project, it is necessary to become situated, embedded, and immersed as a software contributor, reviewer, discussant, or reader within such projects, and within such a networked computing environment and workplace setting.

In all the projects reported here, the researcher was a reader who acted as an interested but unfamiliar software developer seeking to understand, review, or discuss with other participants contemporary or legacy software development problems, opportunities, features, and constraints here [cf. 20]. This occurred while asynchronously running the end-user application sessions and networked computing environment indicated here, for between 0.5-10+ hours at a time, totaling more than 200 hours over a period of 10 months. The data exhibits that appear in the Section 5 are a comparative though minute sample of such

[1] Some modes of participation may be restricted--for example, anyone may check out and download source code, but typically only those approved by the core development team may upload and check in modified code into a shared repository [14].

Web-based experiences and shared artifacts, all presented as screen displays as could be seen by a community participant, as captured while and where they encountered by the researcher.

The fourth principle is that of *abstraction and generalization*. The choice to examine and compare requirements engineering activities and artifacts across four different software development communities is the response to this motivation. The requirements engineering process in Section 5 provides both a description and comparison that spans four distinct communities. The model abstracts details that are presented as summary terms (identified by sub-section headings) that span multiple open source projects in the sample space in order to create a more general model that covers the details across all the examined projects. Similarly, the classification of open source software requirements artifacts as software informalisms in Section 6 also reflects a similar kind of generalization across project and across community.

The fifth principle is that of *dialogical reasoning* which compares the received wisdom of extant theory or methodology in the software requirements engineering community, with that found empirically through participant observation in open source software development efforts. This appears in Sections 4 and 5.

The sixth principle is that of *multiple interpretations* which highlights the need to recognize that different participants see and experience things differently, though they may still be able to communicate and share these things with some degree of similarity or replication. The descriptive model presented in this report is a unique characterization that does not appear in any of the open source software development efforts or communities described. Thus, the interpretation here is therefore subject to the seventh and last principle, which is that of *suspicion to possible biases or systematic distortions* in this presentation. The reader is cautioned to look for alternative explanations or arrangements of data that might give rise to a different model of the requirements process to that which follows. If found, such models should be published as a contribution for review and comparison to the one presented here. Alternatively, the model presented here could be revised and updated to account for alternative interpretations, as a further generalization that better accounts for the other principles listed here. These issues are addressed in Sections 7 and 8.

3. Understanding open software development across different communities

We assume there is no *a priori* model or globally accepted framework that defines how open software is or should be developed. Subsequently, our starting point is to investigate open software practices in different communities from an ethnographic perspective [2, 28, 38].

We have chosen four different communities to study. These are those centered about the development of software for networked computer games, Internet/Web infrastructure, X-ray astronomy and deep space imaging, and academic software design research. In contrast to efforts that draw attention to generally one (but sometimes many) open source development project(s) within a single community [e.g., 11, 32], there is something to be gained by examining and comparing the communities, processes, and practices of open software development in different communities. This may help clarify what observations may be specific to a given community (e.g., GNU/Linux projects), compared to those that span multiple, and mostly distinct communities. In this study, two of the communities are primarily oriented to develop software that supports scholarly research (X-ray astronomy and academic software design research) with rather small user communities. In contrast, the other two communities are oriented primarily towards software development efforts that may replace/create commercially viable systems that are used by large end-user communities. Thus, there is a sample space that allows comparison of different kinds. So to begin, each community in this study can be briefly characterized.

3.1 Networked computer game worlds

Participants in this community focus on the development and evolution of first person shooters (FPS) games (e.g., *Quake Arena*, *Unreal Tournament*), massive multiplayer online role-playing games (e.g., *Everquest*, *Ultima Online*), and others (e.g., *The Sims* (maxis.com), *Neverwinter Nights* (bioware.com)). Interest in networked computer games and gaming environments, as well as their single-user counterparts, has exploded in recent years as a major (now global) mode of entertainment and playful fun. The release of *DOOM*, an early FPS game, onto the Web in open source form in the mid 1990's, began what is widely recognized the landmark event that launched the development and redistribution of so-called PC game

mods [Cleveland 2001]. Mods are updates to or variants of proprietary (closed source) computer game engines that provide extension mechanisms like game scripting languages to modify and extend a game. Mods are created by small numbers of users who want and are able to modify games, compared to the huge numbers of players that enthusiastically use the games as provided. The scope of mods has expanded to include new game types, game character models and skins (surface textures), levels (game play arenas or virtual worlds), and artificially intelligent game bots (in-game opponents). The companies, id Software (makers of *DOOM* and the *Quake* game family) and Epic Games (makers of the *Unreal* game family) helped encourage the open extension of proprietary game engines. This was done through both game licenses that require publicly distributed mods to be open source, and the provision of mod tools (level editors, model builders) and game engine programming or scripting languages for modifying game objects, behavior, as well as the potential to create entirely new games.[2] These companies also recruit new game development staff from the community of mod developers (see Exhibit 4).

3.2 Internet/Web infrastructure

Participants in this community focus on the development and evolution of systems like the Apache web server, Mozilla Web browser[3], K Development Environment (KDE), InterNet News server, OpenBSD, `mono` (an open source implementation of .NET, mostly independent from Microsoft), and thousands of others[4]. This community can be viewed as the one most typically considered in popular accounts of open source software projects. The GNU/Linux operating system environment is of course the largest, most complex, and most diverse sub-community within this arena, so much so that it merits separate treatment and examination. Many other Internet or Web infrastructure projects constitute recognizable communities or sub-communities of practice. The software systems that are the focus generally are not standalone end-user applications, but are often targeted at *system administrators* or *developers as the targeted user base,* rather than the eventual end-users of such systems. However, notable exceptions like Web browsers, news readers, instant messaging, and graphic image manipulation programs are growing in number within the end-user community

3.3 X-ray astronomy and deep space imaging

Participants in this community focus on the development and evolution of software systems supporting the Chandra X-Ray Observatory, the European Space Agency's XMM-Newton Observatory, the Sloan Digital Sky Survey, and others. These are three highly visible astrophysics research projects whose scientific discoveries depend on processing remotely sensed data through a complex network of open source software applications that process remotely sensed data [35]. In contrast to the preceding two development oriented communities, open software can play a significant role in scientific research communities. For example, when scientific findings or discoveries resulting from remotely sensed observations are reported[5], then members of the relevant scientific community want to be assured that the results are not the byproduct of

---

[2] *Unreal* begat *Half-Life* under a proprietary license, which gave rise to *Half-Life: CounterStrike*, the most popular FPS game (and game mod) at present [7]. *Counter-Strike* was developed and distributed as open source by two independent game player-developers. These developers were then financially engaged by Half-Life's commercial developer, Valve Software, to participate in the royalty stream generated from retail sales of *CounterStrike*, though the CS mod is still publicly accessible on the Web.

[3] It is reasonable to note that the two main software systems that enabled the World Wide Web, the NCSA Mosaic Web browser (and its descendants, like Netscape Navigator and Mozilla), and the Apache Web server (originally know as "HTTPd") were originally and still remain active open source software development projects.

[4] The SourceForge community web portal (http://www.sourceforge.net) currently stores information on more than 250K developers and 30K open source software development projects, with more than 10% of those projects indicating the availability of a mature, released, and actively supported software system.

[5] For example, see http://antwrp.gsfc.nasa.gov/apod/ap010725.html which displays a composite image constructed from both X-ray (Chandra Observatory) and optical (Hubble Space Telescope) sensors. The open software processing pipelines for each sensor are mostly distinct and are maintained by different organizations. However, their outputs must be integrated, and the images must be registered and oriented for synchronized overlay, pseudo-colored, and then composed into a final image, as shown on the cited Web page. There are dozens of open software programs that must be brought into alignment for such an image to be produced, and for such a scientific discovery to be claimed and substantiated [31, 35].

some questionable software calculation or opaque processing trick. In scientific fields like astrophysics that critically depend on software, open source is increasingly considered an essential precondition for research to proceed, and for scientific findings to be trusted and open to independent review and validation. Furthermore, as discoveries in the physics of deep space are made, this in turn often leads to modification or extension of the astronomical software in use in order to further explore and analyze newly observed phenomena, or to modify/add capabilities to how the remote sensing mechanisms operate.

3.4 Academic software systems design

Participants in this community focus on the development and evolution of software architecture and UML centered design efforts, such as for ArgoUML (http://argouml.tigris.org) or xARCH at CMU and UCI (http://www.isr.uci.edu/projects/xarch/). This community can easily be associated with a mainstream of software engineering research. People who participate in this community generally develop software for academic research or teaching purposes in order to explore topics like software design, software architecture, software design modeling notations, software design recovery (reverse software engineering), etc. Accordingly, it may not be unreasonable to expect that open software developed in this community should embody or demonstrate principles from modern software engineering theory or practice. Furthermore, much like the X-ray astronomy community, members of this community expect that when breakthrough technologies or innovations have been declared, such as in a refereed conference paper or publication in a scholarly journal, the opportunity exists for other community members to be able to access, review, or try out the software to assess and demonstrate its capabilities. An inability to provide such access may result in the software being labeled as "vaporware" and the innovation claim challenged, discounted, or rebuked. Alternatively, declarations of "non-disclosure" or "proprietary intellectual property" are generally not made for academic software, unless or until it is transferred to a commercial firm. However, it is often acceptable to find that academic software, whether open source or not, constitutes nothing more than a "proof of concept" demonstration or prototype system, not intended for routine or production use by end-users.

3.5 Community Characteristics

Each community is constituted by people who *identify themselves* with the development of open software within one of the four arenas noted above.[6] Though participants may employ pseudonyms (user-id's) in identifying themselves within a community, they do not assume nor rely on anonymous identifiers, as is found in other communities for socializing in cyberspace [Preece 2000, Smith and Kollock 1999]. Open software developers or contributors tend to act in ways where *building trust and reputation*, "*geek fame*", and *being generous with one's time, expertise, and source code* are valued traits of community participants [Pavlicek 2000]. They work to develop and contribute *software representations or content* (programs, design diagrams, execution scripts, code reviews, test case data, Web pages, email comments, etc.) to *Web sites* within each community. *Making contributions*, and *being recognized by other community members as having made substantive contributions*, is often a prerequisite for advancing technically and socially within a community [Fielding 1999, Kim 2000]. As a consequence, participants within these communities often participate in different *roles* within both *technical* and *social networks* [Smith and Kollock 1999, Preece 2000] in the course of developing, using, and evolving open software.

Administrators of open software community Web sites serve as *gatekeepers* in the choices they make for what information to post, when and where within the site to post it, as well as what not to post [36]. Similarly, they may choose to create a *site map* that constitutes a classification of *site and domain content,* as well as a geography of *community structure and boundaries* [4]. Community participants regularly use bboards to engage in *online discussion forums* or *threaded email messages* as a central way to observe, participate in, and contribute to public discussions of topics of interest [39]. However, these people also engage in *private online or offline discussions* that do not get posted or publicly disclosed, due to their perceived sensitive content. Finally, in each of the four communities examined here, participants choose on occasion to author and publish *technical reports or scholarly research papers* about their software development efforts, which are publicly available for subsequent examination, review, and secondary analysis.

---

[6] An alternative scheme for automatically discovering the membership of a Web-based community uses graph traversal (crawling linked web pages) and s-t maximum flow network algorithms [13].

Each of these highlighted items point to the public availability of data that can be collected, analyzed, and re-represented within narrative ethnographies [16, 20], computational process models [8, 24, 27, 34], or for quantitative studies [6, 21]. Significant examples of each kind of data have been collected and analyzed as part of this ongoing study. This paper includes a number of examples that serve as this data. Subsequently, we turn to review what requirements engineering is about, in order to establish a baseline of comparison for whether what we observe with the development of open software system requirements is similar or different, and if so how.

4. The classic software requirements engineering process

Experts in the field of software requirements engineering identify a recurring set of activities that characterize this engineering process [10, 17, 22, 28]. These activities, which are generally construed as necessary in order to produce a reliable, high quality, trustworthy system, include:

Eliciting requirements: identifies system stakeholders, stakeholder goals, needs, and expectations, and system boundaries. Elicitation techniques like questionnaire surveys, interviews, documentation review, focus groups, or joint application development (JAD) team meetings may be employed.

Modeling or specifying requirements: focuses attention to the systematic modeling of both functional and non-functional software requirements. One can model functional requirements of operational domain problems by specifying system processing states, events (input events, output events, process execution flags or signals, error detection, and exception handling triggers), and system data. Specifying system data may include identification of data objects, data types, data sources, end-user screen displays, and meta-data, as well as construction of a data dictionary. Functional requirements should also specify system data flow through system or subsystem states as controlled or synchronized by events. Beyond this, advanced modeling techniques may include construction of visual animations or simulated walkthroughs of overall system functionality. In contrast, one can model non-functional requirements as goals, capabilities, and constraints that situate the functional system within some context of operation. This can involve identifying an enterprise model, problem domain model, system model type, and data model type.

Analyzing requirements: entails a systematic reasoning of the internal consistency, completeness, or correctness of a specification. It does not check to see if the requirements are externally correct or an accurate model of the world. That determination may result from observing a visual animation of the specification during operational execution (a simulation). More sophisticated analyses may check for reachability, termination, live-lock and dead-lock, and safety in the modeled system.

Validating requirements: engages domain experts to assess feasibility of modeled system solution, as well as to identify realizable, plausible, and implausible system requirements. Systematic techniques for inspecting requirements to assess system usability and feasibility may also be employed. As a result of validation, the requirements engineer can better calibrate customer expectations about what can be developed.

Communicating requirements: entails documenting requirements, for example, through the creation of an software requirements specification (SRS) document, establishing criteria for requirements traceability, and managing the storage and evolution of the preceding requirements artifacts.

With these activities at hand, it is possible to consider whether this requirements engineering process captures or suitably characterizes what occurs in the development of requirements for open software systems, and how it occurs. The objective is not to establish conformity or distance metrics, nor some other quantitative measures that purport to reveal insights about the comparative quality of different open software systems, which might then be reused in other experimental situations [40]. Instead, the objective is to help determine whether the classic process adequately characterizes and covers what is observable in the development of open software requirements, or whether a new alternative model of requirements development for open software is needed.

Recently, experts in requirements engineering have begun to recognize the limits of the traditional requirements engineering process [28, 38]. They call for better modeling and analysis of the problem domain, as opposed to just focusing on the functional behavior of the software. They draw attention to the need to develop richer models for capturing and analyzing non-functional requirements. They also point to opportunities to bridge the gap between requirements elicitation techniques based on contextual and ethnographic techniques [15, 16, 38], and those techniques for formal specification and analysis. Thus, we can take into account these suggested improvements as well in our study.

5. Open software processes for developing requirements

In contrast to the world of classic software engineering, open software development communities do not seem to readily adopt or practice modern software engineering or requirements engineering processes. Perhaps this is no surprise. However, these communities do develop software that is extremely valuable, generally reliable, often trustworthy, and readily used within its associated user community. So, what processes or practices are being used to develop the requirements for open software systems?

From our study to date, we have found many types of software requirements activities being employed within or across the four communities. However, we have yet to find examples of formal requirements elicitation, analysis, and specification activity of the kind suggested by software requirements engineering textbooks [10, 22] in any of the four communities under study. Similarly, we have only found one example[7] online (in the Web sites) or offline (in published technical reports) of documents identified as "requirements specification" documents within these communities. However, what we have found is different.

5.1 Requirements elicitation vs. assertion of open software requirements

It appears that open software requirements are articulated in a number of ways that are ultimately expressed, represented, or depicted on the Web. On closer examination, requirements for open software can appear or be implied within an email message or within a discussion thread that is captured and/or posted on a project's Web site bboard for open review, elaboration, refutation, or refinement. Consider the following example found on the Web site for the KDE system (http://www.kde.org/), within the Internet/Web Infrastructure community. This example displayed in Exhibit 1[8] reveals *asserted capabilities* for the Qt3 subsystem within KDE. These capabilities (identified in the exhibit as the "Re: Benefits of Qt3?" discussion thread) highlight implied requirements for multi-language character sets (Arabic and Hebrew, as well as English), database support ("…there is often need to access data from a database and display it in a GUI, or vice versa…"), and others. These requirements are simply asserted without reference to other documents, sources, standards, or JAD focus groups--they are requirements because some developers wanted these capabilities.

Asserted system capabilities are *post-hoc* requirements characterizing a functional capability that has already been implemented. The concerned developers justify their requirements through their provision of the required coding effort to make these capabilities operational. Senior members or core developers in the community then vote or agree through discussion to include the asserted capability into the system's distribution [12]. The historical record may be there, within the email or bboard discussion archive, to document who required what, where, when, why, and how. However, once asserted, there is generally no

[7] "Software Requirements Specifications for the Central Manager (DDNS_CM) CSCI of the Distributed Data Network System (DDNS)", 29 July 2001, http://www.sourceforge.net/projects/ddns. It seems fair to observe that this SRS seems to follow guidelines embodied in military standards for software development, perhaps suggesting its origination in an industrial firm.
[8] Each exhibit appears as a screenshot of a Web browsing session. It includes contextual information, following the second research principle, thus requiring and benefiting from a more complete display view.

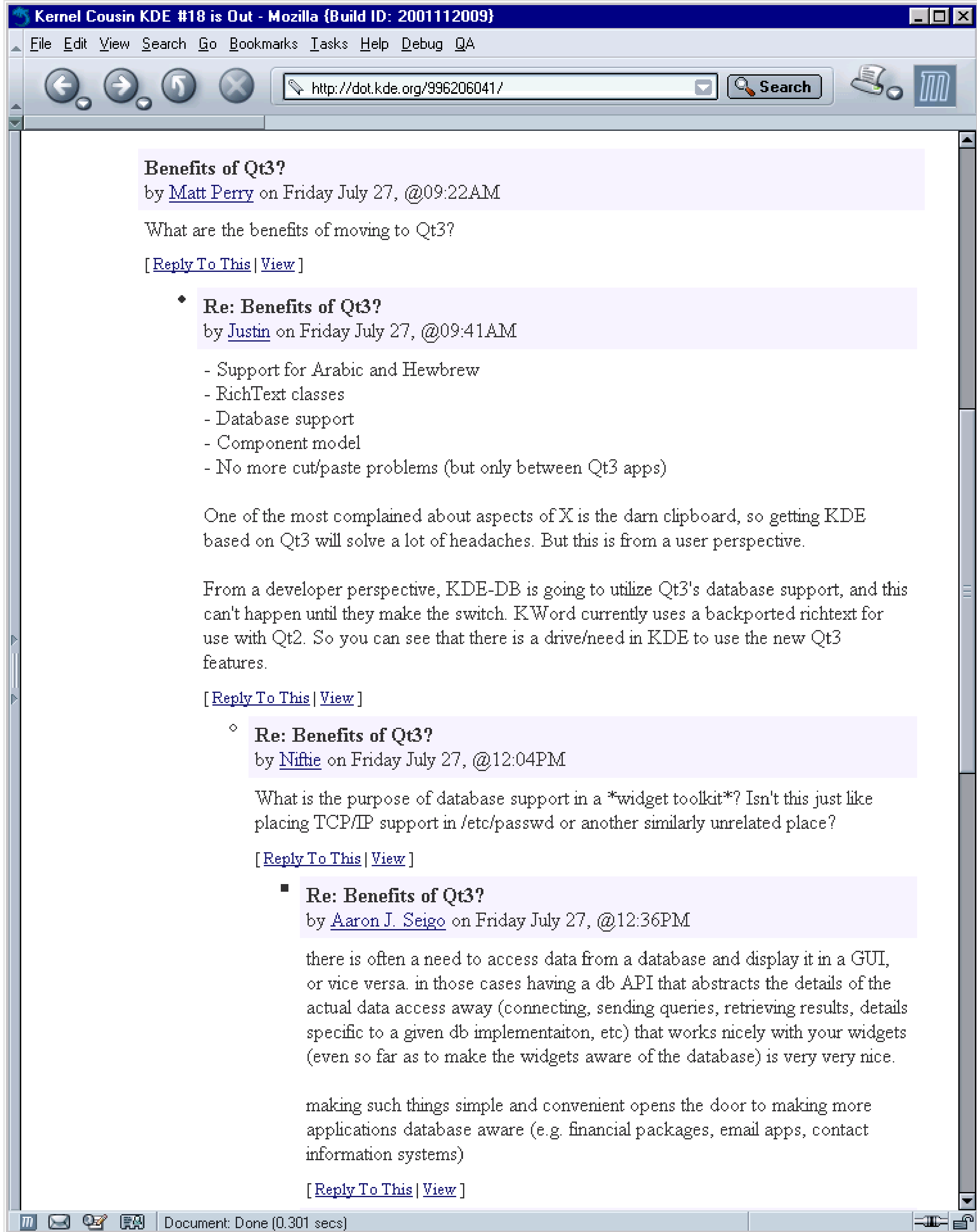


Exhibit 1. A sample of implicit requirements for the KDE software subsystem Qt3 expressed in a threaded email discussion. Source: http://dot.kde.org/996206041/, July 2001.

further effort apparent to document, formalize, or substantiate such a capability as a system requirement. Asserted capabilities then become invisible or transparent, taken-for-granted requirements that can be labeled or treated as *obvious* (i.e., a shared awareness) to those familiar with the system's development.

Another example reveals a different kind open software requirement. This case, displayed in Exhibit 2,[9] finds a requirements "vision" document that conveys a non-functional requirement for "community software development" in the bottom portion of the exhibit. This can be read as a non-functional requirement for the system's developers to embrace community software development as the process to develop and evolve the ArgoUML system, rather than say through a process which relies on the use of system models represented as UML diagrams [cf. 38]. Perhaps community software development, and by extension, community development, are recognized as being important to the development and success of this system. It may also be a method for improving system quality and reliability when compared to existing software engineering tools and techniques (i.e., those based on UML, or supporting UML-based software design).

A third example reveals yet another kind of elicitation found in the Internet/Web infrastructure community. In Exhibit 3, we see an overview of the `mono` project. Here we see multiple statements for would-be software component/class owners to sign-up and commit to developing the required ideas, run-time, (object service) classes, and projects. These are non-functional requirements for people to volunteer to participate in community software development, in a manner perhaps compatible with that portrayed in Exhibit 2. The systems in Exhibits 2 and 3 must also be considered early in their overall development or maturity, since they call for functional capabilities that are needed to help make sufficiently complete for usage.

Thus, in understanding how the requirements of open software systems are elicited, we find evidence for elicitation of volunteers to come forward to participate in community software development. A similar example inviting new participants into the world of game mods appears in Exhibit 4. We also observe the assertion of requirements that simply appear to exist without question or without trace to a point of origination, rather than somehow being elicited from stakeholders, customers, or prospective end-users of open software systems. As previously noted, we have not yet found evidence or data to indicate the occurrence or documentation of a requirements elicitation effort arising in an open software development project. However, finding such evidence would not invalidate the other observations; instead, it would point to a need to broaden the scope of how software requirements are captured or recorded.

5.2 Requirements analysis vs. requirements reading, sense-making, and accountability

In open software development, how does requirements analysis occur, and where and how are requirements specifications described? Though requirements analysis and specification are interrelated activities, rather than distinct stages, we first consider examining how open software requirements are analyzed.
Exhibits 5 and 6 come from different points in the same source document, a single research paper accessible on the Web, associated with the Chandra X-ray Center Data System (CXCDS) for sensing and imaging deep space (astronomical) objects that radiate in the X-ray spectrum. Exhibit 5 suggests to the reader that the requirements for the CXCDS are involved and complex (as seen in the "Abstract"), and Exhibit 6 seems to confirm this claim, as least to an outsider interpreting Figure 2 shown in the exhibit. As a data-flow diagram, Exhibit 6 either suggests or denotes part of the specification of requirements for the CXCDS. But how do software developers in this community (astrophysicists) understand what's involved in the functional operation of a complex system like this? One answer lies in the observation that developers who seek such an understanding must read this research paper quite closely, as well as being able to draw on their prior knowledge and experience in the relevant physical, telemetric, digital, and software domains. A close reading likely means one that entails multiple re-readings and sense-making relative to one's expertise and prior development experience [cf. 1]. A more casual though competent reading requires some degree of confidence and trust in the authors' account of how the functionality of the

[9] The ArgoUML tool [33] within the academic software design community at http://argouml.tigris.org/.

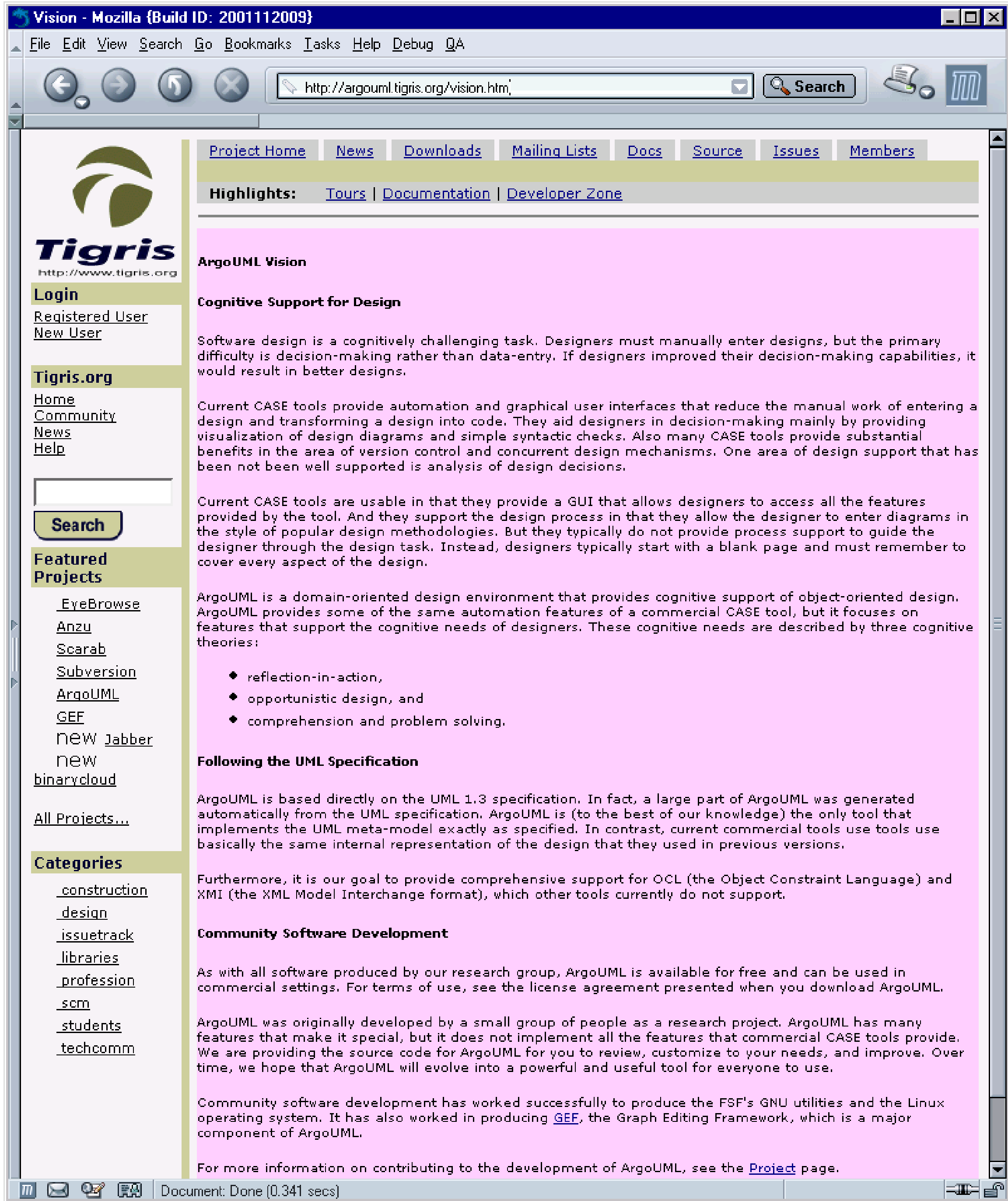


Exhibit 2. A software requirements vision statement highlighting community development as a software development objective (i.e., a non-functional requirement). Source: http://argouml.tigris.org/vision.html, July 2001.

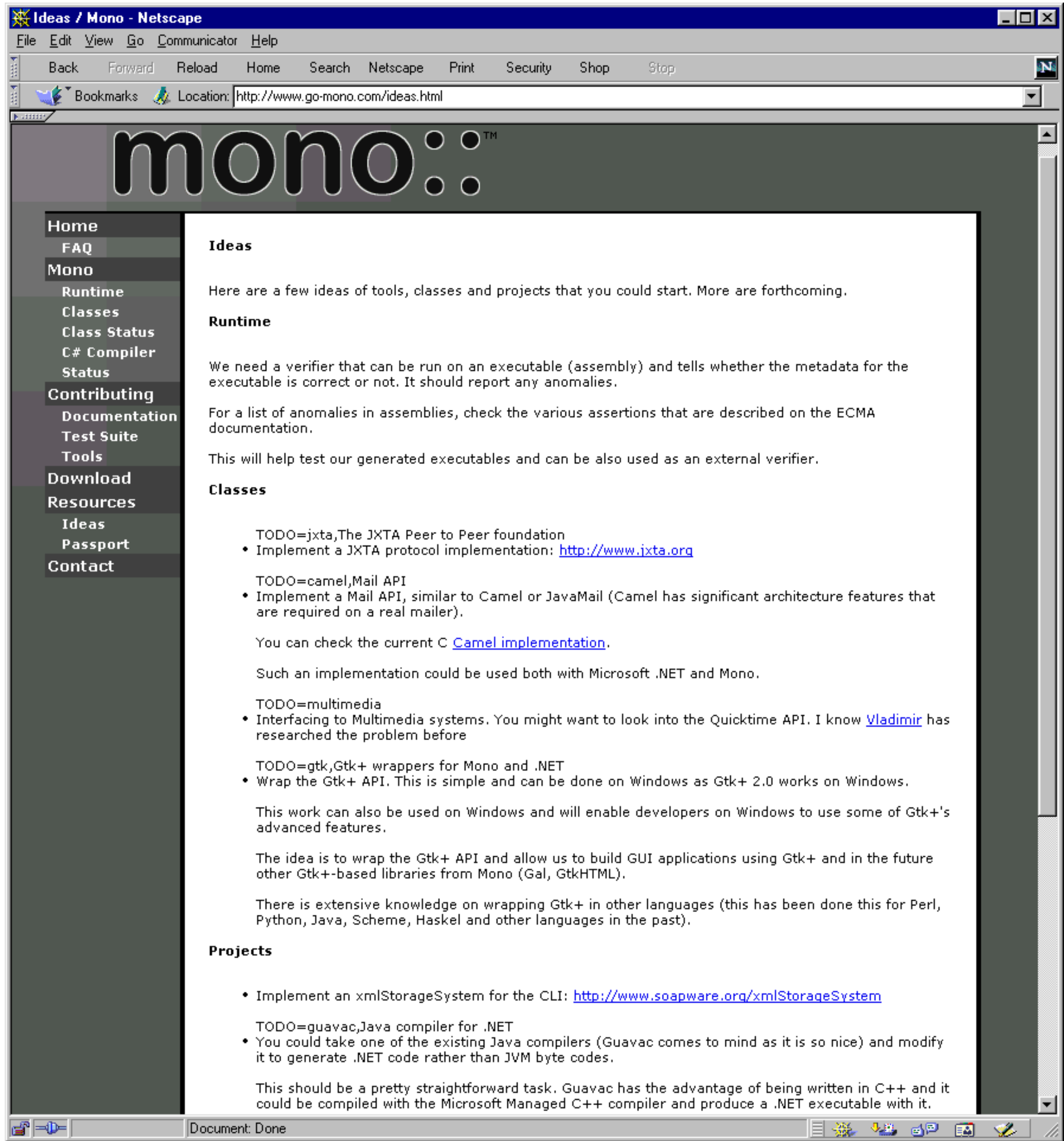


Exhibit 3: A non-functional requirement identifying a need for volunteers to become owners for software components (or classes) not yet bound to a developer. Source: http://www.go-mono.com/ideas.html, July 2001. The contents on this page have since been updated to reflect remaining tasks needing contributors, as well as adding new tasks for development.

Exhibit 4. An asserted capability (in the center) that invites would-be open software game developers to make extensions of whatever kind they require among the various types of available extensions ("…create your own levels, mods, skins, models, and more"). Source: http://www.unrealtournament.com/editing

The Chandra Automatic Data Processing Infrastructure - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Forward Stop Refresh Home Search Favorites History Mail Size Print Edit

Address http://adass.org/adass/proceedings/adass00/P1-32/ Go Links

The Chandra Automatic Data Processing Infrastructure

David Plummer and Sreelatha Subramanian
Harvard-Smithsonian Center for Astrophysics, 60 Garden St. MS-81, Cambridge, MA 02138

Abstract:

The requirements for processing Chandra telemetry are very involved and complex. To maximize efficiency, the infrastructure for processing telemetry has been automated such that all stages of processing will be initiated without operator intervention once a telemetry file is sent to the processing input directory. To maximize flexibility, the processing infrastructure is configured via an ASCII registry. This paper discusses the major components of the Automatic Processing infrastructure including our use of the STScI OPUS system. It describes how the registry is used to control and coordinate the automatic processing.

1. Introduction

Chandra data are processed, archived, and distributed by the Chandra X-ray Center (CXC). Standard Data Processing is accomplished by dozens of ``pipelines'' designed to process specific instrument data and/or generate a particular data product. Pipelines are organized into levels and generally require as input the output products from earlier levels. Some pipelines process data by observation while others process according to a set time interval or other criteria. Thus, the processing requirements and pipeline data dependencies are very complex. This complexity is captured in an ASCII processing registry which contains information about every data product and pipeline. The Automatic Processing system (AP) polls its input directories for raw telemetry and ephemeris data, pre-processes the telemetry, kicks off the processing pipelines at the appropriate times, provides the required input, and archives the output data products.

2. CXC Pipelines

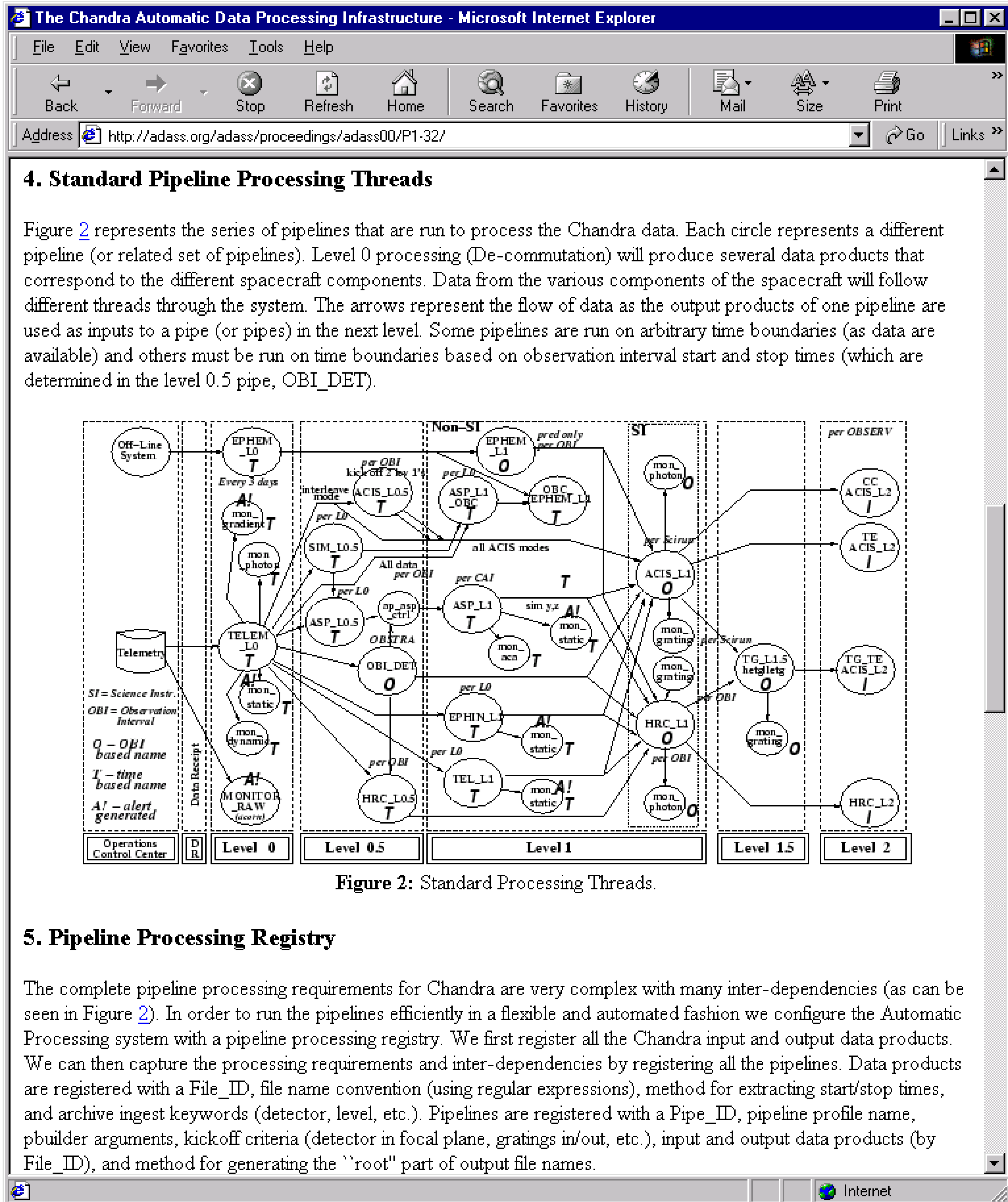
A CXC pipeline is defined by an ASCII profile template that contains a list of tools to run and the associated run-time parameters (e.g., input/output directory, root-names, etc.). When a pipeline is ready to run, a pipeline run-time profile is generated by the profile builder tool, pbuilder. The run-time profile is executed by the Pipeline Controller, pctr. The pipeline profiles and pctr support conditional execution of tools, branching and converging of threads, and logfile output containing the profile, list of run-time tools, arguments, exit status, parameter files, and run-time output. This process is summarized in Figure 1.

Pipeline Profile Template
Run–time arguments
pbuilder
Run–time Profile
Input data
pctr
Output data
Log file

Figure 1: The CXC Pipeline Processing Mechanism.

Internet

Exhibit 5. An asserted capability indicating that the requirements are very involved and complex, and thus require an automated, registry-based system software architecture for configuring dozens of application software pipelines. Source: [31].



The Chandra Automatic Data Processing Infrastructure - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Forward Stop Refresh Home Search Favorites History Mail Size Print

Address http://adass.org/adass/proceedings/adass00/P1-32/ Go Links

4. Standard Pipeline Processing Threads

Figure 2 represents the series of pipelines that are run to process the Chandra data. Each circle represents a different pipeline (or related set of pipelines). Level 0 processing (De-commutation) will produce several data products that correspond to the different spacecraft components. Data from the various components of the spacecraft will follow different threads through the system. The arrows represent the flow of data as the output products of one pipeline are used as inputs to a pipe (or pipes) in the next level. Some pipelines are run on arbitrary time boundaries (as data are available) and others must be run on time boundaries based on observation interval start and stop times (which are determined in the level 0.5 pipe, OBI_DET).

Figure 2: Standard Processing Threads.

5. Pipeline Processing Registry

The complete pipeline processing requirements for Chandra are very complex with many inter-dependencies (as can be seen in Figure 2). In order to run the pipelines efficiently in a flexible and automated fashion we configure the Automatic Processing system with a pipeline processing registry. We first register all the Chandra input and output data products. We can then capture the processing requirements and inter-dependencies by registering all the pipelines. Data products are registered with a File_ID, file name convention (using regular expressions), method for extracting start/stop times, and archive ingest keywords (detector, level, etc.). Pipelines are registered with a Pipe_ID, pipeline profile name, pbuilder arguments, kickoff criteria (detector in focal plane, gratings in/out, etc.), input and output data products (by File_ID), and method for generating the ``root'' part of output file names.

Internet

Exhibit 6. A specification of data-flow relationships among a network of software module pipelines that constitute the processing threads that must be configured in order to transform remotely sensed telemetry data into digital images of deep space objects. Source: [31]

CXCDS architecture is configured, in order to accept what is presented as plausible, accurate, and correct.

The notion that requirements for open software system are, in practice, analyzed via the reading of technical accounts as narratives, together with making sense of how such readings are reconciled with one's prior knowledge, is not unique to the X-ray astronomy software community. These same activities can and do occur in the other three communities. If one reviews the functional and non-functional requirements appearing in Exhibits 1-4, it is possible to observe that none of the descriptions appearing in these exhibits is self-contained. Instead, each requires the reader (e.g., a developer within the community) to closely or casually read what is described, make sense of it, consult other materials or one's expertise, and trust that the description's author(s) are reliable and accountable in some manner for the open software requirements that have been described [15, 29]. Analyzing open software requirements entails little if any automated analysis, formal reasoning, or visual animation of software requirements specifications [cf. 28]. Yet, participants in these communities are able to understand what the functional and non-functional requirements are in ways that are sufficient to lead to the ongoing development and routine use of various kinds of open software systems.

5.3 Requirements specification and modeling vs. continually emerging webs of software discourse

If the requirements for open software systems are asserted rather than elicited, how are these requirements specified or modeled? In examining data from the four communities, it is becoming increasingly apparent that open software requirements can emerge from the experiences of community participants through their email and bboard discussion forums (see Exhibit 1 for example). These communication messages in turn give rise to the development of narrative descriptions that more succinctly specify and condense into a web of discourse about the functional and non-functional requirements of an open software system. This discourse is rendered in descriptions that can be found in email and discussion forum archives, on Web pages that populate community Web sites, and in other informal software descriptions that are posted, hyperlinked, or passively referenced through the assumed common knowledge that community participants expect their cohorts to possess.

In Exhibit 5 from the X-ray and deep space imaging software community, we see passing reference in the opening paragraph to "the requirements for processing Chandra telemetry (imaging data) are very involved and complex." To comprehend and recognize what these involved and complex requirements are, community members who develop open software for such applications will often be astrophysicists (generally with Ph.D. degrees), and rarely would be simply a competent software engineering professional. Subsequently, the astrophysicists that develop software in this community do not need to recapitulate any software system requirement that would be due to the problem domain (astrophysics). Instead, community members are already assumed to have mastery over such topics prior to software development, rather than encountering problems in their understanding of astrophysicists arising from technical problems in developing, operation, or functional enhancement of remote sensing or digital imaging software. Thus, for openness to be effective, a software developer in this community must be competent in the discourse of astrophysics, as well as with that for the tools and techniques used in developing open software systems.

Subsequently, spanning the four communities and the six exhibits, we begin to observe that the requirements for open software are specified in webs of discourse that reference or link:

- email or bboard discussion threads,
- system vision statements,
- ideas about system functionality and the non-functional need for volunteer developers to implement the functionality,
- promotional encouragement to specify and develop whatever functionality you need, which might also help you get a new job, and
- scholarly scientific research publications that underscore how the requirements of domain-specific software (e.g., for astronomical imaging), though complex, are understood without elaboration, since they rely on prior scientific/domain knowledge and tradition of open scientific research.

Each of these modes of discourse, as well as their Web-based specification and dissemination, is a continually emerging source of open software requirements from new contributions, new contributors or participants, new ideas, new career opportunities, and new research publications [cf. 37].

5.4 Requirements validation vs. condensing discourse that hardens and concentrates system functionality and community development

Software requirements are validated with respect to the software's implementation. Since open software requirements are generally not recorded in a formal SRS document, nor are these requirements typically cast in a mathematical logic, algebraic, or state transition-based notational scheme, then how are the software implementations to be validated against their requirements?

In each of the four communities, it appears that the requirements for open software are co-mingled with design, implementation, and testing descriptions and software artifacts, as well as with user manuals and usage artifacts (e.g., input data, program invocation scripts). Similarly, the requirements are spread across different kinds of electronic documents including Web pages, sites, hypertext links, source code directories, threaded email transcripts, and more. In each community, requirements are described, asserted, or implied informally. Yet it is possible to observe in threaded email/bboard discussions that community participants are able to comprehend and condense wide-ranging software requirements into succinct descriptions using lean media [39] that pushes the context for their creation into the background. Goguen [15] suggests the metaphor of "concentrating and hardening of requirements" as a way to characterize how software requirements evolve into forms that are perceived as suitable for validation. His characterization seems to quite closely match what can be observed in the development of requirements for open software. Subsequently, we find that requirements validation is an implicit by-product, rather than an explicit goal, of how open software requirements are constituted, described, discussed, cross-referenced, and hyperlinked to other informal descriptions of system and its implementations.

5.5 Communicating requirements vs. global access to open software webs

One distinguishing feature of open software associated with each of the four communities is that their requirements, informal as they are, are organized and typically stored in a persistent form that is globally accessible. This is true of community Web sites, site contents and hyperlinkage, source code directories, threaded email and bboard discussion forums, descriptions of known bugs and desired system enhancements, records of multiple system versions, and more. Persistence, hypertext-style organization and linkage, and global access to open software descriptions appear as conditions that do not receive much attention within the classic requirements engineering approaches, with few exceptions [9]. Yet, each of these conditions helps in the communication of open software requirements. These conditions also contribute to the ability of community participants or outsiders looking in to trace the development and evolution of software requirements both within the software development descriptions, as well as across community participants. This enables observers or developers to navigationally trace, for example, a web of different issues, positions, arguments, policy statements, and design rationales that support (e.g., see Exhibit 1) or challenge the viability of emerging software requirements [cf.5, 23]. Nonetheless, these traces appear to lack a persistent representation beyond the awkward "history" file of a Web browser.

Each of the four communities also communicates community-oriented requirements. These non-functional requirements may seem similar to those for enterprise modeling [28]. However, there are some differences, though they may be minor. First, each community is interested in sustaining and growing the community as a development enterprise [cf. 26]. Second, each community is interested in sustaining and growing the community's open software artifacts, descriptions, and representations. Third, each community is interested in updating and evolving the community's information sharing Web sites. In recognition of these community requirements, it is not surprising to observe the emergence of commercial efforts (e.g., SourceForge and CollabNet) that offer community support systems that are intended to address these requirements, such as is used in the ArgoUML community site, http://www.tigris.org, in the academic software design community.

5.6 Identifying a common foundation for the development of open software requirements

Based on the data and analysis presented above, it is possible to begin to identify what items, practices, or capabilities may better characterize how the requirements for open software are developed. This centers of the emergent creation, usage, and evolution of informal software descriptions as the vehicle for developing open software requirements. This is explored in the following section.

6. Informalisms for Open Software System Requirements

The functional and non-functional requirements for open software systems are elicited, analyzed, specified, validated, and managed through a variety of Web-based descriptions. These descriptions can be treated collectively as *software informalisms.* The choice to designate these descriptions as informalisms[10] is to draw a distinction between how the requirements of open software systems are described, in contrast to the recommended use of formal, logic-based requirements notations ("formalisms") that are advocated in traditional approaches [10, 17, 22, 28]. In the four communities examined in this study, software informalisms appear to be the preferred scheme for describing or representing open software requirements. There is no explicit objective or effort to treat these informalisms as "informal software requirements" that should be refined into formal requirements [9, 17, 22] within any of these communities. Accordingly, we can present an initial classification scheme that inventories the available types of software requirements informalisms that have been found in one or more of the four communities in this study. Along the way, we seek to identify some of the relations that link them together into more comprehensive stories, storylines, or intersecting story fragments that help convey as well as embody the requirements of an open source software system.

Eight types of software informalisms can be identified, and each has sub-types that can be identified as follows.

6.1 Community communications

The requirements for open software are asserted, read, discussed, condensed, and made accountable through a small set of computer-based communication tools and modalities. In the absence of co-located workplaces, a community's communication infrastructure serves as the "place" where software requirements engineering work is performed, and where requirements artifacts are articulated, refined, stored, or discarded. These communication systems, appear in the form of: (a) messages placed in a Web-based bboard discussion forums; (b) email list servers; (c) network news groups; or less frequently in (d) Internet-based chat (instant messaging)[11]. Messages written and read through these systems, together with references or links to other messages or software webs, then provide some sense of context for how to understand messages, or where and how to act on them.

6.2 Scenarios of usage as linked Web pages

Open software developers who do not meet face-to-face create, employ, read, and revise shared mental constructions of how a given system is suppose to function. Since shared understanding must occur at a distance in space or time, then community participants create artifacts like screenshots, guided tours, or navigational click-through sequences (e.g., "back", "next" Web page links) with supplementary narrative descriptions in attempting to convey their intent or understanding of how the system operates, or how it appears to a user when used. This seems to occur when participants find it simpler or easier to explain what is suppose to happen or be observable at the user interface with pictures (or related hypermedia) than with just words. Similarly, participants may publish operational program execution scripts or recipes for how to develop or extend designated types of open software artifacts. These hypermedia scenarios of usage may serve a similar purpose to formally elicited and modeled Use Cases, though there is no apparent effort to codify these usage scenarios in such manner or notational form in any of the communities in this study.

---

[10] As Goguen [15] observes, formalisms are not limited to those based on a mathematical logic or state transition semantics, but can include descriptive schemes that are formed from structured or semi-structured narratives, such as those employed in Software Requirements Specifications documents.
[11] Instant messaging can be more widely observed in the networked computer game community in contrast to the academic software design and X-ray astrophysics community, where we are yet to find traces of instant messaging activities in support of open software development.

6.3 HowTo Guides

Online documents that capture and condense "how to" perform some behavior, operation, or function with a system, serve as a semi-structured narrative that assert or imply end-user requirements. "Formal" HowTo's descriptions include explicit declarations of their purpose as a HowTo and may be identified as a system tutorial. Community participants may seek these formal HowTo's when they need to add a system module or class structure, or contribute other resources or efforts to the open software project. In contrast, informal HowTo's may appear as a selection, composition, or recomposition of any of the proceeding. These informal HowTo guides may be labeled as a "FAQ"; that is, as a list of frequently asked questions about how a system operates, how to use it, where to find it's development status, who developed what, known bugs and workarounds, etc. However, most FAQs do not indicate how frequently any of the questions may have been asked, or if effort has been made to measure or track FAQ usage/reference.

6.4 External Publications

In each of the four communities in this study, there are external publications that describe open software available for consumption by the public or by community members. Most common among these are technical articles, while books are less common, though of growing popularity in the Internet/Web infrastructure and computer game community. Many developers find that books, especially those derived from composition and extension of other open software informalisms, are both a valuable and convenient source for recording, recontextualizing, and explaining the functional and non-functional requirements of an open software system, or how it was developed [e.g., 11, 14, 32].

On the other hand, professional articles that inform interested readers or promote the author's interests in a certain open software technology, help identify general functional and non-functional software requirements for these systems. These article may appear in trade publications, like the *Linux Journal* or *Game Developer* [7]. Academic articles that are refereed and appear in conference proceedings or scholarly journals [25, 33, 35], serve a similar purpose as professional articles, though usually with more technical depth, theoretical recapitulation, analytical detail, and extensive bibliography of related efforts. However, it may be the case that readers of academic research papers bring to their reading a substantial amount of prior domain knowledge. This expertise may enable them to determine what open software requirements being referenced may be obvious from received wisdom, versus those requirements that are new, innovative, or otherwise noteworthy.

6.5 Open Software Web Sites and Source Webs

As already suggested, open software is most easily found on the Web or Internet. Neither information infrastructure is an absolute necessity for open software. However, such global infrastructure is an enabler of open software communities, processes, and practices. But open software communities take advantage of a community Web site as an information infrastructure for publishing and sharing open descriptions of software in the form of Web pages, Web links, and software artifact content indexes or directories. These pages, hypertext links, and directories are community information structures that serve as a kind of organizational memory and community information system. Such a memory and information system records, stores, and retrieves how open software systems and artifacts are being articulated, negotiated, employed, refined, and coordinated within a community of collaborating developer-users [5, 23, 1].

Web pages in each of the four open software communities include *content* that incorporates text, tables or presentation frames, diagrams, or navigational images (image maps) to describe their associated open software systems. This content may describe vision statements, assert system features, or otherwise characterize through a narrative, the functional and non-functional capabilities of an open software system. Whether this content can be considered a software requirements specification document is unclear, since to do so would seem to allow almost any document with a narrative about some software system to be considered as an SRS document, and thus a formal requirements specification.

Web content that describes an open software system often comes with many embedded Web *links*. These links associate content across Web pages, sites, or applications. These links may simply denote traversal links (i.e., "goto" links to related software components, for example, go to the source code for a named procedure call) or other source/Web content where there is no further explicit meaning or semantics

assigned to the link. Alternatively, they may serve as enumerated navigational index links (e.g., site indexes) that helps direct a community participant (or outsider) to find their way around the community, its community information base, and the community's open source code base. Beyond this, they can serve as links to implied "helper applications" or tools invoked by navigational access to remotely served file types that are registered on the client, and associated with external application programs or plug-in programs that are invoked when selected or traversed [26, 27].

Each of the open software communities in this study provides access to Web-based source code directories, files, or compositions for download, build and/or installation. These directories and files contain operational software or open source code, as well as some related support files or Web pages. Source code denote requirements implementations, rather than requirements specifications. Program execution scripts, which take the form of C-shell, Tcl, Perl, Python, or Java scripts on Linux/Unix systems, may be employed for invoking other system modules. These execution scripts are functional in the sense that they invoke or cause system behavior that is implemented in the open software source code. Since scripts are generally platform specific, they effectively impose their own functional requirements on an implemented system. Thus, it is not surprising to find examples in each community for Web pages or files that describe these requirements explicitly, while the requirements of the open software system whose behavior in under control of the script has its requirements left implicit. The same kinds of concerns and explication of functional requirements is also found for make files (compilation or build scripts), CVS files that specify, control and synchronize concurrent software versions, and deployment-installation compositions (e.g., "tarballs" or zip files) for coordinating shared software production and distribution [14].

6.6 Software bug reports and issue tracking

One of the most obvious and frequent types of discourse that appears with open software systems is discussion about operational problems with the current version of the system implementation. Bugs and other issues (missing functionality, incorrect calculation, incorrect rendering of application domain constructs, etc.) are common to open software, much like they are with all other software. However, in an open software development situation, community participants rely on lean communication media like email, bug report bboards, and related issue tracking mechanisms to capture, rearticulate, and refine implicit, mis-stated, or unstated system requirements [39]. We find the capabilities of bug report or issue-tracking systems like Bugzilla in the Internet/Web infrastructure community, are also appearing in the academic software design community and networked game communities. In contrast, software developers in X-ray astrophysics community still rely on threaded email discussion lists to manage their (re) emerging requirements, often with relatively few message postings.

6.7 Traditional software system documentation

Open software systems are not without online system documentation or documentation intended to be printed in support of end-users or developers. For all of the systems examined in this study, it was possible to locate online man pages or help pages that describe commands and command parameters for how to invoke or use a system. Similarly, it was possible to locate online user manuals for most of these systems. It was apparent in both situations that online documentation was usually dated, and subsequently inconsistent with current functional capabilities or system commands.

This may just be what should be expected of both closed and open software systems. However, it is apparent that there are many other information resources, that is, the other software informalisms, that are available to developers and end-users to help them detect or resolve inconsistencies in such documentation.

The overall set of software informalisms serve as a context for reconstructing what a system's functional requirements were, are, or can be. Thus, while open software manuals are not necessarily any better or worse than for other software, the context for their use may enable the typical inconsistencies one encounters to be more readily resolved. Subsequently, there are few incentives to make online manuals or help files for open software systems any better than the minimum needed to assist an unfamiliar user or developer to get started, or where to look for further help. Good enough documentation is good enough.

6.8 Software extension mechanisms and architectures

The developers of software systems in each of the four communities seek to keep their systems open through provision of a variety of extension mechanisms and architectures. These are more than just open application program interfaces (APIs); generally they represent operational mechanisms or capabilities. The extensions include embedded scripting languages, such as UnrealScript for *Unreal Tournament*; and Perl/Python for Internet/Web infrastructure applications. Open software architectures accommodate operational plug-in modules, as in the case for the Apache web server and Chandra system infrastructure. Other open architectural schemes accommodate reconfigurable processing pipelines, like the Chandra Data Processing Infrastructure. Finally, in the networked computer game community, we see game vendors providing tools and utilities to assist advanced users so that they can develop their own extensions or custom programs in order to keep an open software system alive and continuously evolving.

Whether these mechanisms and architectures can or should be treated as software formalisms is perhaps subject to debate. However, across the four community, it is apparent that software extensions mechanisms and extensible software architectures contribute to, as well as enable, the continuing emergence open software requirements.

Overall, it appears that none of these software informalisms would defy an effort to formalize them in some mathematical logic or analytically rigorous notation. Nonetheless, in the four software communities examined in this study, there is no perceived requirement for such formalization, nor no unrecognized opportunity to somehow improve the quality, usability, or cost-effectiveness of the open software systems, that has been missed. If formalization of these software benefits has demonstrable benefit to members of these communities, beyond what they already realize from current practices, these benefits have yet to be articulated in the discourse that pervades each community.

7. Understanding open software requirements

In open software development projects, requirements engineering efforts are *implied activities* that routinely emerge as a by-product of community discourse about what their software should or should not do, as well as who will take responsibility for realizing such requirements. Open software system requirements appear in the form of situated discourse within private and public email discussion threads, emergent artifacts (e.g., source code fragments included within a message) and dialectical social actions that negotiate interest, commitment, and accountability [15, 37]. More simply, traditional requirements engineering activities do not have first-class status as an assigned or recognized task within open software development communities. Similarly, there are no software engineering tools used to support the capture, negotiation, and cost estimate (e.g., level of effort, expertise/skill, and timeliness) of open software development efforts, though each of these activities occurs regularly but informally.

Open software systems may be very reliable and high quality in their users' assessments. Nonetheless requirements do exist, though finding or recognizing them demands familiarity and immersion within the community and its discussions. This of course stands in contrast to efforts within the academic software engineering or requirements engineering community to develop and demonstrate tools for explicitly capturing requirements, negotiating trade-offs among system requirements and stakeholder interests, and constructive cost estimation or modeling [e.g., 3]. Furthermore, in open software systems, the developers are generally end-users of the systems they develop, whereas in traditional software requirements engineering efforts, developers and users are distinct, and developers tend not to routinely use the systems they develop. Perhaps this in why open software systems can suffice with reliance on software informalisms, while traditional software engineering efforts must struggle to convert informal requirements into more formal ones.

Developing open software requirements is a *community building process* that must be institutionalized both within a community and its software informalisms to flourish [30, 36]. In this regard, the development of requirements for open software is not a traditional requirements engineering process, at least, not yet. It is instead socio-technical process that entails the development of constructive social relationships, informally negotiated social agreements, and a commitment to participate through sustained contribution of software discourse and shared representations. Thus, community building and sustaining participation are essential

and recurring activities that enable open software requirements and system implementation to emerge and persist without central corporate authority.

Open software Web sites serve as hubs that centralize attention for what is happening with the development of the focal open software system, its status, participants and contributors, discourse on pending/future needs, etc. Furthermore, by there very nature, open software Web sites (those accessible outside of a corporate firewall) are generally global in reach and accessibility. This means the potential exists for contributors to come from multiple remote sites (geographic dispersion) at different times (24/7), from multiple nations, potentially representing the interests of multiple cultures or ethnicity. All of these conditions point to new kinds of requirements—for example, community building requirements, community software requirements, and community information sharing system (Web site and interlinked communication channels for email, forums, and chat) requirements. These requirements may entail both functional and non-functional requirements, but they will most typically be expressed using open software informalisms, rather than using formal notations based on some system of mathematical logic.

8. Conclusions

The paper reports on a study that investigates, compares, and describes how the requirements engineering processes occurs in open source software development projects found in different communities. A number of conclusions can be drawn from the findings presented.

First, this study sought to discover and describe the practices and artifacts that characterize how the requirements for developing open software systems are developed. Perhaps the processes and artifacts that were described were obvious to the reader. This might be true for those scholars and students of software requirements engineering who have already participated in open software projects. However, advocates of open source software do not identify or report on the processes described here [11, 29, 32]. Thus, we must ask what is obvious to whom, and on what source of knowledge or experience is it based? For the majority of students who have not participated, it is disappointing to not find such descriptions, processes, or artifacts within the classic or contemporary literature on requirements engineering [10, 17, 22, 28]. In contrast, this study sought to develop a baseline characterization of the how the requirements process for open software occurs and the artifacts (and other mechanisms).

Given such a baseline of the "as-is" process for open software requirements engineering, it now becomes possible to juxtapose one or more "to-be" prescriptive models for the requirements engineering process, then begin to address what steps are needed to transform the as-is into the to-be [34]. Said differently, what is the process that gets open software development from "here to there", from the as-is to the to-be? Such a position provides a basis for further studies which could examine how to redesign open software practices into those closer to that advocated by classic or contemporary scholars of software requirements engineering. This would enable students or scholars of software requirements engineering, for example, to determine whether or not open source software development would benefit from more rigorous requirements elicitation, analysis, and management, and if so, how. Similarly, it might help determine when a software requirements process that relies on the use of informalisms will be more/less effective than one rooted in formalisms.

Second, this report describes a new set of processes that constitute how open software requirements are developed or engineered in the form of a narrative model (cf. Section 5). We can therefore begin a follow-on step to develop a more comprehensively detailed model of these processes [20, 38], which in turn can be further analyzed or simulated if codified as a computational process model [24, 27, 34]. Such a process model could then be aligned and combined with those for traditional requirements engineering (cf. Section 4). Such an integration of processes would broaden the scope of requirements engineering to include open software development, not as an example of poor quality software engineering, but as an alternative to the dominant tradition advocated by contemporary software requirements experts and scholars. Clearly, the development of open software systems entails social and technical relations that differ from those advocated within traditional software or requirements engineering texts. It is unclear what kinds of software systems are most amenable to an open source approach, and which still seem to require the struggle to formalize traditional software requirements specifications.

Third, this study reports on the centrality and importance of open software requirements processes and software informalisms to the development of open software systems, projects, and communities. This result might be construed as an advocacy of the 'informal' over the 'formal' in how software system requirements are or should be developed and validated, though it is not so intended. Instead, attention to software informalisms used in open software projects, without the need to coerce or transform them into more mathematically formal notations, raises the issue of what kinds of *engineering virtues* should be articulated to evaluate the quality, reliability, or feasibility of open software system requirements so expressed. For example, traditional software requirements engineering advocates the need to assess requirements in terms of virtues like consistency, completeness, traceability, and correctness [10, 17]. From the study presented here, it appears that open software requirements artifacts might be assessed in terms of virtues like encouragement of community building; freedom of expression and multiplicity of expression with software informalisms; readability and ease of navigation; and implicit versus explicit structures for organizing, storing and sharing open software requirements. "Low" measures of such virtues might potentially point to increased likelihood of a failure to develop a sustainable open software system. Subsequently, improving the quality of such virtues for open software requirements may benefit from tools that encourage community development; social interaction and communicative expression; software reading and comprehension; community hypertext portals and Web-based repositories. Nonetheless, resolving such issues is an appropriate subject for further study.

Overall, open software development practices are giving rise to a new view of how complex software systems can be constructed, deployed, and evolved. Software informalisms and their corresponding software applications/tools are not yet part of, nor integrated with, the traditional requirements engineer's toolset. Open software development does not adhere to the traditional engineering rationality or virtues found in the legacy of software engineering life cycle models or prescriptive standards.

The development open software system requirements is inherently and undeniably a complex web of socio-technical processes, development situations, and dynamically emerging development contexts [2, 15, 20, 37, 38]. In this way, the requirements for open software systems continually emerge through a web of community narratives. These extended narratives embody discourse that is manifest through an open software requirements engineering process. Participants in this process capture in persistent, globally accessible, open software informalisms that serve as their organizational memory [1], hypertextual issue-based information system [5, 23], and a networked community environment for information sharing, communication, and social interaction [18, 30, 36, 37]. Consequently, ethnographic methods are needed to elicit, analyze, validate, and communicate what these narratives are, what form they take, what practices and processes give them their form, and what research methods and principles are employed to examine them [15, 16, 19, 20, 28, 38]. Employing these methods reveals what is involved in this open software process, and how developer-users participate to create their discourse using software requirement informalisms in the course of developing the open software systems they seek to use and sustain. This report thus contributes a new study of this kind.

Acknowledgements

The research described in this report is supported by a grant from the National Science Foundation #IIS-0083075, and from the Defense Acquisition University by contract N487650-27803. No endorsement implied. Mark Ackerman at the University of Michigan, Mark Bergman, Xiaobin Li, and Margaret Elliott, at the UCI Institute for Software Research, and also Julia Watson at The Ohio State University are collaborators on the research project described in this paper.

8. References

1. ACKERMAN, M.S. and HALVERSON, C.A.: 'Reexamining Organizational Memory', *Communications ACM*, **43**, (1), pp. 59-64, January 2000.
2. ATKINSON, C.J.: 'Socio-Technical and Soft Approaches to Information Requirements Elicitation in the Post-Methodology Era', *Requirements Engineering*, **5**, pp. 67-73, 2000.
3. BOEHM, B., EGYED, A., KWAN, J. PORT, D., SHAH, A., AND MADACHY, R.: 'Using the WinWin Spiral Model: A Case Study', *Computer*, **31**, (7), pp. 33-44, 1998.

4. BOWKER, G.C. and STAR, S.L.: '*Sorting Things Out: Classification and Its Consequences'*, MIT Press, Cambridge, MA, 1999.
5. CONKLIN, J. and BEGEMAN, M.L.: 'gIBIS: A Hypertext Tool for Effective Policy Discussion', *ACM Transactions Office Information Systems*, **6**, (4), pp. 303-331, October 1988.
6. COOK, J.E., VOTTA, L.G., and WOLF, A.L.: 'Cost-effective analysis of in-place software processes', *IEEE Transactions Software Engineering*, **24**, (8), pp. 650-663, 1998.
7. CLEVELAND, C.: 'The Past, Present, and Future of PC Mod Development', *Game Developer*, pp. 46-49, February 2001.
8. CURTIS, B., KELLNER, M.I. and OVER, J.: 'Process modeling', *Communications ACM*, **35**, (9), pp. 75- 90, 1992.
9. CYBULSKI, J.L. and REED, K.: 'Computer-Assisted Analysis and Refinement of Informal Software Requirements Documents', *Proceedings Asia-Pacific Software Engineering Conference* (APSEC'98), Taipei, Taiwan, R.O.C., pp. 128-135, December 1998.
10. DAVIS, A.M.: '*Software Requirements: Analysis and Specification'*, Prentice-Hall, 1990.
11. DIBONA, C. OCKMAN, S. and STONE, M.: '*Open Sources: Voices from the Open Source Revolution'*, O'Reilly Press, Sebastopol, CA, 1999.
12. FIELDING,R.T.: 'Shared Leadership in the Apache Project', *Communications ACM,* **42**, (4), pp. 42-43, April 1999.
13. FLAKE, G.W., LAWRENCE, S., and GILES, C.L.: 'Efficient Identification of Web Communities', *Proc. Sixth Intern. Conf. Knowledge Discovery and Data Mining*, (ACM SIGKDD-2000), Boston, MA, pp. 150-160, August 2000.
14. FOGEL, K.: '*Open Source Development with CVS*'. Coriolis Press, 1999.
15. GOGUEN, J.A.: 'Formality and Informality in Requirements Engineering (Keynote Address)', *Proc. 4th. Intern. Conf. Requirements Engineering,* pp. 102-108, IEEE Computer Society, 1996.
16. HINE, C.: '*Virtual Ethnography'*, SAGE Publishers, London, 2000.
17. JACKSON, M.: '*Software Requirements & Specifications: Practice, Principles, and Prejudices'*, Addison-Wesley Pub. Co., Boston, MA, 1995.
18. KIM, A.J.: '*Community-Building on the Web: Secret Strategies for Successful Online Communities'*, Peachpit Press, 2000.
19. KLEIN, H. AND MYERS, M.D.: 'A Set of Principles for Conducting and Evaluating Intrepretive Field Studies in Information Systems', *MIS Quarterly*, **23**, (1), pp. 67-94, March 1999.
20. KLING, R. and SCACCHI, W.: 'The Web of Computing: Computer technology as social organization'. In M. Yovits (ed.), *Advances in Computers*, **21**, pp. 3-90. Academic Press, New York, 1982.
21. KOCH, S. and SCHNEIDER, G.: 'Results from software engineering research into open source development projects using public data', *Diskussionspapiere zum Taetigkeitsfeld Informationsverarbeitung und Informationswirtschaft*, H.R. Hansen und W.H. Janko (Hrsg.), Nr. 22, Wirtschaftsuniversitaet Wien, 2000.
22. KOTONYA, G. and SOMMERVILLE, I.: '*Requirements Engineering: Processes and Techniques',* John Wiley and Sons, Inc, New York, 1998.
23. LEE, J.: 'SIBYL: a tool for managing group design rationale', *Proceedings of the Conference on Computer-Supported Cooperative Work*, Los Angeles, CA, ACM Press, pp. 79-92, 1990.
24. MI, P. and SCACCHI, W.: ' * A Meta-Model for Formulating Knowledge-Based Models of Software Development', *Decision Support Systems*, **17**, (4), pp. 313-330, 1996.
25. MOCKUS, A., FIELDING, R.T., and HERBSLEB, J.: 'A Case Study of Open Software Development: The Apache Server'*, Proc. 22nd. International Conference on Software Engineering*, Limerick, IR, pp. 263-272, 2000.
26. NOLL, J. and SCACCHI, W.: 'Supporting Software Development in Virtual Enterprises'. *J. Digital Information*, **1**, (4), February 1999, http://jodi.ecs.soton.ac.uk/Articles/v01/i04/Noll/
27. NOLL, J. and SCACCHI. W.: 'Specifying Process-Oriented Hypertext for Organizational Computing', *J. Network and Computer Applications*, **24**, (1), pp. 39-61, 2001.
28. NUSEIBEH, R. and EASTERBROOK, S.: 'Requirements Engineering: A Roadmap', in A. Finkelstein (ed.), *The Future of Software Engineering*, ACM and IEEE Computer Society Press, http://www.softwaresystems.org/future.html, 2000.
29. PAVLICEK, R.: '*Embracing Insanity: Open Source Software Development'*, SAMS Publishing, Indianapolis, IN, 2000.

30. PREECE, J.: '*Online Communities: Designing Usability, Supporting Sociability'*. Chichester, UK: John Wiley & Sons, 2000.
31. PLUMMER, D.A. and SUBRAMANIAN, S.: 'The Chandra Automatic Data Processing Infrastructure', in *ASP Conference. Series, **238**, Astronomical Data Analysis Software and Systems X*, in F. R. Harnden, Jr., F. A. Primini, & H. E. Payne (eds.), San Francisco: ASP, Paper #475, 2000.
32. RAYMOND, E.: '*The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary'*, O'Reilly and Associates, Sebastopol, CA, 2001.
33. ROBBINS, J. E. and REDMILES, D. F.: 'Cognitive support, UML adherence, and XMI interchange in Argo/UML', *Information and Software Technology*, **42**, (2), pp. 71-149, 25 January 2000.
34. SCACCHI, W.: 'Understanding Software Process Redesign using Modeling, Analysis and Simulation , *Software Process--Improvement and Practice*, **5**, (2/3), pp. 183-195, 2000.
35. SHORTRIDGE, K.: 'Astronomical Software--A Review', *ASP Conference. Series., **238**, Astronomical Data Analysis Software and Systems X,* in F. R. Harnden, Jr., F. A. Primini, & H. E. Payne (eds.), San Francisco: ASP, Paper #343, 2000.
36. SMITH, M. and KOLLOCK, P. (eds.): '*Communities in Cyberspace'*, Routledge, London, 1999.
37. TRUEX, D., BASKERVILLE, R. and KLEIN, H.: 'Growing Systems in an Emergent Organization', *Communications ACM*, **42**, (8), pp. 117-123, 1999.
38. VILLER, S. and SOMMERVILLE, I.: 'Ethnographically informed analysis for software engineers', *Int. J. Human-Computer Studies*, **53**, pp. 169-196, 2000.
39. YAMAGUCHI, Y., YOKOZAWA, M., SHINOHARA, T., and ISHIDA, T.: 'Collaboration with Lean Media: How Open-Source Software Succeeds', *Proceedings of the Conference on Computer Supported Cooperative Work,* (CSCW'00), pp. 329-338, Philadelphia, PA, ACM Press, December 2000.
40. ZELOKOWITZ, M.V. and WALLACE, D.: 'Experimental Models for Validating Technology', *Computer*, **31**, (5), pp. 23-31, May 1998.