Open Source Software – Study of Requirement for its Development

Prof. Vinit A. Sinha
Assistant Professor
MCA Department, P.R.M.I.T & R. Badnera – Amravati (M.S)

Abstract: - This Study is result of searching from study of configuration of technical system, organizational context, study of social process and growing relationship for rising in open source software. Main thing include is 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. Open source software now plays vital role in development field of technical areas. Here some development communities are described, examined, and compared to help discover use of OSS in research field. Some kind of software are mentioned here for growth in development of open source in replace of close operating system platform. Subsequently, implementation of this kind of software for better result is the objective of this study.

Keywords: - OSS, SRS, OSS Games

I. 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, and 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.

II. 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 access1, 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 collocated 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.

III. 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.

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.

IV. THE CLASSIC SOFTWARE REQUIREMENTS ENGINEERING PROCESS

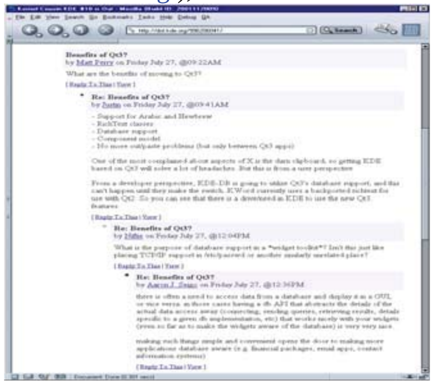
Experts in the field of software requirements engineering identify a recurring set of activities that characterize this engineering process [10, 17, 22, 28]. These activities, which are generally construed as necessary in order to produce a reliable, high quality, trustworthy system, include: *Eliciting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

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 a software requirements specification (SRS) document, establishing criteria for requirements traceability, and managing the storage and evolution of the preceding requirements artifacts.

V. 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 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.

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.



Benefits of Qt3?
by Matt Perry on Friday July 27, @09:22AM

What are the benefits of moving to Qt3?

[Reply To This | View]

- **Re: Benefits of Qt3?**
  by Justin on Friday July 27, @09:41AM

  - Support for Arabic and Hewbrew
  - RichText classes
  - Database support
  - Component model
  - No more cut/paste problems (but only between Qt3 apps)

  One of the most complained about aspects of X is the dam clipboard, so getting KDE based on Qt3 will solve a lot of headaches. But that is from a user perspective.

  From a developer perspective, KDE-DB is going to utilize Qt3's database support, and that can't happen until they make the switch. KWord currently uses a backported richtext for use with Qt2. So you can see that there is a drive/need in KDE to use the new Qt3 features.

  [Reply To This | View]

  - **Re: Benefits of Qt3?**
    by Hiflo on Friday July 27, @12:04PM

    What is the purpose of database support in a "widget toolkit"? Isn't that just like placing TCP/IP support in /etc/passwd or another similarly unrelated place?

    [Reply To This | View]

    - **Re: Benefits of Qt3?**
      by Aaron J. Seigo on Friday July 27, @12:36PM

      there is often a need to access data from a database and display it in a GUI, or vice versa. in those cases having a db API that abstracts the details of the actual data access away (connecting, sending queries, retrieving results, details specific to a given db implementation, etc) that works nicely with your widgets (even so far as to make the widgets aware of the database) is very very nice.

      making such things simple and convenient opens the door to making more applications database aware (e.g. financial packages, email apps, contact information systems)

      [Reply To This | 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.

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].

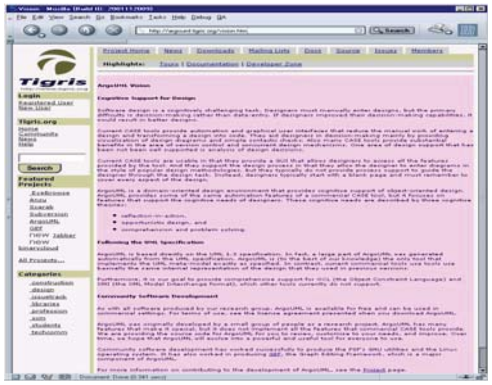


Exhibit 2. A software requirements vision statement highlighting community development as a software Development objective (i.e., a non-functional requirement). Source: http://argouml.tigris.org/vision.html, July 2001.\

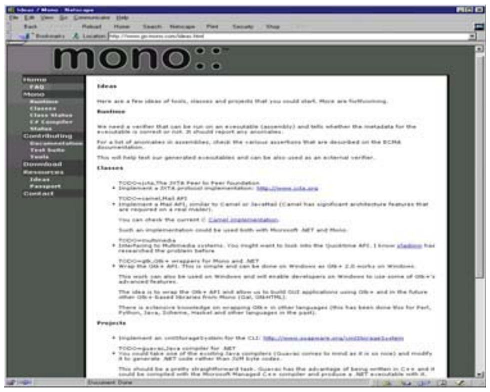
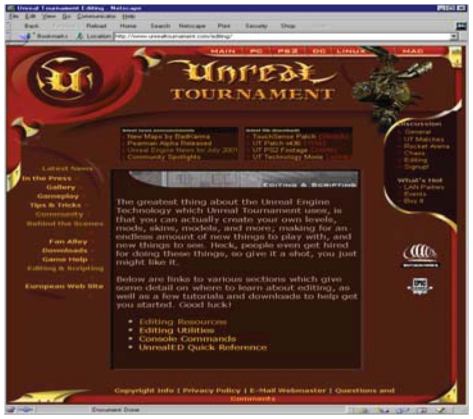


Exhibit 3: A non-functional requirement identifying a need for volunteers to become owners for software Components (or classes) not yet bound to a developer. Source: http://www.go-mono.com/ideas.html, July 2001.



The greatest thing about the Unreal Engine Technology which Unreal Tournament uses, is that you can actually create your own levels, mods, skins, models, and more; making for an endless amount of new things to play with, and new things to see. Heck, people even get hired for doing these things, so give it a shot, you just might like it.

Below are links to various sections which give some detail on where to learn about editing, as well as a few tutorials and downloads to help get you started. Good luck!

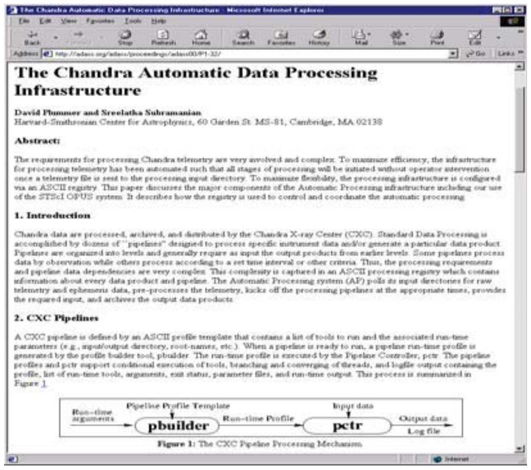
- Editing Resources
- Editing Utilities
- Console Commands
- UnrealED Quick Reference

Copyright Info | Privacy Policy | E-Mail Webmaster | Questions and Comments

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

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.


The Chandra Automatic Data Processing Infrastructure - Microsoft Internet Explorer
Pipeline Profile Template
Run-time arguments
pbuilder
Run-time Profile
Input data
pctr
Output data
Log file

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

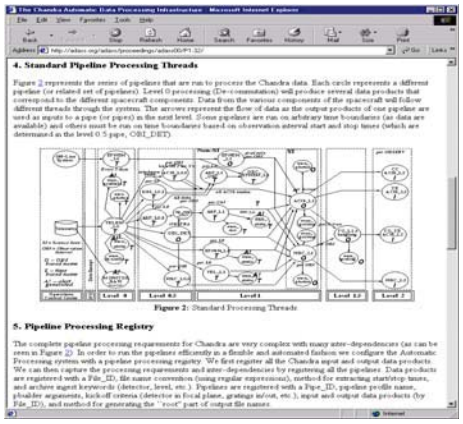
Chandra data are processed, archived, and distributed by the Chandra X-ray Center (CXC). Standard Data Processing is accomplished by dozens of "pipelines" designed to process specific instrument data and/or generate a particular data product. Pipelines are organized into levels and generally require 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.

Figure 1: The CXC Pipeline Processing Mechanism

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

http://adass.org/adass/proceedings/adass00/P1-32/

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.

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]

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.

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.

5.1 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.

5.2 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.

VI. 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 informalisms10 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.

VII. UNDERSTANDING THE REQUIREMENTS OF OPEN SOFTWARE

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]. 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.

VIII. CONCLUSIONS

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).

The development open software system requirements is inherently and undeniably a complex web of sociotechnical 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.

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.