Toward Visualization and Analysis of Traceability Relationships in Distributed and Offshore Software Development Projects

Cleidson R. B. de Souza[1], Tobias Hildenbrand[2], David Redmiles[3]

[1] Departamento de Informática, Universidade Federal do Pará
Campus Universitário do Guamá, Belém, PA, 66.075-110, Brazil
cdesouza@ufpa.br

[2] Lehrstuhl für ABWL und Wirtschaftsinformatik, Universität Mannheim,
Schloss, D-68131 Mannheim, Germany
hildenbrand@uni-mannheim.de

[3] Department of Informatics, Donald Bren School of Information and Computer Sciences,
University of California, Irvine
Irvine, CA 92697-3430, USA
redmiles@ics.uci.edu

Abstract. Offshore software development projects provoke new issues to the collaborative endeavor of software development due to their global distribution and involvement of various people, processes, and tools. These problems relate to the geographical distance and the associated time-zone differences; cultural, organizational, and process issues; as well as language problems. However, existing tool support is neither adequate nor grounded in empirical observations. This paper presents two empirical studies of global software development teams and their usage of tools. The results are then used to motivate and inform the construction of more useful software development tools for offshore projects. This research focuses on issues that are tool-related but have not yet been solved by existing tools. The two software tools presented as solutions, Ariadne and TraVis, explicitly address yet unresolved issues in global software development and also integrate with prevalent other solutions.

Keywords: Traceability Relationships, Dependencies, Visualization, Distributed and Global Software Development, Offshore Software Development.

1 Introduction

Software development is one of the most common examples of collaborative work. Software developers interact face to face and through different artifacts to reach their common goal. To support this effort, several approaches and tools have been proposed. For instance, software processes [1] establish the sequence of activities that software developers need to perform, and modularization techniques [2] allow a

software system to be decomposed and developed in pieces (artifacts) that can be later reintegrated. Tool support has also been adapted, ranging from simple tools such as instant messenger systems and email, to configuration management tools [3, 4], and to even more specialized collaborative tools such as Ariadne [5, 6], Augur [7], Jazz [8], and Palantír [9], among others.

More recently, due largely to economic considerations [10], engineers from different countries and continents have collaborated in *global* software development (GSD) efforts—software projects that span countries, time zones, and even continents [10]. Therefore, projects involving organizations from other continents are called "offshore" projects. However, these projects bring additional problems to the collaborative endeavor of software development: the geographical distance and the associated time-zone differences; cultural, organizational, and process issues; as well as language problems [11, 12]. Despite the commonalities in such global efforts, adequate tool support for these activities has not yet been fully realized. Furthermore, the few existing tools that have been proposed for offshore scenarios are rarely grounded in empirical data. The contribution of this paper is exactly that—it presents two empirical studies about GSD teams in offshore scenarios and their utilization of tools. The results are then used to motivate the construction of more adequate and useful software tools in this context. The empirical studies identify persistent problems in organizations conducting GSD projects. The focus is on issues that are tool-related but have not yet been solved by existing solutions. These particular issues are used to motivate the construction of two novel software tools, Ariadne and TraVis, which explicitly address yet unresolved issues in global software development and complement existing tools in practice.

This paper is organized as follows. The next section presents the empirical studies and section 3 then describes the tools that were developed grounded on the analysis of the empirical data. A discussion section about the tools specific usefulness and value added follows. Finally, conclusions and suggestions for future work are presented.

2 Empirical Studies

This section presents the two empirical studies that were conducted. The first one is a qualitative study that aimed to explore the advantages and disadvantages of offshore software development (OSD). The second one is an analysis of feature and change requests for a collaboration platform aimed at collaborative and distributed software development teams.

2.1 The MBL Offshore Software Development Project

This empirical study describes a qualitative investigation of an offshore software development project that took place in a large software development organization. A qualitative approach was adopted because it allowed us to investigate our own research questions while it also offered the flexibility to explore issues from the perspectives of the informants [13].

2.1.1 Setting and Methods

Our fieldwork was conducted in a large software development company we will call LAR (a pseudonym). LAR is one of the largest software development companies in the United States, with products ranging from operating systems to software development tools, including e-business and tailored applications. The project we studied, MBL (another pseudonym), was responsible for developing a mobile application that had not yet been released during the period of the study. The project staff was divided into three major groups: user interface (UI) designers, software developers, and the quality assurance (QA) team. The staff was distributed over five different sites spread in three different countries: North Carolina, US; Massachusetts, US; Beijing, China; Shanghai, China; and Taipei, Taiwan. To be more specific, user interface design and evaluation was performed by six professionals in North Carolina, and the implementation was performed in all other sites distributed as follows: nine developers in Massachusetts, five in Shanghai, five in Beijing, and four in Taipei. The quality assurance team was divided between the US and Chinese sites: three engineers were located in Massachusetts and six engineers in Beijing. The main coordination of the project and the project manager for this project were located in Massachusetts, where all the data were collected.

Data were collected through document analysis and semi-structured interviews [14]. Among other documentation, we collected artifacts, emails and instant messages exchanged among the software engineers. We were also granted access to shared discussion databases used by the software engineers. All of this information was used in addition to notes generated by the interviews. We conducted 17 semi-structured interviews with members of all teams from the different sites: some interviews were conducted face to face, and others were conducted by telephone, with one interview conducted by using instant messaging. The interview questions were designed to encourage the participants to talk about their everyday work, including work processes, problems, tools, communication, collaboration, and coordination efforts between their collocated and distributed colleagues. Interviews also aimed to explore the relationship between software dependencies and the coordination of software development projects, or, to be more specific, the potential usage of dependency information to facilitate collaborative software development. Interviews lasted between 20 and 70 minutes. All the material collected has been analyzed using grounded theory techniques [13]. The following sections describe the results of this analysis.

2.1.2 MBL's Software Development Process

As expected, user interface designers wrote specifications in the very beginning of the process. These specifications were aimed at managers and contained high-level descriptions of the functionalities of the application. Detailed specifications, also called design documents, were written by software developers and contained parts of source code (method calls, Java interfaces, etc.). The goal of these specifications was to allow the reader to learn how to use the software component being implemented; that is, they provided as much detail as possible about how the specifications would be implemented without actually coding them. UI designers and software developers reviewed the initial versions of these documents, which were stored in shared folders accessible to all team members.

Specifications and design documents were used to develop test plans, which would be shown to the developers once they finished their implementation work. In fact, testing had already started: the QA engineers tested the software, and when they found an issue, they filled a request in the bug database that would automatically generate an email to the developers. It was the developers' responsibility to find out where in the architecture the bug was to be found because the testing was performed from the point of view of the user. At the time, most issues were simply ignored by developers because they were aware that the issues reflected either things that were not implemented yet or things that they already knew were not working. Based on the specifications, software developers created methods in the source code but without actually providing an implementation for them. When the data collection was conducted, UI designers had just started opening issues in the database for problems in the software. Previously, they informed the developers about issues by using email or instant messages.

2.1.3 Changes in the Specifications and Notifications

UI designers performed usability tests in the specifications even before the implementation was finished. The results of these tests could lead designers to propose changes to the specifications they had created, and as a consequence could impact software developers' work. Indeed, according to one of members of the test team, the specifications changed fairly often. Another software developer reported overhearing his colleagues mentioning they had finished implementing a particular feature and then UI designers requested a change: "Well, I just implemented it this way and now they want us to change it" (informant 16). Other informants also reported not being notified about changes in the specifications. Another reason that led to changes in the specifications involved requests from the software developers. They requested changes to the UI specifications, arguing that some of the UI designs were not technically feasible in the amount of time they had.

Changes in the specifications also impacted the quality assurance team. Members of this team had to write test plans and rewrite them every time the specifications were updated. According to a tester: "it [the process of re-writing test plans] is a very boring job" (informant 09). To minimize this problem, QA engineers broke the test plans into two parts—the test design and the test data—so that only one of them had to be changed when the specification changed. The QA manager in China reported that QA members could not wait for the specifications to be finished to write their test plans because, if they did, they would not have enough time to write the test cases.

In any of these cases, changes in the specifications were preceded by discussions between the UI designers and the software developers through conference calls, emails, instant messaging, or even meetings (informants 14 and 16).

2.1.4 Notification of the Changes

A relevant aspect that came up in the data regarded change notifications, that is, notifications that were sent because of changes in the different artifacts. A software engineer in China reported that he would not receive notifications from his "contact person" in the US. In contrast, the UI designer interviewed reported that she would notify developers and testers of changes in the specifications. Whenever notifications

were received, however, they overwhelmed developers because they were not tailored to them, as a developer in the US properly put it (informant 03):

> I will never go into one document trying to figure out what other people did. ... But I also wouldn't like it—the way that—I get a … ten million different email messages just because somebody did over something that had nothing to do with me.

This quote also illustrates the importance to developers of change management systems that track the changes in artifacts.

2.1.5 Dependency Analysis

As mentioned in the previous section, one of the goals of the interviews was to investigate the usage of software dependency information to coordinate distributed projects. One informant mentioned that by inspecting configuration files of the project, he could find out which *components* depended on the component he was developing. With that information, this developer could find out which *developers* depended on his code, another piece of information that could be used to guide the notifications to be sent whenever artifacts changed.[1]

Another software developer reported a similar interest in finding out who the developers were that were calling her code: "There is no really … good way to keep track of who's consuming your code." The distributed nature of this project was particularly relevant in this case, as the conversation below attests:

> Informant 03: … local people like Mike [pseudonym]. It's easy just to say, "Well, have you tried this? Have you tried this as well?" … "I didn't have time. I'm going to try it tomorrow." And then the second day I bumped into him again. I would bug him and say, "Well, have you tried it? How did it go?" [He would] say, "Oh, that didn't work. I would send you the exception." It's much easier at this point, but let's say the … people in Taiwan … I don't actively go ahead and chase them around and say, "How many—this interface, how many methods have you exercised?" You don't need this. I don't—I mean I don't have time, simply I just don't have time to do that. So … is so much harder.
>
> Researcher: Would you like to know that information?
>
> Informant 03: I definitely would have liked to—would love to—have that information as early as possible … because that makes my life much easier. For example, Mike [a local colleague] starts to use my stuff so late and we have … builds supposedly by 23rd. That's when we have all of the … work. And he only starts to test my stuff on the 21st and by [the] 22nd he realized there are two methods [that are] not really doing what I want it to do. And he told me on the 22nd and I only have one day—actually, not even one day … to do it because we do build like once a day. Well, I would have had like any other build … have to

[1] We will illustrate in the following section that this is exactly the principle used in Ariadne.

> get up at four. … So, of course, I would love to hear them telling me, "Okay, I have ... I did all of testing, [and it] worked."

The quote above illustrates how it is important for developers to know *who* is consuming their codes and *when* this integration with their codes starts. This information is useful because it allows the developer to anticipate the work that will be requested of them before the deadline. This information is necessary from both collocated and distributed colleagues. The informal conversations that are afforded by the collocation simplify this process among local colleagues; in contrast, however, developers are not able to find this status information when their colleagues are distributed over different countries. This is again made clear by informant 03:

> With Mike [a collocated developer], I can say, "Can you please try …?" With Taiwan, I don't do that. I don't really know them that well. I talk to them, but unless something … unless something [is] really important, you don't …

In fact, this informant reported that in one occasion a developer in China was already using her code and she did not know it.

Developers in China reported the exact same problem during the interviews: the Shangai team was developing a software component already in use by the Beijing team for two months and by the US team for only a week. According to the Shangai team leader, his distributed colleagues were giving low priority to his component, and that could lead to problems in the end of the implementation phase for his team because they would have only a short time to fix potential problems. This happened with both the Beijing and the US teams. In fact, he reported having to email his colleagues in Beijing asking them to integrate his component into their code and requesting a deadline for that. During weekly "checkpoints," he would confirm how the integration was going. This team leader trained the Beijing developers on the usage of his component. During the period of data collection, a Beijing developer was in the United States, training the developers on the same component. To accompany this process, the Shangai team leader and the Beijing developer who was in the United States had weekly "checkpoints." It was through one of these checkpoints that the Shangai team found out that the US team had started using his component.

These results are not surprising (the effect of distance on the coordination of the work has been known for decades); however, the possibility of using software dependency information to minimize the coordination problems is an important result of this study. This will be discussed in more detail in the following section.

2.2 Feature and Change Requests for Offshore Collaboration Platforms

Collaborative software development platforms (CSDPs) comprise and unify not only source code management, but multiple software development and knowledge management tools. CSDPs include build management systems, issue trackers, Wikis, and discussion forums [25]. These tools have often been successfully used in distributed open source software development projects as well as offshore software development scenarios [24, 25],. Using a CSDP is state-of-the-art in OSD [26] and

also serves as a method for capturing and maintaining more relevant traceability and rationale information [23, 27]. Therefore, in a second empirical effort, customer feature and change requests of one of the market-leading collaborative software development platform vendors, VCI (pseudonym), have been investigated and analyzed for evidence about yet open tool–related issues in distributed and offshore software development.

2.2.1 Settings and Methodology

VCI has a broad customer base, ranging from large producing companies and financial service providers to software development companies. In fact, most companies use VCI's tool for globally distributed development projects. VCI's collaboration platform supports most common collaboration features such as a document management system (DMS), a Wiki system, issue trackers, reporting, Wiki-enabled discussion forums, and chat rooms, as well as code-related features such as source code and build management (cp. [25]). The platform is not a stand-alone solution for software development, but also integrates various established tools (e.g., the Eclipse development environment, CVS, and Subversion), as well as numerous other tools. The platform does not prescribe one particular development process, which can be an advantage with several heterogeneous sites involved.

Customers are allowed and encouraged to post their requests concerning the platform directly to VCI's change request tracker. This tracker is based on VCI's own tool and establishes something comparable to a community of practice among its customers. The tracker also supports an item-related discussion among users and support personnel. Moreover, users can link change requests to other requests and artifacts and refer to those.

The data inspected was extracted from the change request tracker and contained 183 items from 34 different customers. Each item has a unique identification (ID) number, a brief summary in addition to a more detailed description with comments from different users, as well as responses from VCI's support team, a time stamp, a resolution status, and an issue category.[2] Table 1 shows the distribution of these feature requests over different categories and users. These figures illustrate that besides many unspecified items, the design of the Internet-based (Web) user interface and the lately added Wiki system drew most of the attention from customers. However, many unspecified issues also revolved around UI and Wiki.

In a second analytic step, 232 already resolved items were also investigated because some of those were not yet incorporated in the latest release of the software or were only partly implemented so far. Moreover, some customer needs could not be implemented due to time restrictions, but were valid expressions of their requirements as well. The distribution of categories is comparable to one of the unresolved issues (cp. Table 1) and thus complements the initial data set.

As for the interpretative analysis, each tracker item was investigated, including its cross references and the customers involved[3]. This in-depth analysis revealed that

[2] Please note that the categories have been set by the vendor and several other attributes were not considered for this particular analysis.

[3] In the following sections, different customers, as representatives of their companies, will be coded with capital letters starting with "customer A."

there are currently three major fields of interest among the CSDP users: *artifact change propagation*, *artifact linking and traceability*, and *relationship visualization*.

Table 1 Distribution of Feature Requests over Categories and Users

Category	#Requests	#Users
Client	1	1
Communication	4	4
Database	15	4
Documentation	5	4
Eclipse Plug-in	3	1
Release	2	2
Remote Interface	12	2
Run Time	2	2
Server	11	7
Web User Interface	29	12
Wiki	22	5
Unspecified	77	23
TOTAL	**183**	**34** (unique)

2.2.2 Artifact Change Propagation

Many customers referred to the default notification mechanism for artifact changes as being too exhaustive or not fine-grained enough (e.g., customers C and D, among others). As one customer put it, "there [sometimes] is a mail flood" produced by the platform. In offshore software development (OSD) projects, however, an automated subscription and notification mechanism is critical for global change propagation and management [15]. VCI's collaboration platform already provides capabilities to adapt the propagation and notification patterns, but customers still bring up very special requirements, such as the "ability to subscribe individuals to be notified for a particular task" (customer E) and being able to configure the notification content in order to "quickly decide whether it could be important or not" (customer D).

Regarding distributed change management, there is also a demand for an automatically generated "change history" for certain artifacts or aggregated sets of artifacts. Examples include "a compilation of all changes in a whole subtree [of artifacts]" (customer A) and a more complete presentation of related notifications as a whole because they do not want to have "two separate systems, [the CSDP] and the email system" (customer B). This leads to issues of traceability of development processes and rationale, which are discussed in the next section.

2.2.3 Artifact Linking and Traceability

Capturing and managing traceable information about artifacts, processes, and development rationale seems to be a major issue among VCI's customers. Many issues were related to this problem, half of which are still unresolved in the current version of the platform.

Customers like the idea of being able to link related artifacts by using either Wiki links or the standards association mechanism incorporated in the CSDP. They use Wiki pages and Wiki comments attached to various other artifact types to create a

project-specific traceability network (as stated by customers A and F). However, various customers mention sophisticated ways of linking artifacts not yet supported by the platform. Customer A asked for a unified way of linking different types of artifacts and even automatic synchronization between Wiki links and associations: "The Wiki-description of documents and tracker items should be scanned for inter-Wiki links to tracker items or documents, and those should figure in the [association] tab automatically."

This suggests that an easier and more concise way of capturing and managing links is required. Other customers also required easier and semi-automated linking of external artifacts, such as Internet, intranet, and Wikipedia resources. This requirement has been explicitly stated by customer A and confirmed by the VCI management in another tracker issue: "We need to be able to configure our own [CSDP] extensions."

The concept of rationale management is deeply related to that of artifact linking. Customers use the platform to capture their rationale for decision processes in different software development disciplines, such as requirements engineering, architectural design, and implementation. For instance, customers postulate their need to export the full contents of a tracker into their process reports. Customer G, for instance, expressed a need for a "way to export the full contents of a tracker including the comments."

Even though many requests revolved around Wiki-related issues, this seems to have spurred the general thoughts on different means of linking artifacts and retrieving this information from the platform. When analyzing the dates of committed requests, the discussion about traceability networks established by Wiki webs has apparently ignited more general requests pertaining to artifact linking. For example, customer A requested automatic synchronization of Wiki links with other association mechanisms: "[links] need to be [automatically] added to the [association] tab and removed."

Instead of just providing listed links to certain artifacts or resources, as customer A did, several parties demanded alternative ways of representing the network of links. VCI, as represented by their customer support, responded to this in the following way, according to VCI management: "In the first step we provide the relationship visualization with a quite simple GUI. In the next releases we will add graph, hypergraph, [and] MindMap visualization." VCI regards these features as complementary to the existing reporting functionality of their CSDP.

2.3 Brief Summary of Empirical Findings

As has been shown in the empirical studies in section 2, tool-related issues in offshore software development revolved around change management and traceability issues, in addition to the more general issues of distributed collaboration and asynchronous communication, which have been improved by the use of a CSDP in the VCI study. Both studies showed that change propagation and notification are still major issues in different tool settings. The MBL study also revealed many issues pertaining to dependency analysis and management, whereas the VCI customers had problems with linking different artifacts and visualizing these relationships.

3 Tool Support for Offshore Software Development

This section describes two different tools created to support OSD. The first, Ariadne, is an Eclipse plug-in, whereas the second tool, TraVis, is built on top of the same collaborative software development platform VCI used in our second empirical study. More important, the designs of both tools were informed by the empirical data described in the previous section. Both tools focus on the identification, analysis, and visualization of dependency and traceability information that exist among the software development artifacts. With this information, it is possible to identify software developers associated with these artifacts and provide tailored notification of changes and proper impact analysis. Furthermore, it is also possible to perform social network analysis to identify developers who play special roles in the software development process.

3.1 Ariadne

3.2.1 Functionality and Features

Ariadne is designed to perform automatic dependency analysis on software projects shared in configuration management repositories, and to generate visualizations of social dependency information. Generating social dependencies involves three types of dependency information. Initially, Ariadne identifies the technical dependencies in the source code by constructing call-graphs. According to Callahan and colleagues, a call-graph "summarizes the dynamic invocation relationships between procedures" [16]. Second, by describing dependencies in the source code, a call-graph potentially unveils dependencies among software developers responsible for the software components [17, 18]. In order to reveal dependencies among developers, it is necessary to populate the call-graph with "social information." The ultimate goal is to create a data structure that describes which software developers depend on which other software developers for a given piece of code. An example of this data-structure, called a social call-graph [17]. Last, because social call-graphs describe both technical dependencies and authorship information, they can be used to generate sociograms describing the dependence relationship only among software developers. That is, they can show social dependencies among developers that exist because of dependencies in the source code on which they are working. A sociogram, as used in social network analysis [19], is a graphical representation of a set of items, vertices, or nodes connected to one another via links or edges. The sociogram of the Tyrant project is shown on Figure 1.

The sociograms generated by Ariadne can be used by software developers to identify two important pieces of information: who they depend on and who depends on their work. As the MBL data indicate, this information is very important to facilitate the coordination of distributed software development projects. We have also used these sociograms to understand free/open source software development [20].

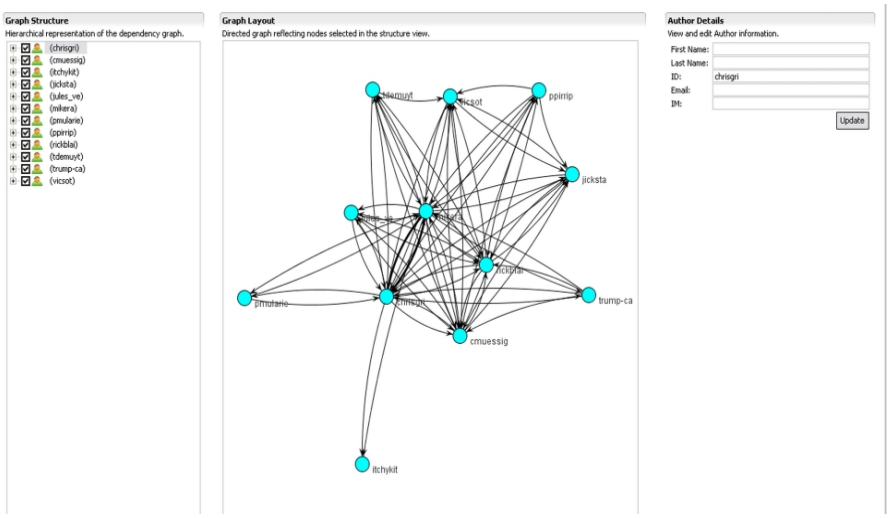

Figure 1 Tyrant's Sociogram

3.1.2 Architecture

Ariadne is implemented as a Java plug-in to the popular Eclipse IDE. As such, Ariadne is integrated into this environment and makes use of Eclipse functionality and its plug-in model. The dependency processing functionality is encapsulated in a main control plug-in that delegates source-code analysis, annotation of the source-code analysis data, and visualization of the created data structure to sub-plug-ins. As a result, Ariadne offers users the flexibility to use dependency generators for a diverse set of source languages, configuration management repositories, and methods of visualization.

Ariadne automatically selects (while offering users the ability to override this choice) appropriate plug-ins for analyzing the user's project based on the project's context. Once the control plug-in has located appropriate sub-plug-ins to analyze the project's source code and query the project's configuration management, the control plug-in automatically generates social dependencies for that project. By using one of the installed visualization plug-ins, it is possible to display all three types of dependency information to the user: technical dependencies, social call-graph, and sociograms.

Our current implementation can present call-graphs and social call-graphs at three different levels of abstraction, based on the programming language's hierarchy (e.g., packages, classes, methods in Java). Essentially, information is aggregated at each hierarchy level to, potentially, average the different results provided by diverse call-graph extractors [21]. For instance, class dependencies are displayed as the aggregation of method dependencies (i.e., the call-graph).

Ariadne was initially implemented to analyze only Java projects and extract information from CVS repositories. Later, we redesigned it based on a layered architecture to be general enough to support various programming languages, configuration management (CM) systems, and visualizations. The configuration

management and dependency management parts of the API are used to isolate the programming language and configuration management tools from the visualizations provided by Ariadne. Through this approach, independent developers can contribute new plug-ins (configuration management tools and programming languages) to Ariadne while reusing previous visualizations. It is also possible to easily design new visualizations to already supported programming languages and CM tools. Ariadne has default graph and tree view visualizations built in. Note that although Eclipse has a generic Team API for accomplishing simple tasks involving version-controlled files, programmers must use the internal (unpublished) API to accomplish more complicated tasks. The inability to directly manipulate remote resources motivated us to create our own remote resource API.

To facilitate the understanding and usage of Ariadne's API, we utilize the façade design pattern [22] to aggregate methods to be used to query program dependency, authorship information and both types of information combined (the social call-graph). For example, developers may query the classes that depend on a particular class, the authors of a particular piece of code, all the authors of a file, how the ownership of a class changes from one release to the next, and so on.

3.2 TraVis – Trace Visualization

The TraVis (Trace Visualization) tool leverages the use of dependencies among distributed assets unified in one CSDP and their users by allowing the visualization and analysis of these different relationships. The traceability and rationale information is captured as distributed CSDP users develop and document their processes in OSD projects. The artifacts are then annotated and connected with their respective descriptions, discussions (e.g., design-related), as well as inter-related process steps represented as issue tracker items. TraVis captures both traces explicitly modeled as associations and more implicit links as built by Wiki systems integrated in the CSDP.

This way, they form a heterogeneous network of information. Managing all this information on one single CSDP allows linking all the artifacts, activity descriptions, and responsible users, consequently establishing the actual "traceability network," as described by [23]. This network is complemented by capturing rationale information, that is, making decision processes (e.g., design or code-related changes) traceable by storing the history of artifacts and users' justifications behind decisions (see [27]). TraVis provides advanced visualization and analysis capabilities for traceability networks, including several logical filters for displaying certain aspects (e.g., particular artifact types, process categories, or user groups). Thus, different role-based views, e.g. for source code developers, designers, and project managers, can be defined. Moreover, TraVis is able to display networks originating from particular artifacts, activities, and users (see Figure **Fehler! Verweisquelle konnte nicht gefunden werden.**2).

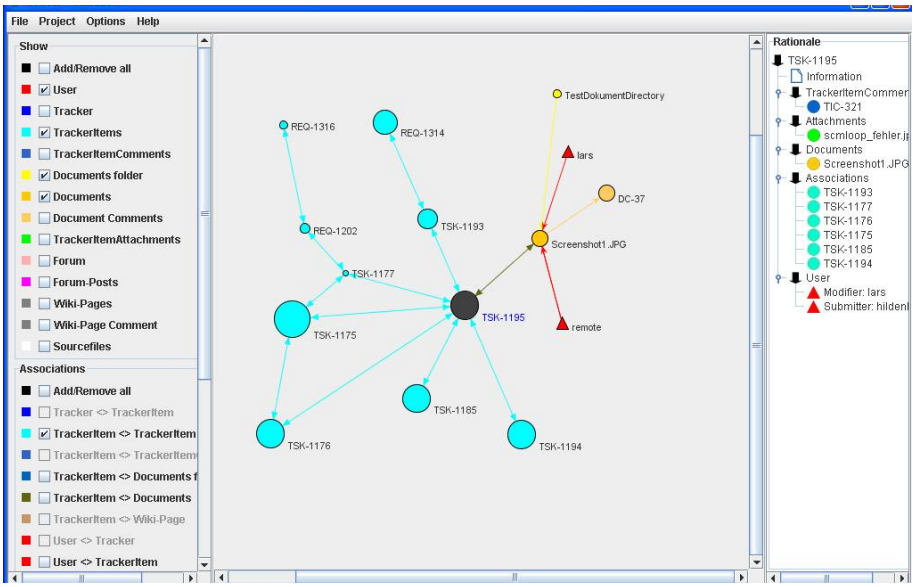

Figure 2 Value-Based Trace Visualization with TraVis

This allows analyzing the graphs that grow around one particular entity (in Figure 2: task 1195), for instance in order to conduct impact analyses centered on certain artifacts that are subject to change. Moreover, the latest version of TraVis displays artifact nodes according to their user value (*value-based software engineering*, VBSE, see [28]). Because OSD projects contain a plethora of linked information, we implemented methods like VBSE to reduce and enrich the network information in order to be more useful to both developers and managers (see [29]). To this aim, not only the node but also the edges contain additional information, such as the type of relationship (i.e. link semantics) or the rationale for linking one artifact with another or replacing it by a changed version. Rationale information (e.g., comments on committed code or changed artifacts) is displayed in one context with the related artifacts (see right column in Figure 3).

Technologically, TraVis extracts traceability information from CSDPs over their remote APIs (e.g., via Web Services) or directly from their Web interfaces. TraVis stores this information internally and so far, no redundant persistent storage is implemented. TraVis then constructs the traceability network structure from the CSDP information. Artifacts, such as documents, code, Wiki pages, and tracker items, as well as related users are graphically represented as nodes, whereas their relationships are stored as edges. For visualization purposes, the open source software "Java Universal Network/Graph Framework" (JUNG)[4] was chosen. It enables many helpful features such as zooming as well as manual and automatic graph reorganization according to pre-defined patterns and algorithms.

[4] http://jung.sourceforge.net/ (09/30/2006)

4 Discussion

The most important aspect that can be identified in both the MBL and the VCI studies is the possibility of using dependency information to facilitate the coordination of OSD activities. Being able to find out when a dependency started to exist between software artifacts is an important aspect raised in the MBL study. Providing that information can facilitate the coordination of software development projects through the awareness of other developers' work status [4, 30]. This aspect is deeply embedded into both Ariadne and TraVis. In fact, Ariadne supports the *automatic* identification of dependencies among software components. By doing that, Ariadne allows software developers to determine the set of developers who are using their codes as well as when this usage started. Ariadne is limited to source code to allow the automatic creation of dependency links because the source code can be properly parsed. Ariadne finds the dependency links automatically, so developers do not have to worry about manual artifact linking and trace capturing. As postulated by several VCI customers, Ariadne's automatic extraction of dependencies addresses the issue of extensive exports of traceability network information—at least for code-related artifacts and their authors.

In contrast, TraVis is based on the *manual* creation of dependency relationships. Its information base is not limited to source code only, however, and it supports all ranges of software development artifacts. TraVis also provides a real-time view of the project's traceability network of artifacts and users, which contains valuable information about dependencies between software developers (similar to Ariadne) and therefore supports better awareness for what other people work on. TraVis thus allows for real-time graphical representations and analyses. TraVis is also able to display the current status of inter-related tasks represented as tracker items. This allows for even better awareness in OSD scenarios. By providing a finer-grained dependency and tracking analysis of artifacts produced by distributed teams, both TraVis and Ariadne permit individual users to understand their roles in the broader software development process with respect to other users and the artifacts that they produce. This understanding is crucial to the coordination of collaborative software development efforts [31].

Once the dependency information is available to our tools, it is possible to tailor the notification messages that are sent due to changes in the artifacts; that is, notifications about the changes can now be sent only to the subset of developers who are interested in the changes. We assume that this subset is basically the set of developers who depend on the artifact being changed, but that assumption is grounded on our empirical data (see sections 2.1.3 and 2.2.2). The goal here is to reduce the number of notifications that software developers receive because a common problem in both the MBL and VCI studies was the overwhelming flood of notification messages initiated by other software developers or software tools due to changes in the artifacts. TraVis's role-based views and filtered visualization can also be used to alleviate this problem. TraVis can also help information brokers in offshore settings, as in the MBL study (section 2.1), to manage their immense workload of emails. By providing information about other users related to networks of artifacts and the possibility of starting an instant messaging session or Voice-over-IP conversation using the collaboration platform interface, TraVis supports self-selection of relevant

users and an easy communication process. This, in turn, should relieve the brokers in their dispatching tasks. Again, the filters and the value-based software engineering perspective provide a better overview of the project to both managers and developers. In some ways comparable to TraVis, Ariadne also facilitates the management of notifications by providing visualizations of technical and social dependencies. The developers are thus able to actually see who is working on code artifacts related to theirs and contact these persons purposefully.

As mentioned before, notification messages are necessary because of changes in the artifacts. Often, these changes are preceded by discussions. However, these discussions were not visible to all interested parties (section 2.1.3) due to the distributed nature of the project. TraVis addresses issues related to unexpected changes in software development artifacts (including specifications) as follows: rationale information from a Wiki-based software development platform allows one to notice whenever there is a discussion going relating to a certain artifact of interest. For example, if the specification is assembled using a Wiki or the specification document is annotated and discussed through the Wiki, TraVis can display this correspondence with the affected artifact. By doing that, the relevant information is accessible to everybody entitled to read it according to the roles defined on the CSDP. In short, software developers who depend on a particular artifact can find out that this artifact is likely to change and therefore impact their own work by accessing the discussion that takes place about the artifact.

Finally, a relevant aspect of both Ariadne and TraVis is the focus on visualizations. Visualizations shift the load from the cognitive system to the perceptual system, capitalizing on the human visual system's ability to recognize patterns and structures in the visual information [32]. In terms of relationship visualization, Ariadne provides different types of network graphs (i.e., call-graphs, social call-graphs, and sociograms (see section 3.1)). As discussed in the previous paragraphs, these representations can be utilized to facilitate team communication in distributed and collocated settings.

TraVis visualizes all possible artifact relations and is thus able to provide traceability and rationale information to different project stakeholders. Role-based filters and the value-based perspective guarantee the appropriateness of the traceability network for diverse users. The integration with Wiki-enabled CSDPs ensures that capturing and managing the network information is easy and concise. TraVis also includes a set of predefined views and filters (see section 3.2.1) in order to enable alternative visualizations of relationships. As can be seen in Figure 2, different network compositions and layouts can be chosen. By means of a Wiki plug-in, it is also possible to include TraVis snapshots in a Wiki page, for example, for better communication. Due to TraVis's component-based architecture, an easy transition to other network representations, such as semantic networks and topic maps, is assured. In short, TraVis provides increased awareness within offshore software development projects based on a broad range of traceability and rationale visualizations that are created with information extracted from the collaborative development platform.

TraVis supports artifact change propagation as well as linking and the visualization of these relationships in multiple respects: First, it supports change histories (see section 2.2.2) by providing rationale information for each artifact. Hence, TraVis users can always check the artifacts' history and related discussions. Like Ariadne,

TraVis creates better overall awareness through visualizations of the artifacts' dependencies and their related users. As has been concluded before, this enables more targeted communication processes complementary to the CSDP's notification engine.

5 Conclusions and Future Work

When comparing the issues arising in both studies, one can notice that they mostly deal with problems of change management and propagation as well as dependency management in traceability networks. These issues are analyzed in distributed and OSD contexts in order to empirically ground our requirements for tools to support these processes.

In addition to commonplace source code management and CSDP solutions, the analysis and visualization tools Ariadne and TraVis are designed and presented. The discussion in section 4 shows that these tools provide advanced functionality as demanded by practitioners in real-world offshore projects. However, in doing so we take a rather tool-centered perspective on OSD issues, deliberately not addressing social and inter-cultural problems in the first place.

Our future work basically includes two streams of research: First, further empirical studies will be conducted with other companies developing software in distributed and offshore scenarios. Second, a deeper integration of the two tools, Ariadne and TraVis, is planned to allow Ariadne to read other than code-related information from CSDPs. Because CSDPs are evolving as state-of-the-art in OSD, however, TraVis will be the basis for further tool development and will be enhanced by *automated* link extraction and social network analysis capabilities from Ariadne.

In addition to continuing our empirically informed design, the tools will be evaluated in various settings, such as open source software development (OSSD) projects. OSSD projects already use CSDPs and are therefore comparable to OSD settings from a tool-based perspective. They contain freely available data than can be applied to show the usefulness of our approaches in large-scale distributed projects. We also intend to have OSSD practitioners evaluate our various visualizations in order to determine the most useful ones. The tools will also be evaluated within globally distributed student projects involving universities in Brazil, Germany, Puerto Rico, and the United States.

Acknowledgments. This work was supported in part by the National Science Foundation under awards 0205724 and 0326105; by IBM through the Eclipse Innovation Program; and by the Brazilian government under CAPES Grant BEX 1312/99-5 and CNPq grant 479206/2006. This work is also a result of the project CollaBaWue supported by the German state of Baden-Wuerttemberg. CollaBaWue is part of the research association PRIMIUM. The authors would like to thank Li-Te Cheng, David Millen, and John Patterson for their comments on earlier versions of this paper.

References

1. Garg, P.K., and M. Jazayeri, eds. Process-Centered Software Engineering Environments. 1996, Los Alamitos, CA: IEEE Computer Society Press.
2. Parnas, D.L., On the Criteria to Be Used in Decomposing Systems into Modules. Communications of the ACM, 1972. 15(12): p. 1053-1058.
3. Dart, S., Concepts in Configuration Management Systems. Proceedings of the 3rd International Workshop on Software Configuration Management. 1991, Trondheim, Norway: ACM Press. 1-18.
4. Grinter, R., Supporting Articulation Work Using Configuration Management Systems. Computer Supported Cooperative Work, 1996. 5(4): p. 447-465.
5. de Souza, C.R.B., et al. From Technical Dependencies to Social Dependencies. in Workshop on Social Networks for Design and Analysis: Using Network Information in CSCW. 2004. Chicago.
6. Trainer, E., et al. Bridging the Gap between Technical and Social Dependencies with Ariadne. in Eclipse Technology Exchange. 2005. San Diego, CA.
7. Froehlich, J., and P. Dourish. Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. in International Conference on Software Engineering. 2004. Edinburgh, UK.
8. Cheng, L.-T., et al., Building Collaboration into IDEs. Edit -> Compile -> Run -> Debug ->Collaborate? in ACM Queue. 2003. p. 40-50.
9. Sarma, A., Z. Noroozi, and A. van der Hoek. Palantír: Raising Awareness among Configuration Management Workspaces. in Twenty-fifth International Conference on Software Engineering. 2003. Portland, Oregon.
10. Carmel, E., Global Software Teams: Collaborating Across Borders and Time-Zones. 1999: Prentice-Hall.
11. Herbsleb, J.D. and D. Moitra, Global software development. IEEE Software, 2001. V18(N2): p. 16-20.
12. Meyer, B., The Unspoken Revolution in Software Engineering. IEEE Computer, 2006. 23(1): p. 121-124.
13. Strauss, A. and J. Corbin, Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory. Second ed. 1998, Thousand Oaks, CA: SAGE Publications.
14. McCracken, G., The Long Interview. 1988: Thousand Oaks, CA: SAGE Publications.
15. de Souza, C.R.B., S.D. Basaveswara, and D. Redmiles. Supporting Global Software Development with Event Notification Servers. in Workshop on Global Software Development. 2002. Orlando, FL.
16. Callahan, D., et al., Constructing the Procedure Call Multigraph. IEEE Transactions on Software Engineering, 1990. 16(4): p. 483-487.
17. de Souza, C.R.B., et al. How a Good Software Practice Thwarts Collaboration—The Multiple Roles of APIs in Software Development. in Foundations of Software Engineering. 2004. Newport Beach, CA: ACM Press.
18. de Souza, C.R.B., On the Relationship between Software Dependencies and Coordination: Field Studies and Tool Support, Department of Informatics, Donald Bren School of Information and Computer Sciences. 2005, University of California, Irvine. p. 186.
19. Wasserman, S., and K. Faust, Social Network Analysis: Methods and Applications. Structural Analysis in the Social Sciences. 1994, Cambridge, UK: Cambridge University Press.
20. de Souza, C.R.B., J. Froehlich, and P. Dourish. Seeking the Source: Software Source Code as a Social and Technical Artifact (to appear). in ACM Conference on Group Work. 2005. Sanibel Island, FL, USA.

21. Murphy, G., et al., An Empirical Study of Static Call Graph Extractors. ACM Transactions on Software Engineering and Methodology, 1998. **7**(2): p. 158-191.
22. Gamma, E., et al., Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. 1995, Reading, MA: Addison-Wesley.
23. Lindvall, M., and K. Sandahl. Practical Implications of Traceability Software—Practice and Experience, New York: John Wiley & Sons, Inc., 1996, 26, 1161-1180.
24. Augustin, L., D. Bressler, and G. Smith. Accelerating Software Development through Collaboration. Proceedings of the 24th International Conference on Software Engineering (ICSE'02), ACM Press, 2002, 559-563.
25. Robbins, J.: Adopting Open Source Software Engineering (OSSE) Practices by Adopting OSSE. In: J. Feller B. Fitzgerald, S. A. Hissam, and K.R. Lakhani, (ed.): Tools Free/Open Source Processes and Tools. Cambridge, MA: MIT Press, 2005, 245-264.
26. Rodriguez, F., M. Geisser, K. Berkling, and T. Hildenbrand. Evaluating Collaboration Platforms for Offshore Software Development Scenarios. Proceedings of the First International Conference on Software Engineering Approaches For Offshore and Outsourced Development, Zurich, Switzerland, 2007.
27. Dutoit, A.H., R. McCall, I. Mistrik, and B. Paech (ed.) Rationale Management in Software Engineering Springer Verlag, 2006.
28. Boehm, B. Value-Based Software Engineering Software Engineering Notes, 2003, 28, 1-12.
29. Egyed, A., S. Biffl, M. Heindl, and P. Gruenbacher. A Value-Based Approach for Understanding Cost-Benefit Trade-Offs During Automated Software Traceability Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE '05), ACM Press, 2005, 2-7.
30. Heath, C., and P. Luff, Collaboration and Control: Crisis Management and Multimedia Technology in London Underground Control Rooms. Computer Supported Cooperative Work, 1992. 1(1-2): p. 69-94.
31. Grinter, R.E. Doing Software Development: Occasions for Automation and Formalisation. in Fifth European Conference on Computer Supported Cooperative Work (ECSCW'97). 1997. Lancaster, UK: Kluwer Academic Publishers.
32. Robertson, G.G., S.K. Card, and J.D. Mackinlay, Information Visualization using 3D Interactive Animation. Communications of the ACM, 1993. 36(4): p. 57-71.