Original paper

Research in Engineering Design 14 (2003) 75–88
DOI 10.1007/s00163-002-0027-8

Capturing communication and context in the software project lifecycle

Vera Zaychik, William C. Regli

Abstract Capturing design process knowledge is a complex multidisciplinary problem. The advent and increased acceptance of digital computer-supported cooperative work tools enabled us to study how engineering collaboration might be captured and archived to support engineering lifecycle activities. To improve project communications among software engineers and create structured e-mail archives, we developed an environment called CodeLink. CodeLink is a design rationale support tool, integrating e-mail–based collaboration with the software development process, providing teams of developers with a means to automatically associate specific code elements with e-mail messages. In this paper we describe CodeLink's architecture, user interface, and the results of an informal user study. We believe that by integrating collaborative work tools with development tools, we can enrich the communication within engineering teams and build repositories that detail collaborative decisions made in the development process. These repositories can then be used to facilitate software maintenance and extract design rationale.

Keywords Collaborative design, Computer-supported cooperative work, Software design, Software project management, Software maintenance

Received: 31 July 2002 / Revised: 14 November 2002
Accepted: 21 November 2002 / Published online: 15 February 2003

V. Zaychik, W.C. Regli (✉)
Geometric and Intelligent Computing Laboratory,
Department of Computer Science, Drexel University,
3201 Chestnut Street, Philadelphia, PA, 19104, USA
E-mail: regli@drexel.edu
Fax: +1-215-8951582

Present address: V. Zaychik
Advanced Technology Laboratory,
Embedded Processing Laboratory, Lockheed Martin,
A&E Building, 2 West, 1 Federal Street, Camden, NJ, 08102, USA

This work was supported in part by National Science Foundation Graduate Research Fellowship and Knowledge and Distributed Intelligence in the Information Age Initiative Grant CISE/IIS-9873005. Additional funding was provided by the Office of Naval Research under Grant N00014-01-1-0618.

1 Introduction

Software engineering project knowledge is contained within source code, documentation, requirements and design documents, bug databases, communications between developers, and the memories of individual developers. Project communication and collaborative exchanges contain a great deal of knowledge about design intent, rationale, and decision-making that could be harvested to improve project management and facilitate software maintenance. However, it is very difficult to extract useful information from informal communications because these media (i.e. e-mail or voice conversations) are often poorly structured or not captured.

While electronic means of communication between software engineers (e.g. e-mail, instant messaging, computer-supported cooperative work tools) have become widely used, they still lack support for certain important factors of the software development process. For example, in open-source projects, where participation is open to developers around the world and is not restricted to any geographic area, e-mail is the dominant communication medium. This is part of a fundamental trend toward increased use of asynchronous communication instead of face-to-face meetings: rather than walk to the next cubicle, we place a request for information at our convenience, which does not have to be satisfied immediately by the recipient. While e-mail tools have improved significantly over the years, they still lack the ability to provide the kind of *context-based* support to the collaborative work process found in a face-to-face meeting.

This paper presents an approach to enable context-aware e-mail collaboration among software developers. Informally, context is the sum of information we readily obtain from the participants by paying attention to our surroundings and nonverbal cues: the exact subject of conversation, turn-taking, and so on. It is the collection of circumstances or conditions under which the communication act occurs. For example, in face-to-face conversations, when we want to specify something, we can often point. Such facilities are not available in e-mail. For the most basic example in e-mail, when the need arises to specify a point in a referenced document, the sender is often forced to describe the location (e.g. third paragraph, second line or class Foo, function Bar, line 154).

Informal communication contains a great deal of project-related information, often information not found anywhere else [39]. The more developers rely on electronic communications, the greater the quantity of data that can

be available from the communications for future access. Organizations often maintain e-mail archives of mailing lists and e-mail discussions in parallel to the software project files and documentation. Unfortunately, they provide insufficient search capabilities: the information might be there, but most people are not willing to sift through hundreds of messages for the relevant few that deal with a particular part of the project.

Our approach provides developers with the ability to create *anchored conversations* [8] about their software project by automatically including *context hyperlinks* to specific places in a software project source file or document. This, in turn, stems the loss of vital project information: a message is archived based on the developers' context at the time it is sent. Our research aims to study the following questions:

- Can we automatically extract useful project information from e-mail communications between designers with minimal interference while avoiding text parsing and analysis, instead using context extraction at the time of composition of the message?
- Can we improve communication between developers by providing tools that allow them to see the relationships between code elements and communications?

To study these issues, we developed a software tool called CodeLink. CodeLink automates the extraction of project context and uses it to augment e-mail–based collaboration as well as build a repository of collaborative life of the design project. CodeLink does this by incorporating references/links to specific parts of source code files into an e-mail message using formal representations of the collaborative process and design context. When developers compose project-related e-mails, the source code files they are working on are analyzed to extract semantic information (e.g. to what functions are the developers referring?), and a code snapshot is taken that is archived in a dedicated concurrent versions system (CVS) [13] repository. To formalize the representations of message context and contents, we developed a message ontology based on the DARPA Agent Markup Language (DAML). All e-mail messages containing code context links inserted in the above-described manner are archived in a repository and indexed using extracted semantic information. These communications can be annotated, linked to any URL, and arbitrarily grouped. By using a semantically grounded language like DAML, we can use DAML-based inferencing tools to later reason about the captured design process. Figure 1 provides an overview of the overall system architecture.

This work introduces a new way to actively integrate computer-supported cooperative work (CSCW) tools, such as e-mail, with the software design development process. Using this approach, relationships between code changes and collaborative discussions about code can be created and used by developers and project managers to better understand the thought processes that went into the creation of a system. We believe that our approach to representation and active capture of collaborative work can have applicability in other "engineering design" domains as well, and this point is addressed in [20, 40, 41].

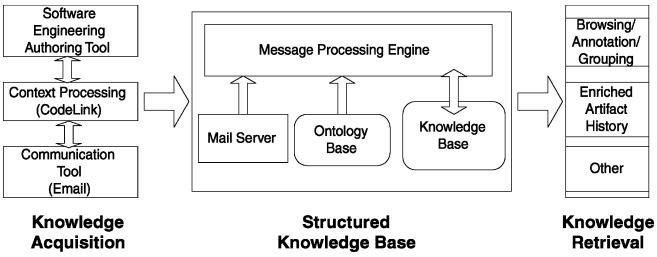

Fig. 1. General approach to the problem

This paper is organized as follows: Sect. 2 provides an overview of related work; Sect. 3 describes CodeLink in detail; while Sect. 4 describes a user study we conducted. Section 5 outlines limitations of the current approach as well as future research directions.

2
Related research

We briefly describe related research in CSCW, context awareness, software design and engineering methodology, and design rationale that has influenced this project.

2.1
CSCW and context awareness

E-mail has become the primary message tool used by 97% of North American knowledge workers on a regular basis. It has outpaced other media as the preferred way to receive or give input during work (66% versus 13% for face-to-face meetings and 12% for phone) [16]. Mailing lists are widely used to coordinate open-source projects [12], but e-mail has also become common when simply communicating with someone in the next office, or even in the same office [21, 9]. Yet, it still presents many problems when used for certain tasks, as it is not capable of handling social context [37], workflow, and negotiation.

One aspect of communication still not well supported by asynchronous tools is context awareness. In face-to-face interactions, a great deal of information is expressed by using nonverbal cues, which help in putting participants "on the same page" [9]. These cues are usually classified as *context*. Context availability can improve search capabilities in computer databases by enabling more precise queries, which can result in enhanced recall and precision of results. Thus, it is important to obtain the context of e-mail communications at the time of their creation. Then, search of archived messages can be performed based not only on text parsing, but also on context.

Most context-aware applications have been developed in the domain of mobile and wearable computing and in domain-independent groupware, but there are some exceptions. The closest relative to CodeLink is Anchored Conversations [8], an application-independent tool for collaborative authoring that provides a chat utility with anchors, which act as substitutes for deictic references (referring to items using words like *this, those, here, there*). However, the notion of context is used very narrowly due to the domain-independent nature of the tool: only enough information is extracted about the environment to allow these anchors to be placed unambiguously.

A major weakness of existing CSCW applications is that users have to be introduced to new tools, instead of well-known or common ones. This results in a disruption of the design process and possibly in flat rejection of the tool. To deal with this problem, Grudin [19] suggests building on existing and accepted tools where possible. A custom e-mail client that demonstrates a particular functionality is not likely to include most other features of popular applications such as Outlook (Microsoft) or Communicator (Netscape). Another common problem is that for such tools to be successful, they must be adopted by all members of the group [19]. This is commonly referred to as *critical mass* or *the prisoner's dilemma*. In this research, we added functionality to a commonly used e-mail system to increase the chances of acceptance and adoption.

2.2
Software design and software engineering

Despite considerable recent research and industrial activity, the software engineering process remains more of an art than an engineering science. Several research directions have been pursued to improve the existing situation. One such direction is *process modeling*, which aims to define some policies and rules to be followed in a process, and then create a system to enforce those rules. The resulting systems are called process-centered software engineering environments (PSEEs). Such systems usually provide a formal modeling language to define the policies and rules and a set of integrated tools to enable the process within the set framework. There are several exemplary systems of this kind, for example, SPADE-1 [1], OIKOS [30], EPOS [11], Promo [14], and PRIME [32]. Promo is an unusual approach as it is a virtual environment based on MOO concepts.

However, the above-mentioned PSEEs generally do not support communication, especially synchronous communication. SPADE-1 supports only asynchronous communication through integration with an ImagineDesk toolkit [2]. Even in that case the communication is not an integral part of a software process, but rather another tool available during the process. On the other hand, the CodeLink system considers all communication to be an important part of the process that requires tighter integration with development tools than what is currently available.

Other relevant works in the field of software engineering and methodology describe techniques and approaches to integrate different tools more efficiently and effectively. Engels et al. [15] describe an approach to building integrated software development environments using attributed graphs to model and implement object structures such as software documents and their relationships. The resulting approach is suited for environments developed from scratch, but not as well suited to integration of already-existing tools. Greif et al. [18] describe authors' experiences implementing a distributed collaborative editing system (CES) using a special-purpose programming language, Argus. In comparison, CodeLink integration is based on specific engineering tools and the specific needs of developers. The formal part of our integration process uses a structured modeling language to describe the context information extracted during the communication.

2.3
Design rationale

Over the past decade, and enabled by the Internet, software projects have become increasingly decentralized. Software projects quickly grow too large to be handled locally, forcing outsourcing of development activities to remote locations. Combined with high turnover rates, which deplete the human expertise and knowledge accumulated over a long period of time, the problem of coordination is becoming more and more complex. One issue is how to preserve and communicate to others the *how* and *why* of development, information that is very valuable during the maintenance and evolution of software. The term most widely used in the research literature to describe these concepts is design rationale (DR).

Documenting DR during the design process has been shown to be vital to improved correctness and speed in both the engineering [4] and the software [3] domains. In areas that mainly use ad-hoc tools, such as open-source projects, the need is felt most strongly: although mailing lists of communications between developers are archived and available online, the lack of structure of these archives creates a barrier to effective retrieval and management, and thus an entry barrier for new developers to join projects [12].

Design rationale is an explanation of why an artifact, or some part of an artifact, is designed as such [26]. It includes all the background knowledge about the artifact, such as trade-offs, decisions made, alternatives considered, and deliberation. Many systems and approaches to DR have been created over the years. All approaches in current research can be categorized along the lines of the data is structured, captured, and retrieved, and whether the design is process- or feature-oriented [35].

Where the domain and problems are vague and/or poorly understood, process-oriented systems are used. Process-oriented systems treat DR as an explicit history of the design process. In feature-oriented software systems, the notion of "task" is emphasized, and specific knowledge and rules of the domain are used. The design rationale is stored and represented using either an argumentation-based or description-based approach. The argumentation-based approach uses graphical representations for storing arguments about the artifact. This is the most widely used approach, and many different argumentation models have been created. The descriptive approach records the history of design steps (who did what, and when) and is mostly used in dynamic design domains for solving vague problems. Whichever representation is used, all information capture happens either via user intervention or automatically, or by some mixture of both. The approach described in this paper focuses on automatic procedures, which are described in more detail in Sect. 3.2. In other systems, the retrieval of design rationale is usually done by some combination of navigation and query by the user, but several systems in the literature also use automatic trigger methods to inform the designer of the availability of information. The latter method can be implemented as an agent looking over one's shoulder and critiquing the design, or it can be simply a notification of the availability of additional information.

Although most current DR systems are either completely generic or tailored to solve specific engineering or architectural design problems, several systems have been developed for software engineering. Comet [28], a commitment-based system for sensor-based tracker software, uses explicit representation and reasoning with commitments to aid the software development, especially when considering reuse of or change to a certain module. Comet analyzes source code to get commitments (structure and behavior specifications of modules) to perform impact analysis. Developers can also explicitly state commitments. The cooperative maintenance network-centered hypertextual environment (COMANCHE) [6], a multiuser language-independent environment for cooperative maintenance, allows different programmers to concurrently access and manipulate information related to maintenance requests, the design and implementation decisions made, and their motivations. It allows programmers to annotate any form of textual documents to provide *rationale in the small*, that is, rationale concerned with implementation activities (as opposed to *rationale in the large*, which is concerned with design activities). The process and product information system (PPIS) [29], an information and browsing system for software design and evolution, provides a general-purpose hypertext environment. Designers can manipulate objects and attach links and annotations to them.

Since design rationale research originally began with the argumentation approach, most systems to date rely on user intervention to gather information. This approach has met with limited success, however, because it demands substantial time and attention to enter information [7], or alters the design process [10]. Designers are reluctant to document their actions during the detailed design process [17], and there are significant difficulties in getting them to use argumentation schemas to structure their thinking during real design tasks [5]. As a result, several automatic design rationale capture systems have been developed in the last ten years.

There are two general approaches to automatic design rationale capture. The first approach creates a system specialized for a certain domain with well-defined semantics and/or places certain constraints on the design process. The software architecture analysis method (SAAM) [25] is a scenario-based method for impact analysis of software systems. SAAMPad [36], a meeting room environment centered around an electronic whiteboard, augments the SAAM for capturing the architectural rationale of an evolving software project through automated capture, integration, and access to the discussions and artifacts produced during the SAAM session. A session consists of a discussion by three to ten people centered around the architectural diagrams on the whiteboard, and digital streaming media technology is utilized for recording. Timestamps are used to track important activities occurring during the discussion. SAAMPad supports only the SAAM process, however, and would not be as effective for other methods. Another system, the rationale construction framework (RCF) [31], monitors designer interactions within a CAD environment during the detailed design phase to produce a process history. The user's actions are grouped into meaningful semantic structures, which are interpreted relative to design metaphors to explain the changes to the artifact. For example, the one-to-one part substitution metaphor captures the notion that the designer has swapped one functional component for another. Unfortunately, this method places certain constraints on the design process and relies on manual annotations of artifacts for richer semantics.

Another general approach to automatic design rationale capture is based on the communications perspective. This perspective states that design discourse, i.e. naturally occurring communication among the group members in the process of design, contains design rationale, and that it can be captured without user intervention by recording the thoughts expressed in communication rather than shaping them by requiring explicit argumentation. A study of design teams involved in conceptual mechanical design by Yen et al. [39] showed that formal reports accounted for only 5% of the total noun phrases, while hypermail archives (e-mail) contained 43%. The noun phrase metric for engineering design has been introduced by Mabogunje [27] to assess the design process and predict design team performance. A strong correlation was found between the success of a product as measured by expert evaluation and the number of distinct noun phrases found in documentation. However, the general disadvantage of the automatic capture method is that the recorded information often lacks structure and is difficult to retrieve in a systematic and meaningful manner.

Most communication-based systems allow users to import multimedia data and hyperlinks between artifacts and other data. The result is a web of information with links to requirements, deliberations, simulation, and analysis results. This is an electronic equivalent of a design notebook. The HOS (hyper-object substrate) system [23] provides an environment for computer network design with facilities to import e-mail and news files. The structured rationale is supported through incremental formalization by using simple text analysis and domain knowledge. HOS makes suggestions for formalization to the user for possible links within the acquired information. While the burden of importing relevant information into the system stays with the designer, once the information is inserted, it can be linked to other objects. PHIDIAS [23] provides functionality for two- and three-dimensional graphical design, design argumentation, multimedia information retrieval, and knowledge-based critiquing. It uses graph-based algorithms in a hypermedia network. REMAP/MM [34], a prototype decision support system, provides a graphical interface to synchronous team deliberation and hyperlinks among data records and multimedia objects.

When the communication information captured is not structured in a formal way, but is instead just a web of hyperlinked objects, then it is not really a design rationale environment in the strict sense of the definition. Rather, it is a *design history* environment. The difference is that in design history software, explanations and answers to specific queries about the data are not provided. Rather, the user has to look through the supporting documentation to find the answers. The environment merely provides

a convenient way to attach and later locate the relevant information. OzWeb [24], a hypercode environment for software development, uses WWW technology (HTTP and HTML) to provide access to source code and supporting documentation and allows the incremental addition of links as useful connections are discovered. Recall [39] facilitates and records sketch activity along with video conferencing during conceptual design deliberations. Personal electronic notebook with sharing (PENS) [22] is an authoring client for the Web, that is, an electronic notebook for design notes that requires no knowledge of HTML. It has been used by design teams for group projects in an electromechanical design course at Stanford University. The main features of this tool are that it is lightweight and it works off-line.

The main disadvantage of systems that require the user to import data is that the effort required is too great with no clear short-term value. In order for a DR capture tool to be successful, the users either have to perceive a clear benefit to using the system, or the effort required has to be minimal [19]. We counter this problem by automatically capturing the e-mail exchanges between the developers and providing the ability to include deictic references.

3
CodeLink: approach, architecture and implementation

Our approach is based on the following observations:

1. Designers and software developers resent interruptions and resist process changes. As a result, manual design rationale capture methods have been generally unsuccessful in industry. The goal is to capture process knowledge with minimum overhead and the least interference possible.
2. Automatic capture methods have made some headway, but encounter the problem of lack of structure. Informal communications such as e-mail exchanges are easy to capture, but it is difficult to retrieve needed information from them efficiently. This informal communication is still important because it contains a great deal of process and product information.
3. E-mail applications are domain-independent tools. While this fact has lead to wide acceptance of e-mail in the workplace, it has also caused the loss of contextual information in communication. Users often find workarounds for this problem, but this missing context can result in vagueness and misunderstanding. Some effort is required to insert context information manually.

The goal of CodeLink is to automatically and unobtrusively extract the software development context that should be associated with e-mail–based project collaboration. CodeLink uses these e-mails, along with code snapshots, to build a repository that can be searched in a variety of ways to improve the software development, management, and maintenance process.

3.1
Approach

To access context information within the development environment, we couple the e-mail client with the software development environment. When a developer points to or is editing a certain piece of code, what information is relevant and important to communication about this piece of code? In our approach, there are several important pieces of information that are available for extraction. Not all of them are directly necessary for the immediate goal of referencing a piece of code to the recipients of the communication, but they become important for structuring and indexing accumulated communication data.

A message context C for an exchange between software engineers, is defined as a tuple $C=<P,T,E,t>$, existing at a time t, where P is the project information about which the subjects are communicating, T is the task in which the author of the message is engaged during and shortly before the exchange, and E is the personal environment of the author of the exchange. In turn, P can be defined on different levels of abstraction:

1. On the topmost level, P consists of the project name and location. This can also include the package name if available.
2. When the project is in the development stage (as opposed to design), an additional level is specific file information: file name and version.
3. In object-oriented programming, software is broken up into logical functional units called *classes*. In such cases, class name is also part of the project information.
4. *Function name*: functions are groups of statements performing a particular action.
5. *Line number*: the lowest level about which developers can be communicating is a particular programming statement.

T is the task on which the user is concentrating, his/her actions right before the exchange, other source files being modified. E can be such information as the connection between the exchange and the recipients of the message: the e-mail might be addressed to another employee on the same hierarchy level or to someone higher, a manager perhaps. This might have significance to the exchange and for later retrieval, but it is not easily extracted and even more difficult to analyze. Additionally, a piece of information that can be very useful but is not easily available is the intent of the exchange: is the message a request for information, a reply, or perhaps a notification of change? This information can be somewhat reliably extracted using speech act theory or by requiring the author of the message to specify this explicitly. Extraction of these other important pieces of information is one of our future research goals.

Of the above levels of abstraction of project information, our current implementation considers the following: file name, line number, enclosing function, enclosing class, enclosing package, CVS repository containing the file (project name), and CVS root (project location), i.e. the central location of the repository that is accessed by all developers. The version number is not currently extracted. If CVS version control is not being used for the project, the name of the directory containing the file is assumed to be the name of the project.

While this schema originally supported only Java source files, most programming languages can also map to this schema in some way. Additionally, function, class, and package names can take on “not applicable” values. This is

Field	C++	C	Java	Perl
Function name	return_type *name* (args) {	return_type *name* (args) {	return_type *name* (args) [throws exception] {	sub *name* [(args)] {
Class name	class *name* {	not applicable	class *name* [extends, implements] {	not applicable
Package name	not applicable	not applicable	package *name*;	package *name*;

Fig. 2. Context equivalents for different languages

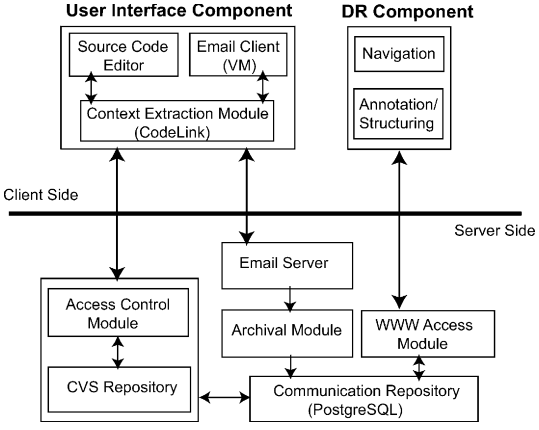

Fig. 3. CodeLink software architecture

Fig. 4. An example of context extraction during the context link inclusion process

due to the fact that the line pointed out by the developer does not have to be part of any function, class, or package. This is the case, for example, if the selected line is the import statement at the beginning of the file. In the same way, C++ code does not have a notion of packages, and that value is always "not applicable." Figure 2 provides the list of languages and context-based cues supported in the current version of CodeLink.

3.2 CodeLink software architecture

The software architecture consists of several server and user modules (Fig. 3).

- On the user side, a context extractor and MIME handler are responsible for enabling sending and receiving messages with references.
- A Web browser allows access to the online browsing/search interface to the communication repository, and to the history of source files annotated with relevant messages.
- On the server side, several services enable the archiving of messages sent, the storage of file snapshots, and the interface to the repository. All server-side components except for the Web interface belong to a Unix user created specifically for this purpose. This user runs all services and is the owner of the CVS and PostgreSQL databases. (PostgreSQL [33] is an open-source object-relational database management system.)

3.2.1 Context extractor

The context extractor, when invoked by the user, analyzes the source code and extracts any relevant information. This information is encoded using a DARPA Agent Markup Language (DAML) ontology and inserted into the e-mail message as a MIME attachment of type DAML/code-link. At the same time, a snapshot of the source file is taken and sent to the CVS access control module using a CVSPUT request. Thus only the context link is attached to the message, and the files themselves are not. Currently, a snapshot is taken every time a piece of code is referenced, even if it is the same piece of code in the same e-mail message. This is not very efficient, and this issue will be addressed in the future. See Fig. 4 for an example of context extraction during the link inclusion process.

DAML is based on the resource description framework (RDF) and the extensible markup language (XML), developed by the World Wide Web Consortium (W3C) and semantic Web communities. DAML provides a set of constructs to flexibly describe knowledge, to create ontologies, and to mark-up information so that it is machine-readable. A main motivation behind the development of this language is to describe information contained in Web pages so that computer agents can read and interpret them; however, it can also be used as a knowledge-representation language. Currently, hypertext markup language (HTML) is used for these purposes, but it is not well suited for computer interpretation and understanding. XML was developed by the W3C so that custom tags can be defined to provide metadata markup. XML is sufficient to describe the syntax of the information, but not the semantics. RDF, on the other hand, can describe semantics, but only on a limited level. For example, only range and domain constraints can be put on the properties, while other restrictions are needed for rich representation. Additionally, sometimes properties of properties need to be specified

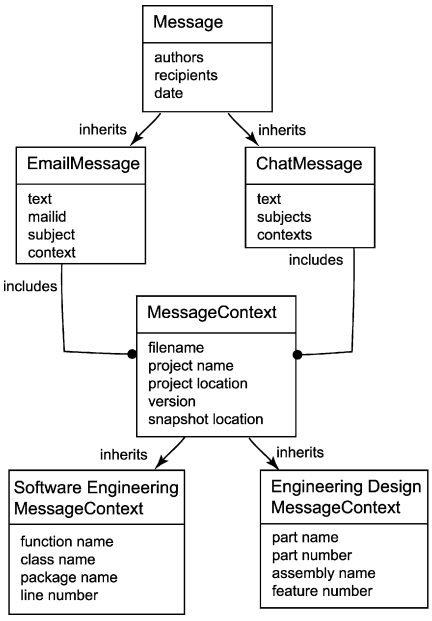

Fig. 5. Message ontology with entities, inheritance, and properties

(for example, to say that a property is unique, transitive, and so on). RDF does not allow for this. Necessary and sufficient conditions for class membership, as well as equivalence or disjointedness of classes cannot be specified using RDF. For all these reasons an extension to RDF using XML syntax was developed and called DAML. DAML uses a concept of namespaces to allow reuse of ontology libraries. This means that when a certain concept is used (class or property), the ontology of its origin needs to be specified. This avoids clashes between different libraries and definitions.

We use DAML for two primary reasons. First, DAML is extremely flexible and extensible. Our DAML ontologies for context and collaboration can be improved and refined, extended to other collaboration modalities (e.g. audio conferencing), and remain backward-compatible with our initial definitions. Second, DAML-based context descriptions are semantically grounded and suitable for use downstream by inference tools that extract design rationale patterns. See Fig. 5 for one possible message ontology with entities, properties, and inheritance hierarchy.

Any number of context references/links can be inserted in any particular e-mail message. Any message containing references to code is automatically forwarded to the archival server. The user can also independently cc: or bcc: any message he/she feels is important to the server itself as it has a dedicated e-mail address.

3.2.2 MIME handler

Once the recipient receives the message, code references can be displayed using a special MIME handler for DAML/code-link attachments. This handler parses the DAML-encoded attachment and sends a CVSGET request to the server specified in the reference. It gets the file back and displays it as an HTML file with a bookmark to the sender's selection.

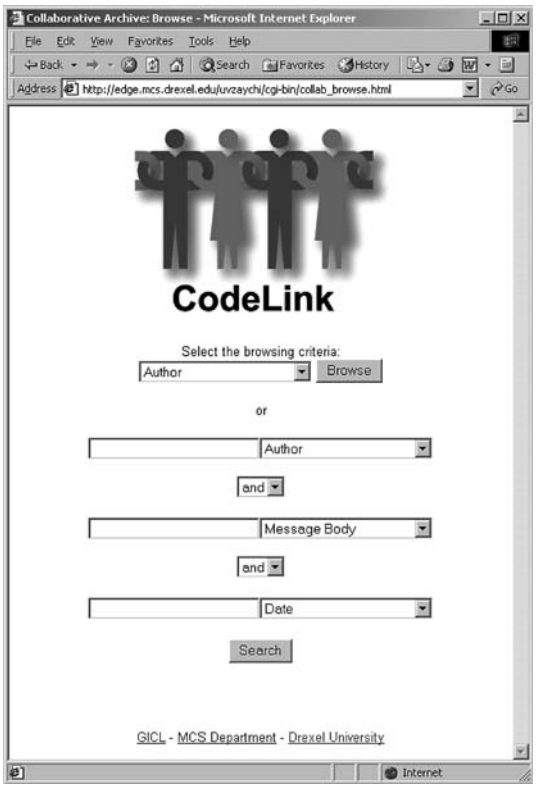

Fig. 6. The main browsing and search interface. A browsing criterion can be selected from the drop-down box. To search, up to three strings combined using and/or can be entered in specified search criteria

3.2.3 CVS access control service

On the server side, a special CVS repository is set up for referenced code files exclusively for storing the instantaneous state of the world at each collaborative exchange (hence, it is not accessed directly by users). Rather, a custom access control service receives CVSGET and CVSPUT requests and fulfills them. When fulfilling a CVSPUT request, the method of retrieval and the version number of the file are returned. The file requested is returned for CVSGET requests. In both cases, if the request is faulty an error is returned to the client. Several client requests can be handled concurrently. CVSGET requests are also received from the Web browsing/search interface.

3.2.4 Archival server

The archival server receives all the e-mail messages from the e-mail server and archives them in the project repository. It parses the messages and the DAML/code-link attachments and saves them to a PostgreSQL database. The repository has a Web browsing/search interface (Fig. 6). This interface supports browsing of messages by author, date, project name and location, package, class, and function names. The messages can also be grouped and then browsed by these groups. The users can add comments and context links to any message in the database. While browsing, the users can narrow their queries if their current browsing criteria return too many matches. See

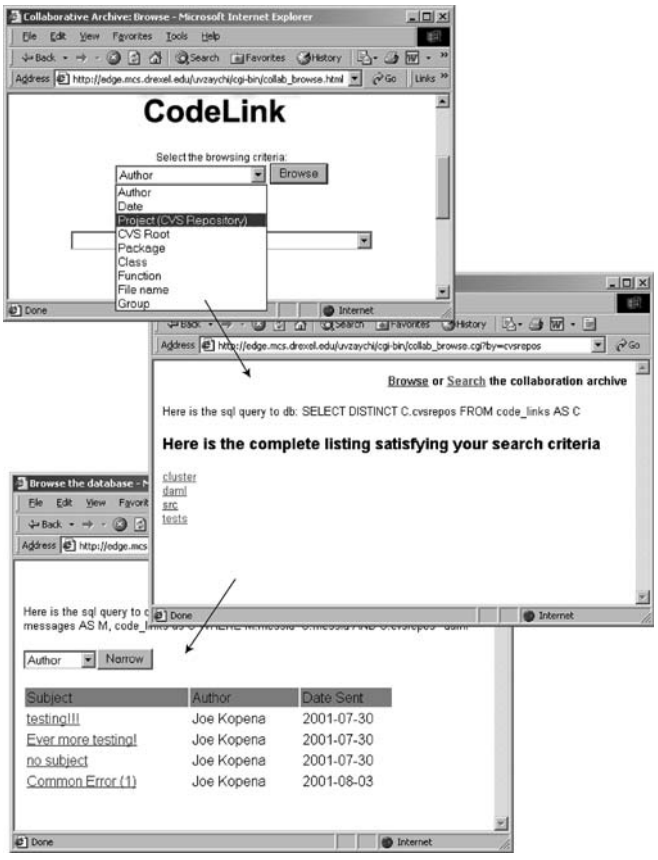

Fig. 7. Example of Web interface browsing by project name

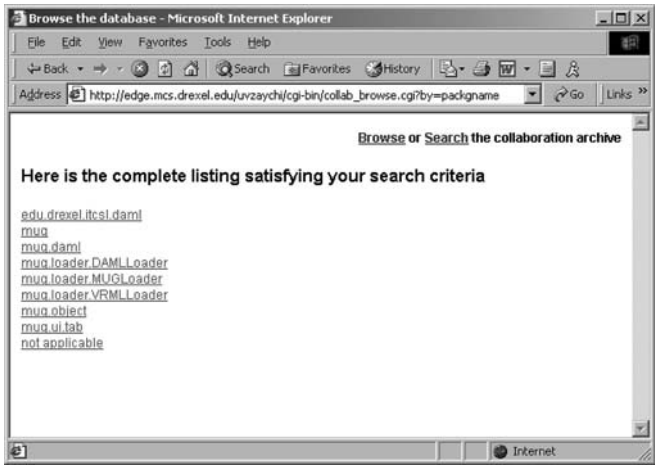

Fig. 8. An example of browsing by package name. Clicking on any package name results in the list of all messages about that package

Fig. 7 for an example of Web interface browsing by project name. Figure 8 shows results of browsing the archive by package name. Finally, Fig. 9 shows an example of a code-link activated through the Web interface.

3.2.5 History-annotated code

Another module annotates the source file with context links to the messages about specific lines of code by interfacing to the message database. This module can be very useful since it maps messages back to the artifact, presenting them in the original context, while not requiring tight integration to or storage in the artifact.

3.3 Implementation

The above-described architecture has been implemented in a prototype called CodeLink currently running in our laboratory. For this proof-of-concept prototype we used the Emacs editor as a source code development environment. Emacs is a very popular GNU application and is widely used. It provides a language called Emacs Lisp (ELisp) with which additional functionalities and modules can be developed and distributed by any Emacs user. Over the years Emacs has grown to include version control interfaces (for example, PCL-CVS is an Emacs interface to CVS), inline Web browsers (W3), e-mail clients (VM), and many other extensions. We use VM [38] as an e-mail client for this project because it runs inside Emacs and is written entirely in ELisp, which makes it very easy to interface with and modify. VM is open-source software that has the usual e-mail client functionality as well as some more advanced commands for tasks like bursting and creating digests, message forwarding, organizing message presentation according to various criteria, and creating rule-based virtual folders. This also means that the module is system-independent: it runs equally well on Windows, Unix, or any other platform as long as Emacs and VM are installed. Although VM is not the most widely used e-mail client, it offers the same attractive qualities as Emacs: ease of integration and extensions. Interfaces of other widely used clients, such as Outlook (Microsoft), Eudora (Qualcomm) or Communicator (Netscape), are limited or difficult to work with.

The context extraction is written in ELisp and uses mode-specific Emacs functions. Currently, Java, C++, C and Perl modes are supported; however, a context link to any type of file can be inserted, in which case the function, class, and package information is not extracted. Figure 2 demonstrates the mapping of context concepts for different languages. In Perl, sub is mapped to function name, and class name is always "not applicable." When linking to C source files, both class and package name are always "not applicable." After extracting language-specific information about the selection, we need to find out whether the current file is a part of some CVS repository. For this, we look for a directory named CVS in the parent directory of the file. If such a directory is found, the file Root states the location of CVS root, and the file Repository gives the name of the repository/project. However, if such information is unavailable, the file is assumed not to be a part of any CVS repository, and the name of the parent directory is used instead. It is assumed that most groups do use some version control system and that CVS is the system of choice for most open-source projects.

It is evident that once the file changes, the context link might no longer be correct as line numbers would change. In order to deal with this problem, a copy of the current state of the file, a snapshot of sorts, is saved and sent to the

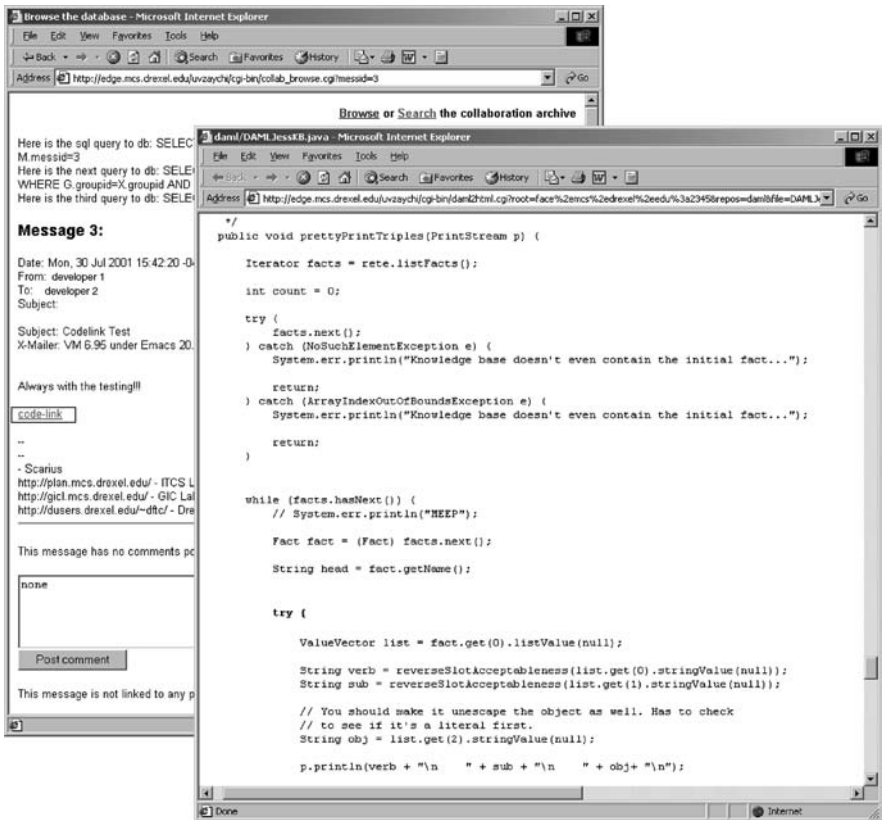

Fig. 9. An example message retrieved through the Web interface and the code-link from the message to the snapshot. The line selected by the author of the message is in *bold*

CVS server. This is achieved by making a direct connection to the custom access control service with a CVSPUT request. The service returns the version number of the snapshot and the exact method with which to get it in the future, which are included in the context link. This way the correct version of the file is always displayed when using links, even when it is different from the current version. The latter is included in the link so that no one central CVS repository is required for all users. As long as the context link contains the method to get to the snapshot, the particular repository used is not important. This implementation is not the only possible one, nor necessarily the best one. Currently, the access control service can become a bottleneck if many requests are dispatched. This can be avoided by inserting the files as attachments directly to the e-mail message, but we decided against such an implementation so as not burden the e-mail message with many different attachments, which can be confusing to the user.

The information thus extracted is encoded using a DAML ontology. The current version of the DAML ontology is located at the GICL Web site[1]. The encoded information is included as a MIME attachment of a new type of DAML, subtype code-link. When the user executes a "send" command on the e-mail message with attachment of type DAML/code-link, such a message is automatically bcc: to the archive. All server-side services except the Web interface are run by a special user on a Unix/Linux platform. This user has an account and an e-mail address. All the messages are forwarded to this e-mail address. In the current implementation we have this special user set up on a Linux server.

3.4 Scenarios

To illustrate how this approach translates into engineering practice, two scenarios of CodeLink in use are presented.

Scenario 1 A group of software engineers is working on a new software project. Developer 1 is working on error correction. Previously, the errors were handled using one error class based on strings. Developer 1 creates several subclasses to handle different errors more specifically and checks the code into the code repository the group is using for version control. Developer 1 then sends an e-mail to all developers to let them know they need to update their code to this new error-handling model. The developer wants to provide code details in the message, refer to specific parts of the implementation, and provide examples of use of the new model. The problem is that there is no easy way to insert context links to specific code instances.

Scenario 2 Developer 1 is assigned a bug dealing with a certain functionality being unavailable in one of the modes of the software. The developer traces the code and discovers that the functionality in question is specifically disabled for that mode, but no reason is given in the comments. The developer removes the restriction but the resulting software produces incorrect results or crashes. It is evident that there was a reason for the original functionality, but where is this information contained? The problem is that all changes to software have underlying reasons, but it is not easy to find such information after the fact.

[1]http://edge.mcs.drexel.edu/uvzaychi/DAML/message-ontology.daml

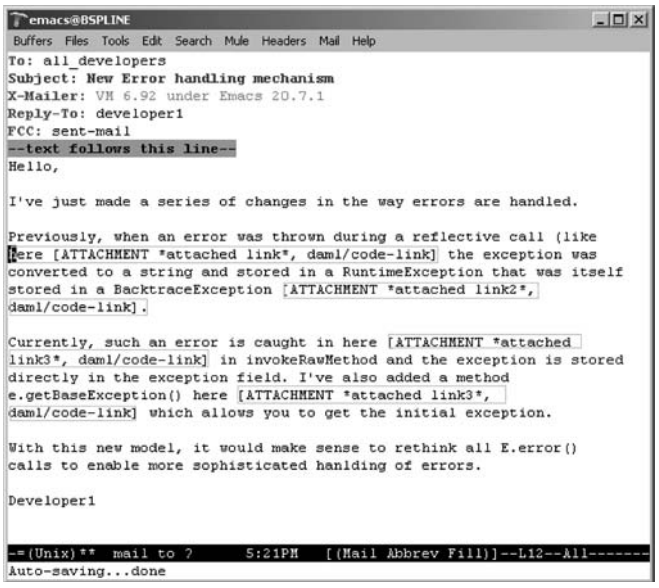

Fig. 10. Screenshot of the e-mail message composed by developer 1 with context links inserted

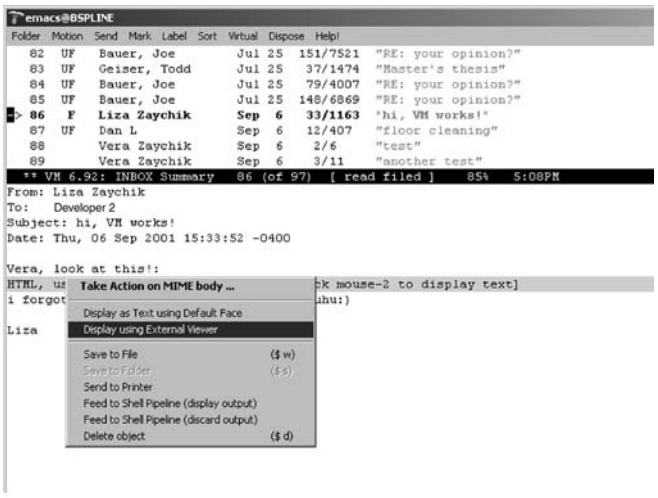

Fig. 11. Displaying code-links using an external viewer

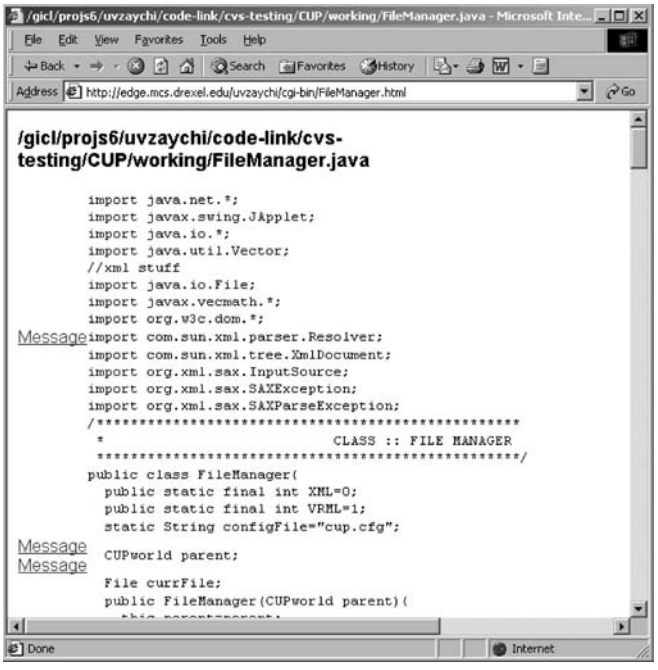

Fig. 12. Screenshot of code with context links to relevant messages

In this work we present a systematic approach to deal with the problems in both of these scenarios. By solving the first problem we find a solution to the second. In scenario 1, the problem that the developer is facing is the lack of an ability to express context information along with content. Scenario 2 describes a more serious problem, missing information. Using our approach and CodeLink, the scenarios can be solved as follows:

Scenario 1 Developer 1 inserts a context link to the old version of the error-handling code and explains why such a model was not sufficient for the project. The developer does that by simply invoking a menu option in the e-mail client and entering the name of the buffer containing the code[2] (Fig. 10). The developer then inserts a context link to the new implementation of error correction using the same method and another link to an example of how errors should be handled in the future. Other developers on the team receive the e-mail and are able to click on the links and see the changes in the code (Fig. 11). The file in question is opened in a browser and jumps directly to the selection made by the originator of the message. Recipients can also look at the example to make sure they understand the new approach.

To extend this scenario, developer 2 finds an inconsistency between the new code and the example and answers the original e-mail pointing to the problem. Developer 2 also has a question as to the overall effectiveness on the new approach. Developer 1 fixes the inconsistency and answers the question of developer 2. Other developers also have minor feedback about the change and answer the original e-mail. Every one of the above-mentioned messages is forwarded to the archival database for maintenance purposes.

Scenario 2 Developer 1 retrieves the history of the source file in question and finds out who wrote the lines disabling the functionality and when. The developer also discovers that several messages were sent about the lines in question at the time of the original implementation (Fig. 12). Developer 1 reviews the messages (like the one in Fig. 13) and discovers that the functionality does not apply to this mode and would not make any sense. One of the messages also contains a context link to the white paper on the subject. Developer 1 then removes the changes he made and also inserts a line of comment in the code explaining the exception. The bug report is closed with detailed explanations and links to the messages in the archive.

[2]Note: this old version of code has to be open in some buffer; however, it is up to the developer to locate such a version.

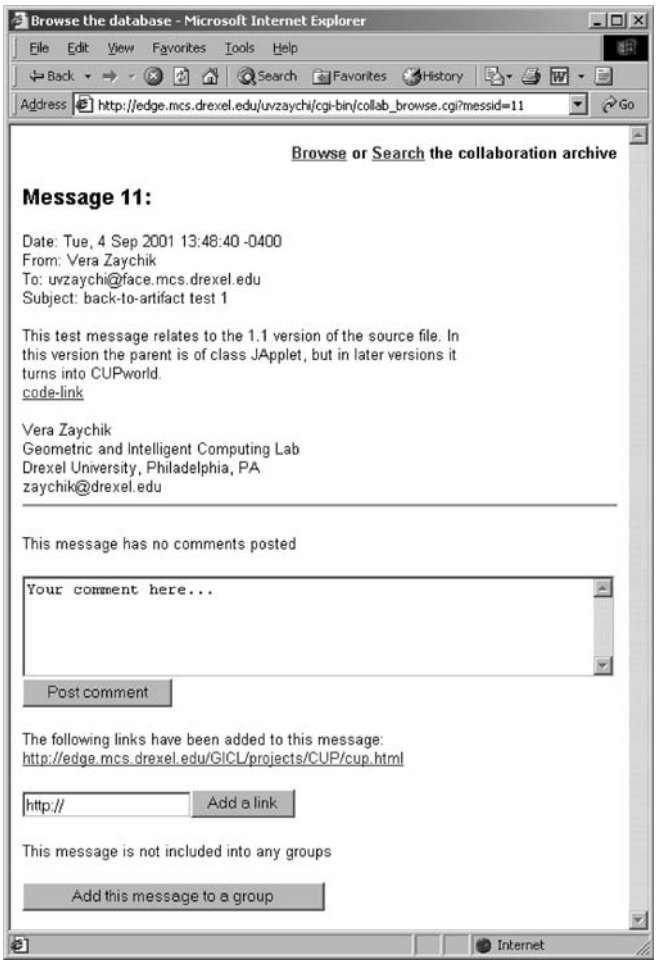

Fig. 13. An example of a message displayed in a Web browser. At the *top* the date, author, recipient, and subject of the message are shown, followed by the body of the message. Any code-link is a hyperlink. Clicking on a code-link results in a new Web browser window showing the link. The message can be annotated with comments and hyperlinks and added to groups

4
User case study

In order to demonstrate the validity of the CodeLink approach, we performed a simple user study. The goal of our study was to assess not only the utility and stability of CodeLink, but also to identify usage patterns: how will software developers make use of such a service? The usability can be assessed by evaluating ease of link insertion and viewing, as well as message browsing and searching. However, the usefulness is more difficult to determine. In long-term user studies, users' tendency to use or not use a system is the ultimate gauge of whether they find the system useful.

Our study was informal and attempted to answer the following questions:

Usability How easy is it to insert a link into an e-mail message? How much time does it take to insert a link? What are the conditions under which a user feels the need to insert a link?

Usefulness Is the system useful to the users in communicating with their team members? Is the system useful in providing information about the development process of a product?

Statistical data How many messages are exchanged by the developers? What proportion of all e-mail messages exchanged by the developers contain links? How many links are inserted in an e-mail message?

4.1
Study design

CodeLink was made available to four teams of undergraduate and graduate student software developers, each consisting of two people working on different projects. The software was installed by the author, and a short training session was conducted. All but one of the users were undergraduate students majoring in computer science, while the other student was an undergraduate minoring in computer science. This means that every software developer participating in the study had at least one year of programming experience, but most had two to five years of experience. All projects were ongoing, and thus certain practices had been established prior to the introduction of CodeLink into the development process. Users were instructed to use CodeLink for project communication in cases when they needed to reference something specific in code or documentation. This means that personal e-mails were not affected, and not all project e-mail messages would involve the use of our software. The author of CodeLink was available at all times to answer questions and to provide technical assistance with CodeLink and also with Emacs and VM. For the duration of the study, all project-related e-mail communication among the users was saved. The study lasted three weeks.

After 3 weeks of access to the functionality of CodeLink, users were given a questionnaire consisting of 16 open-ended questions (see Appendix) and distributed via e-mail. The time required to answer all questions was estimated at no more than 15 min. Every user study participant responded to the questionnaire in full.

4.2
Results

The study resulted in approximately 40 project-related messages with links. Behavior varied across the groups. Two teams exchanged about 20 messages each, while the other teams exchanged almost no messages at all. About half of all messages exchanged included links; usually one link was included per message (see Fig. 14 for the comparative ratio of links in messages).

We believe some straightforward conclusions can be drawn from this study:

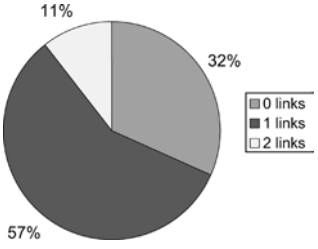

Fig. 14. Percentage of e-mail messages with no, one, or two code-links included

1. Users had no or little problem inserting or displaying links. Some chose to do so quite often. Most of the e-mail messages with links were meant to point out specific pieces of code that needed work, had a bug, or that somebody needed help with.
2. Most user problems were caused by their unfamiliarity with the e-mail client, VM. Only one developer was a novice Emacs user, and that person sent the fewest messages. In short, people better acquainted with the e-mail client and the development environment were more likely to compose messages with links.
3. CodeLink was extensively used in a guru–novice scenario, where one teammate knew the product or programming language to a much greater extent than the other. In this case, the guru used links to refer to specific parts of code for explanation, while the novice used them to ask questions. E-mail messages composed in this scenario were very descriptive and contained a great deal of information about the software code.
4. In several cases users chose to forward an e-mail to the archive manually. Their stated reasons were that it was very easy to do and sometimes they wanted to make sure that the message archive had a copy of the e-mail. This phenomenon was unexpected at the time of the design of the user study since such action required manual effort on the part of the developer, which is generally avoided.
5. The Web archive search and browsing interface was not used by the study subjects. The users indicated that although they thought the archive would be very useful in the long term, they rarely needed to refer to it shortly after the exchange of the messages. We believe that the real utility of the archive would be in long-term software project maintenance and for use by technical management interested in reviewing project process.
6. Users also noted that at times important messages that should be in the archive could not be sent using the linking mechanism because they pertained to high-level concepts and did not deal with any particular lines of code or even files. This issue is to be expected and should be addressed in future work on this project.

4.3 Discussion

The manual forwarding of messages to the archive was an unexpected phenomenon. On the surface, it contradicts our assumption that users avoid and are annoyed by manual capture methods. This case is special because the forwarding process was mostly automated. Users needed to only enter a predetermined e-mail address in the bcc: or cc: field of their e-mail messages. The rest of the archival process took place automatically with no tedious input process required. At the same time, it shows that users lacked understanding of some of the principles of the system: no project information is extracted unless a link is inserted, and thus all manually forwarded e-mail messages will only be partially referenced in the archive.

Perhaps one of the best indicators of the usefulness of the system, continued use by the study subjects after the study period, produced disappointing results. We were not able to prove in the short experimental period that CodeLink was a necessary addition to the developer toolbox. This result confirms the conclusions of other design rationale [35] projects: the acceptance and use of new collaboration and knowledge archive technologies is as much a management and social issue as a technical issue.

The guru–novice use scenario proved to be most useful in generating knowledge that could be used in the future by other members of the group. If real-world integration of CodeLink into the software development process is to take place, management and organizational support is required. The benefits of the system should be outlined to the users, and there should be organizational incentives for the improvement of the development process using this system. This does not mean we advocate organizations and management forcing their employees to use CodeLink, but there should be clear support of any practices that enhance the development process.

The user study did show that, on a small-group project scale, CodeLink helps improve communication among the team members. The link insertion process proved to be intuitive to the users and did not introduce any delay to communication. A large-scale study of the same type should be conducted to gather more reliable statistical data. It is imperative that the e-mail client and the development environment used for the study are the ones that the study subjects already use in their software development process. One of the most important results of the user study is the emergence of the guru–novice communication pattern. This pattern currently promises to produce the most e-mail messages that are useful for information retrieval.

5 Conclusions and future work

This paper presented our work on CodeLink, a tool to integrate e-mail–based cooperative work with the software development process. We believe that this work contributes to existing research in several areas. First, we introduced an approach to integrating generic CSCW tools with software engineering tools to enable the extraction of domain-specific context information in developer communications. This approach is very general and can be applied to other engineering domains as well as to media other than e-mail (i.e. voice or video conferencing). Second, we showed informally that the ability to insert context-specific links into e-mail messages can improve communications among engineers by reducing the time required to specify references to code and by removing ambiguity in text-based code references. This discovery seemed particularly useful in a novice–guru developer situation, where a new employee is learning their way around a large and complex system. Last, the archives of context-based e-mail communications can be used to support the software management and maintenance process by enabling a wide range of queries, groupings, and interrogations in order to find the important patterns and messages that occur during the project lifecycle.

In developing our approach, we created a formal representation of design communication, collaboration, and context using DAML. In addition, we created a set of

software hooks to integrate e-mail with the software development process. Obviously, CodeLink is only a proof-of-concept and not a complete solution. We envision developing true knowledge-based agents that autonomously assist designers during group projects—agents that are capable of sophisticated reasoning about context and collaboration and that can actively aid teams of engineers during complex group work. The CodeLink approach does not have to be a stand-alone application; rather, it can be a part of some bigger suite of tools, such as SourceForge.[3] In fact, any system that includes a mailing list can benefit from integrating an ability to link to code. Hypertext systems would benefit to an even greater extent because more pieces of information can be linked.

Among the limitations to our current approach are, first, that the current prototype system is not very extensible, i.e. other context information pieces cannot be easily added or changed mid-project. Thus the context extraction mechanism and the message archive must be tailored to the chosen domain, in our case, software engineering. This problem might be solved by using more advanced database techniques. At the same time, the DAML ontology would need to be extended to allow for different domain contexts using inheritance mechanisms. Second, and more significantly, the current system assumes developers want to send messages with code references and that the ratio of such messages to all the project messages is high. This is a new kind of functionality, not presently seen in commercial e-mail tools or integrated development environments. We believe that this functionality does not exist because of a lack of integration between (usually generic) CSCW tools and a person's actual work environment. If future user studies prove this assumption to be false, then our approach must be rethought. This ratio depends on many things: how distributed the work process is, what stage it is in, and the roles between the developers. For example, in a mentor–trainee relationship there is a high likelihood of request–reply exchanges with code references in the replies. From our observations in open-source projects, the size of the project and the number of developers make a difference in the ratio of code-related messages. Older and bigger projects usually draw many participants, and the ratio becomes smaller. Overall, only numerous user studies can show how much relevant information can be captured with the approach described in this paper.

Our current work also does not consider security and privacy issues, i.e. developers might feel apprehensive about their messages being archived, since such records might be used against them in the future (to show their incompetence, for example). In the current approach not *all* messages are archived, only those that are known to have project-related information due to the links to code. All other messages are ignored, whether they are personal or project-related. There is a trade-off between having a greater sense of security while using the application and losing project-related messages without links. This problem should be investigated in detail.

[3]http://www.sourceforge.net

We believe the approach illustrated in this paper is generally applicable to other collaboration-intensive engineering domains: mechanical, electrical, civil, and architectural. This point is discussed in detail in [20, 40]. We chose to deploy the approach for a software engineering problem mainly because of the ready access to real software data and programming expertise. Transferring this approach to an engineering domain requires dedicated domain expertise to create the appropriate knowledge representation schema as well as quantities of real engineering data (i.e. getting code for large software systems created by students is much easier than getting professional engineers to part with their proprietary engineering data and work on a prototype user study).

Our plans for future work on this project include several different areas: visualization of the archive, larger and more comprehensive user studies, integration with other communication media, extending and deepening the notion of context, and performing natural language processing combined with different retrieval strategies for explicit extraction of design rationale.

6 Appendix

This appendix includes the information found in Scheme 1, User study questionnaire.

Appendix: User Study Questionaire

```
Please answer all questions to the best of your knowledge and
understanding. It takes about 15 min to complete.

This questionaire is meant to establish usefulness and usability of
CodeLink.

1) How long have you been using CodeLink?
2) What project have you been working on while using CodeLink?
3) Is this a group project?
4) If yes, how many people (including you) are in the group?
5) What methods of communication do you use to contact others in
   the group? (from the most to least used order)
6) How often do you communicate with other members of the group?
7) Since the time you started using CodeLink, how many project
   messages total have you sent until now?
8) Since the time you started using CodeLink, in how many of those
   messages CodeLink was used?
9) When you inserted links using CodeLink, what was the motivation
   behind the action? (i.e. why did you decide to insert that
   specific link in that specific email)
10) Did you have problems inserting links using CodeLink? If yes,
    describe.
11) Did you have problems viewing links using CodeLink? If yes,
    describe.
12) What improvements would you recommend to the developer of
    CodeLink?
13) Did you bcc/cc CodeLink archive manually at any point? If so,
    why?
14) How often do you browse/search the CodeLink message archive
    online?
15) For what purposes?
16) Any other questions/comments are welcome.
```

Scheme 1. User study questionnaire. Please answer all questions to the best of your knowledge and understanding. This questionnaire is meant to establish the usefulness and usability of CodeLink. It takes about 15 min to complete

References

1. Bandinelli S, Fuggetta A, Ghezzi C, Lavazza L (1994) SPADE: an environment for software process analysis, design, and enactment. In: Finkelstein A, Kramer J, Nuseibeh B (eds) Software process modeling and technology. Research Studies Press, Baldock, UK
2. Bandinelli S, Di Nitto E, Fuggetta A (1996) Supporting cooperation in the SPADE-1 environment. IEEE Trans Software Eng 22:841–865
3. Bratthall L, Johansson E, Regnell B (2000) Is a design rationale vital when predicting change impact? A controlled experiment on software architecture evolution. In: Proc PROFES'00, 2nd International Conference on Product Focused Software Process Improvement, Oulu, Finland, 20–22 June
4. Brazier FMT, Van Langen PHG, Treur J (1997) A compositional approach to modelling design rationale. Art Intell Eng Des Anal Manuf 11:125–139
5. Buckingham-Shum SJ, Hammond N (1994) Argumentation-based design rationale: what use at what cost? Human–Comp Studies 40:603–652
6. Canfora G, Casazza G, De Lucia A (2000) A design rationale based environment for cooperative maintenance. Int J Software Eng Knowledge Eng 10:627–645
7. Carroll JM, Moran TP (1991) Special issue on design rationale. Human–Comp Interact 6:197–442
8. Churchill EF, Trevor J, Bly S, Nelson L, Cubranic D (2000) Anchored conversations: chatting in the context of a document. In: Proc CHI2000 Conference on Human Factors in Computing Systems, The Hague, April 1–6, ACM, pp 454–461
9. Clark HH, Brennan SE (1991) Grounding in communication. In: LB Resnick, JM Levine, SD Teasley (eds) Perspectives on socially shared cognition. American Psychological Society, Washington, DC, pp 127–149
10. Conklin JE, Burgess Yakemovic KC (1991) A process-oriented approach to design rationale. Human–Comput Interact 6: 357–391
11. Conradi R, Hagaseth M, Larsen JO, Nguyen MN, Munch BP, Westby PH, Zhu W, Jacchert ML, Liu C (1994) EPOS: object-oriented and cooperative process modeling. In: Finkelstein A, Kramer J, Nuseibeh B (eds) Software process modeling and technology. Research Studies Press, Baldock, UK
12. Cubranic D, Booth KS (1999) Coordination in open-source software development. In: Proc 8th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, Palo Alto, Calif, 16–18 June, IEEE, pp 61–66
13. Concurrent versions system http://www.cvshome.org
14. Doppke JC, Heimbigner D, Wolf AL (1998) Software process modeling and execution within virtual environments. ACM Trans Software Eng Methodol 7:1–40
15. Engels G, Lewerentz C, Nagl M, Schafer W, Schurr A (1992) Building integrated software development environments. 1. Tool specification. ACM Trans Software Eng Methodol 1:135–167
16. Fadden K, Battles S (2000) Messaging for innovation: building the innovation infrastructure through messaging practices. Pitney Bowes, Stamford, USA http://www.usps.com/strategicdirection/_pdf/executiv.pdf
17. Fischer G, Lemke AC, McCall R (1991) Making argumentation serve design. Human–Comput Interact 6:393–419
18. Greif I, Seliger R, Weihl W (1992) A case study of CES: a distributed collaborative editing system implemented in Argus. IEEE Trans Software Eng 18:827–839
19. Grudin J (1994) Groupware and social dynamics: eight challenges for developers. Comm ACM 37:92–105
20. Hayes EE, McWherter D, Regli W, Sevy J, Zaychik V (2000) Software architecture to facilitate automated message recording and context annotation. In: Berry NM, Leif L (eds) Network intelligence: Internet-based manufacturing, Proc SPIE, vol 4208
21. Hollan J, Stornetta S (1992) Beyond being there. In: Proc ACM Conference on Human Factors in Computing Systems (CHI'92), Monterey, Calif, 3–7 May, ACM, pp 119–125
22. Hong J, Toye G, Leifer LJ (1995) Personal electronic notebook with sharing. In: Proc Fourth Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'95), Berkeley Springs, West Virg, 20–22 April, IEEE Computer Society, pp 88–94
23. Shipman FM III, McCall RJ (1997) Integrating different perspectives on design rationale: supporting the emergence of design rationale from design communication. Art Intell Eng Des Anal Manuf 11:141–154
24. Kaiser GE, Dossick SE, Jiang W, Yang JJ (1997) An architecture for WWW-based hypercode environments. In: Proc 1997 International Conference on Software Engineering, Boston, 17–23 May, ACM, pp 3–13
25. Kazman R, Bass L, Abowd G, Webb SM (1994) SAAM: a method for analyzing the properties of software architectures. In: Proc International Conference on Software Engineering (ICSE16), Sorrento, Italy, May, pp 81–90
26. Lee J, Lai K (1991) What's in design rationale. Human–Comp Interaction 6:251–280
27. Mabogunje A, Leifer LJ (1997) Noun phrases as surrogates for measuring early phases of the mechanical design process. In: Proc 9th International Conference on Design Theory and Methodology (ASME/DETC), Sacramento, ASME
28. Mark W, Tyler S, McGuire J, Schlossberg J (1992) Commitment-based software development. IEEE Trans Software Eng 18:870–886
29. Monk S, Sommerville I, Pendaries JM, Durin B (1995) Supporting design rationale for system evolution. In: Schäfer W, Botella P (eds) Proc 5th European Software Engineering Conference (ESEC'95), Sitges, Spain, 25–28 September, Springer, Berlin Heidelberg New York, pp 307–323
30. Montangero C, Ambriola V (1994) OIKOS: constructing process-centered SDEs. In: Finkelstein A, Kramer J, Nuseibeh B (eds) Software process modeling and technology, Research Studies Press, Baldock, UK
31. Myers KL, Zumel NB, Garcia P (1999) Automated capture of rationale for the detailed design process. In: Uthurusamy R, Hayes-Roth B (eds) Eleventh Conference on Innovative Applications of Artificial Intelligence, Orlando, 18–22 July, AAAI, Menlo Park, Calif, pp 876–883
32. Pohl K, Weidenhaupt K, Domges R, Haumer P, Jarke M, Klamma R (1999) Prime — toward process-integrated modeling environments. ACM Trans Software Eng Methodol 8:343–410
33. http://www.postgresql.org
34. Ramesh B, Sengupta K (1995) Multimedia in a design rationale decision support system. Decision Support Syst 15:181–196
35. Regli WC, Hu X, Atwood M, Sun W (2000) A survey of design rationale systems: approaches, representation, capture and retrieval. Eng Comp 16:209–235
36. Richter H, Schuchhard P, Abowd GD (1999) Automated capture and retrieval of architectural rationale. In: Online Proc First Working IFIP Conference on Software Architecture (WICSA'99), San Antonio, Tex, 22–24 February, Kluwer Academic, Dordrecht
37. Sproull L (1984) The nature of managerial attention. In: Sproull L, Larkey P (eds) Advances in information processing in organizations. JAI, Greenwich, Conn, pp 9–27
38. http://www.wonderworks.com/vm
39. Yen SJ, Fruchter R, Leifer L (1999) Facilitating tacit knowledge capture and reuse in conceptual design activities. In: Proc 11th Int Conference on Design Theory and Methodology, Las Vegas, USA, 12–16 September, ASME, DETC99/DTM-8781
40. Zaychik V, Hewett T, Sevy J, Regli WC (2000) Evaluating collaborative engineering environments. In: IEEE Eighth International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Gaithersburg, Maryland, 14–16 June, pp 118–124
41. Zaychik V, Hewett T, Sevy J, Regli WC (2000) Issues in building and evaluating networked engineering environments. In: Cugini U, Wozny M (eds) Fourth IFIP WG 5.2 Workshop on Knowledge Intensive CAD (KIC-4), International Federation for Information Processing (IFIP) Working Group 5.2, Parma, Italy, 22–24 May, pp 259–265