C. Project Description

1. Overview

Imagine one thousand people working on the same program at the same time. Imagine if those people do not know each other and, in fact, do not individually have overall knowledge of the program but merely want to adapt the program to help solve their own problems. Imagine them being able to produce a thousand variations of the program while sharing the same code base.

This is the vision of *ectropic design*. The term e*ctropic* is the inverse of *entropic*. Ectropic software evolves over time to become more highly structured and better able to accomplish the goals of its users. Ectropic design is a design method by which order and structure are created out of the efforts of multiple, unrelated software developers. An *ectrospace* is an active collaboration space to support ectropic design. The goal of the proposed work is to explore the role of ectrospaces in supporting ectropic design.

There is a precedent for our approach, the *Open Source* movement by which unrelated individuals collectively develop publicly-owned software [Raymond]:

> *The basic idea behind open source is very simple. When programmers on the Internet can read, redistribute, and modify the source for a piece of software, it evolves. People improve it, people adapt it, people fix bugs. And this can happen at a speed that, if one is used to the slow pace of conventional software development, seems astonishing. We in the open-source community have learned that this rapid evolutionary process produces better software than the traditional closed model, in which only a very few programmers can see source and everybody else must blindly use an opaque block of bits.*

But Open Source has an inherent limitation. Open Source has succeeded in exactly those situations where a strong individual has been able to provide a unifying force that binds a community of developers together. The **linux** effort (Linus Torvalds), **gcc** (Richard Stallman), and **Python** (Guido Von Rossum) are three examples of this phenomenon. The unifying force serves several roles: a home base where other developers can find the latest "official" version, a moderator for disputes, a repository of collected knowledge, and, most importantly, an enforcer of conceptual integrity. Frederick Brooks in *The Mythical Man-Month* [Brooks, 1982] describes the importance of *conceptual integrity* as follows (emphasis is ours):

> *Even though they have not taken centuries to build, most programming systems reflect conceptual disunity far worse than that of cathedrals. Usually this arises not from a serial succession of master designers, but from the separation of design into many tasks done by many men. I will contend that conceptual integrity is* ***the most important consideration in system design.***

The alternative to conceptual integrity, whether for Open Source or traditionally developed software, is the gradual but inevitable long-term degradation of system structure [Lehman & Belady]. This phenomenon is analogous to what happens in physical systems where the Second Law of Thermodynamics mandates a monotonic increase in system entropy over time. Entropy is a measure of the randomness or unstructuredness or uselessness of the energy in a physical system.

We propose a mechanism whereby the structuredness and usefulness of a software system actually increases over time, even in the face of multiple, unrelated software developers. We call our approach ectropic design. *Ectropy* is the natural phenomenon by which order is added to a system. It is the opposite of *entropy*, which measures the process by which the usefulness of energy diminishes.

One way to battle entropy in software development is to use a rigorous software development process, whereby the participants, the activities they undertake, and the resulting artifacts they produce are all tightly constrained. The use of a rigorous development process has proven successful in many cases. On the other hand, there have also been numerous instances of project cancellations or customer dissatisfaction despite rigorous use of process.

Our proposal examines an alternative to the rigorous use of process. That is, how can software be successfully developed in a situation where multiple individuals collectively and simultaneously enhance a large-scale software system. Our departure from Open Source is to explore what would be required to maintain conceptual integrity in the absence of an individual serving as a unifying force. Specifically, we want to examine the question of how to produce high-quality, large-scale, software generated by multiple, unrelated developers working concurrently

without an individual moderator providing a unifying force.

The main challenge of distributed development is collaboration, including coordinating the activities of developers, the tangible software artifacts they produce, and the key design concepts. We propose to explore the collaboration issues of ectropic design by applying two key ideas.

- **Ectropic Design (feature-oriented design):** Development, instead of proceeding from a predetermined and static design representations, emerges from the continual accretion of new features. New features are defined in terms of the end-user goals they achieve and how the features interact, both statically and dynamically, with other features. Tools are provided to check features for consistency, to merge features, to map between features and system goals, and to debug feature-specific code in terms of design goals.
- **Ectrospaces (intelligent, proactive collaboration spaces):** Software development is performed in a distributed fashion by multiple developers accessing an intelligent and proactive, multi-representational, collaboration space, an *ectrospace*. The collaboration space is *intelligent* in that the server is aware of the ectrospace's contents and can manipulate it. The space is *proactive* in that it can take initiative to make developers aware of both relevant resources and conflicting constraints. The space supports multiple representations and media, all of which are active and manipulatable by developers.

The proposal challenges basic assumptions of traditional, process-based software development. The major assumption that this proposal challenges is that traditional, process-oriented, monolithic software development is the proper way for correct, efficient, usable, and maintainable software to be developed. A second assumption that we challenge is that a single individual (or closely knit team) is required to maintain the conceptual integrity of a software system as it evolves. A third assumption is that software tools must be passive, awaiting invocation by a programmer, before they can contribute to a development effort.

We plan to utilize ectropic design in several domains. Our focus will be on use in educational situations, to facilitate students in learning to develop large and complex systems through asynchronous collaboration. We also plan to use the environment to support Open Source efforts, such as Squeak (http://www.squeak.org) and others. Finally, as ectropic design is not specific to software design, we are also working with other faculty at Georgia Tech to explore use of our tools and methods in other design contexts.

2. Ectropic Software as an Extension of Our Prior Work

Ectropic design builds on the research of the PI's. Spencer Rugaber is an expert on program analysis and development methodologies and tools. Mark Guzdial is an expert on computer-supported collaborative learning environments.

Rugaber is currently PI on two related research projects exploring the evolution of software. The first is the MORALE project funded by DARPA as part of its Evolutionary Design of Complex Software (EDCS) program. MORALE is an acronym standing for Mission Oriented Architectural Legacy Evolution. It combines software development methods including scenario-driven requirements elicitation (ScenIC), reflective autonomous redesign (MESA), architectural impact assessment (SAAM), user-interface migration (MORPH), and legacy software understanding (Synchronized Refinement) [Abowd et al.][Rugaber 1999]. Ectropic design is founded on the integration of ScenIC [Potts] and MESA [Murdock 1998].

The other research project, *Software Evolution and Interleaving*, (CCR-9708913), is funded by NSF in conjunction with the DARPA EDCS program [Clayton]. It is particularly concerned with the detection of *interleaved strands* in software [Rugaber 1995][Rugaber 1996]. Interleaving occurs in software when the implementations of several design ideas (strands) are intertwined in one contiguous area of code. Such occurrences are a natural result of distributed software development.

Guzdial's Career grant, *Integrating Programming into Engineering Education through Context-Setting and Scaffolding (*REC-955048), led to the development of two different computer-supported collaborative learning environments, as well as the development and evaluation of computer modeling tools for different domains. The goal of the proposal was to explore the use of scaffolded modeling environments and scaffolded collaboration environments to improve engineering education. We developed and evaluated modeling environments in both Chemical Engineering and Computer Science, and showed that a focus on making connections between the problem and model spaces (as opposed to explicit programming) was enough to gain learning benefits [Rappin 1997][Guzdial et al 1996][Guzdial Rappin 1996]. This research became the Ph.D. thesis of Noel Rappin [Rappin 1998]. In addition, we created and evaluated two collaboration spaces, CaMILE and CoWeb, which were used in a

variety of domains. With CaMILE, we showed that an anchored approach to supporting discussion and collaboration led to more sustained discussion than traditional newsgroups [Hmelo, Guzdial, & Turns, 1998][Guzdial 1997][Guzdial Turns 1999]. The new CoWeb space has much in common with the ectrospace concept in that it is a flexible collaboration space in which individual users have enormous flexibility in creating, redefining, and extending the space [Guzdial, Realff 1999].

Guzdial has a new NSF grant (started January 1999), *Integrating Learning Across Undergraduate Engineering Curriculum through Technology-Supported Collaboration* (REC-9814770), builds upon the success of the computer-supported collaborative learning environments of the previous research. He and his co-PIs are creating collaboration spaces for computer science, mathematics, and chemical engineering students to use across curricular boundaries on the topic of computer modeling. The goal is to improve transfer and retention by utilizing collaboration spaces to make connections between domains that are not currently being made in the curriculum [Guzdial, Realff et al. 1999]. Through this work, we developed some of the underlying technology, MuSwikis [Spoon Guzdial 1999], which has led to our vision of ectrospaces.

3. Scenario

A scenario may help in understanding our vision:

George, a software-development manager at the Federal Emergency Management Agency (FEMA), runs to his workstation as soon as he hears the news. "A freakish, fast-moving storm, Hurricane Doris, has formed and struck the East Coast of the United States with devastating effects. Rarely does such a powerful hurricane form so quickly and so near the shore. Power and telephone networks are down from Boston to Charleston. Literally millions are without basic necessities." His FEMA news web page outlines ectropic development efforts currently underway including**: Supply Airlift Plans, Reestablish Electricity,** and **Reestablish Telephones.**

He visits **Reestablish Telephones** first. The page discusses software support for a repair-and-reconnect BOT. His screen brings up a diagram of the relevant components of the phone network, with multiple cursors moving across the page and voices explaining access points and damage assessment. He picks off his desktop a repair-and-reconnect BOT that he has used recently, and drops it on one of the access points. The BOT appears on the diagram and doesn't make it far before stopping. "George," says a voice, "it's John over in Electrical. Just saw the BOT death. Problem is that power is erratic even in on-line stations. Some of the power company people are here, as well as a bunch of telco folk. There's not enough juice to power all the subsystems your BOT requires. Guess we should've let you know about that. There's a protocol for checking complete power outages, but not partial ones. I'll update the metaprotocol to check for partial power situations against process needs when a new process is introduced." "Thanks, John," says George.

Then George sees the announcement of a new ectrospace off the **Reestablish Telephones** page. He opens it, and he is immediately surrounded by stereo audio of discussion on algorithm designs, and his space fills with algorithm design tools. "Good to see you George. It's Alan from the UK." "Hey Alan. Thanks for joining in on this! What are we doing here?" "Jane created this space. None of the BOTS are working because of the partial power problem, so we're working with some of the Optimization folks to come up with a new algorithm. We need a Traveling Salesman walk, but one that interacts with Electrical's plans so that our BOTS can repair where possible, but move in as soon as power is available, with a goal of maximum population coverage."

George sees that a number of Traveling Salesmen algorithms are already established: Some are genetic algorithms, some approximation systems, and some bounded-algorithm teams. He sees each on his ectrospace browser as an expandable tree of user-accessible features, providing a hierarchical description of the system's goals. He can expand a tree down to its code level, but for the most part, he only needs to see the top-level goal of what a particular algorithm element achieves and its corresponding constraints. He gets to work setting up communications and constraints between algorithms. He uses an approximation system's output to seed the genetic algorithm generator. Automatically, distributed constraint management merges and tracks goal trees to assure that the results, inputs, and outputs are consistent.

He realizes that the damage was greatest in the urban areas right along the coast, so he begins developing a BOT that uses the new algorithms, but heavily weighted to access rural areas. As he works, a notifier pops up on his screen, "Goal for this space: 'Maximizing population access.'" He opens the metaspace to adjust the reflection goals, so that the goal includes time-to-restoration estimates. His new BOT is able to reach more people by traveling electrical networks where power damage is not so great.

George and his colleagues work on for many hours using the BOT to restore electricity and telephone access to thousands. By morning, critical infrastructure has been raised to the point that more traditional support mechanisms and organizations can move in. But before a hierarchy and plans could be established for the unusual circumstance,

the individuals working through the ectrospace are able to come up with new approaches which meet established goals and enforce activity protocols, and they have already begun solving the problems.

Sometime later, when a similar disaster struck in Europe, Alan remembered the work in the Hurricane Doris disaster. He revisited the space, chose a collection of strands from the resultant code, and generated a new program for use in the European disaster. Alan's team was able to quickly apply the earlier results and build upon it in the creation of their own solution.

The FEMA scenario illustrates several aspects of ectropic software development: distributed, real-time software development, ectrospaces, strand-based code production, and feature-based goal-trees. The combination of these aspects allows separate teams with diverse knowledge and goals to jointly evolve software.

4. Research Questions

The vision of ectropic software raises a variety of research questions that comprise the core areas we intend to explore.

- **Conceptual Integrity:** How can the conceptual integrity of a system be maintained when a system is being actively altered by programmers who are not only not in the same organization but who do not even know each other? What happens when conflicts arise between developers? How can design constraints be enforced? How can an individual developer hope to understand a corpus of code written by a variety of developers with very different ideas about the *right* way to code? How can system documentation be kept up-to-date with respect to fast-changing programs? How can new collaborators come up-to-speed quickly on a system they have no experience with? How can relevant pieces of a software system be determined and extracted or enhanced? How, in fact, can a potential collaborator rapidly determine whether or not an existing software system is relevant to his or her needs?
- **Collaboration:** What informal, interpersonal communications mechanisms can fill the role provided by formal process in an industrial organization? What automated mechanisms can be created to guide and direct efforts? How do participants define the guidance mechanisms themselves? How can developers find willing collaboration partners or relevant existing software components? How do collaborators come to share and grow a common vision for a software system? What mechanisms can be provided to support/enforce this vision?
- **Technical Issues:** How can source code and documentation be managed in light of multiple versions differing both historically and functionally? How can design constraints management be effectively distributed? How can security be provided and enforced? How can large, fast-changing, multipartite software objects be efficiently accessed and updated by multiple, concurrent users? How can the source code, itself, be organized and managed in such a way that multiple versions can effectively share a code base?

5. Approach

We propose to address the issues of ectropic software by the application of two key ideas:

- A new design method (ectropic design) specifically organized to support distributed developers working with incomplete knowledge of a system's overall structure. Ectropic design is the method by which conceptual integrity is enforced during distributed development.
- Support technology in the form of an active collaboration space (an ectrospace) capable of proactively notifying developers when relevant opportunities arise or when design constraints are compromised.

5.1. Ectropic Design

5.1.1. Conceptual foundations

Any software design method must resolve the tension between the real-world structure of a problem and the nature of the computational mechanisms used to address it. One of the reasons for the success of object-oriented (OO) design is that often the results of object-oriented problem analysis can be directly transformed into an object-oriented design, potentially even using the same classes and methods. But people think in terms of the problems they are trying to solve, the tasks they are trying to perform, the goals they are trying to accomplish, and these problems,

tasks, and goals are often hidden or dispersed throughout an OO system.

Ectropic design addresses this problem by combining two key ideas from current software engineering research: goal trees and behavior traces. A *goal tree* is a representation containing several types of nodes. One type of node denotes a system goal; another indicates constraints among goals, such as one goal needing to finish before another can begin. Moreover, other nodes may exist to indicate the methods by which a goal is realized within a design. The hierarchical nature of goal trees leads to a natural breakdown of required high-level functionality in terms of low-level components.

Uses of Goal Trees in Computer Science Research

Goal trees draw from three current areas of software engineering research: task-based user interface design, goal-based requirements elicitation methods, and functional notations used by researchers in the artificial intelligence community concerned with software reflection, diagnosis and redesign.

- **User-interface task modeling.** One school of user interface design researchers [Dix][Stirewalt&Rugaber] suggests representing user interface design requirements in terms of a task tree. The root node of the tree is the overall goal of the system being designed. Child nodes correspond to subgoals. Combinators may be interspersed to indicate subgoal ordering constraints and opportunities for concurrency. Proponents suggest that interfaces designed using hierarchical task breakdowns lead to systems easier for end-users to use because of the direct mapping from their needs to system capabilities.

- **Goal-based requirements representations.** In a similar manner, software-engineering researchers [Dardenne][Potts] have developed goal trees for representing system requirements. Goals can be of several types including maintenance goals, whereby a system is responsible for maintaining the values of certain system state variables, achievement goals in which a system is responsible for establishing a specific system state, and prevention goals used to make sure that specific system states are never entered. Augmenting the goal tree is a domain model in which important system and environmental actors are defined.

- **Reflective redesign.** Artificial intelligence researchers have had the long-term goal of developing systems capable of understanding their own structure and being able to diagnose failure situations and suggest redesigns. Such approaches are called *computational reflection*, in which a software system is capable of reflecting on its own behavior and reconfiguring itself. An example of a system featuring this capability is the Deep-Space One NASA space mission currently on its way to Jupiter [NASA]. In order for reflection to work, the structure of a system must be explicitly represented and available for access at run time. One school of researchers organizes this representation around a functional decomposition of the goals the system is trying to accomplish. The (TMK) representation [Murdock, 1998], for example, consists of three pieces: tasks, methods, and knowledge. Tasks correspond to goals, methods describe the algorithm or code used to accomplish a task, and knowledge is expressed as a domain model describing system actors and important data types. Tasks and methods are interspersed in a tree. Also present are ordering constrained expressed as a finite state machine. As a system built using TMK runs, it keeps track of which task (and, recursively, subtask) is currently being realized. Because of relevant method information is also present, the reflection component can directly relate end-user tasks to system components.

Uses of Behavior Traces in Computer Science Research

The second main idea of ectropic design is behavior traces. A *behavior trace* is a representation of an actual or desired program execution. As such it tends to illustrate the design of a system by providing examples rather than a comprehensive view. Behavior traces play a central role in program debugging. More recently, in the form of message sequence charts, they have come to play a role in object-oriented design.

Below we describe examples of the use of behavior traces from all phases of software: scenario-based requirements elicitation, message sequence charts used as a design representation, scenario-based architectural impact assessment, dynamic program slicing, and interleaved program strand-based reverse engineering.

- **Scenario-based requirements elicitation.** Requirements are difficult to obtain. Users often think they know what they want until they are pinned down. Consequently, it is important for analysts to take an active role in eliciting requirements from users. One way of doing this is to ask users to devise a scenario (descriptive story) of how the system is intended to be used. Constructing the story forces the user to think through issues that might arise, leading to more cohesive and comprehensive requirements. Scenario-based requirements elicitation [Potts] augments this general approach in several ways. One way is to force the user to generate scenarios

systematically, by, for example, asking questions about what might prevent a certain system goal from being accomplished and what the system should do in that circumstance. Another augmentation is to force the scenarios to have a *point*, (relevant to a specific system goal)

- **Message sequence charts.** A *message sequence chart* (also called an event trace diagram) [ITU] is a two-dimensional design representation in which the vertical axis represents the passage of time, and the horizontal access indicates system components. In the graph itself, a horizontal line indicates that a message took place (at the instance of time indicated on the vertical axis) between the two components determined by the horizontal position of the end points of the line. A message sequence chart contains a set of messages ordered by their occurrence on the vertical axis which collectively illustrate one execution of a program. Message sequence charts are one of UML's design representations, and their construction is considered an essential part of the object-oriented design process.
- **Architectural impact assessment.** *Architectural impact assessment* is the process of determining the effect of a proposed system enhancement on the architecture of the system. The Software Architecture Analysis Method [Kazman] performs impact assessments by using traces of desired new behavior (which they call scenarios) and indicating which components of a system are affected by the new behavior. Using the scenarios forces designers to think through potential system uses and how each component of the system are affected by them.
- **Dynamic program slicing.** Behavior traces, in the form of execution profiles, have long been part of the performance tuning and debugging parts of software development. Another use of behavior traces concerns program understanding; that is, how does a software developer making an enhancement to a system come to understand the system and the proper way to make the change. One way of enhancing program understanding is called program slicing [Weiser]. A *(static) program slice* is a subset of the statements of a program responsible for computing a single program output. For example, if a program is computing several results, only one of which is erroneous, then the program defect responsible for the failure must occur in the slice of that output value [Francel & Rugaber]. A *dynamic* program slice is similar to a static slice except that instead of being a subset of the program statements, it is a subset of the actually executed statements [Korel & Laski, 1988]. As such, a dynamic slice may contain more than one occurrence of a statement.
- **Interleaving.** Programs are low-level descriptions of how end-user's goals can be accomplished by a computer. Program designers construct programs with such goals in mind using programming knowledge in the form of *programming plans* [Rich]. The code responsible for implementing a plan is called a *strand* [Rugaber, 1995]. Programmers are also concerned with optimizing a program so that it makes reduced use of resources such as time and memory. To accomplish the end-user goals while minimizing resource use, programmers often combine the use of several plans within the same textual area of a program. Consequently, understanding an area containing the implementations of several plans can be difficult. Reverse engineering research has called this the *interleaving problem* [Rugaber, 1995], and techniques to detect and extract interleaved strands are being developed.

In ectropic design, behavior traces serve to define *features*, and goal trees serve as roadmaps by which a new features can be added to an existing system while retaining its conceptual integrity.

5.1.2. Approach

Ectropic design supports multiple, unrelated designers engaged in the following tasks:

- Locating an existing software system containing needed functionality
- Locating relevant functionality in an existing system
- Determining the overall purpose of a system and how the purpose is realized in the code
- Determining where and how to attach new functionality to the existing system in an ectropic fashion such that overall system structure is maintained or improved

As with any other approach to design, ectropic design takes the form of a representation, a method, and a validation technique.

Representation

Ectropic designs are represented by goal trees with interlaced behavior traces. Nodes in a goal tree take three forms: a system goal or subgoal, an ordering constraint, or a method description. All three types of nodes are expressed with specialized vocabularies. Goals are expressed in terms of a limited set of environmental actors (for that system) and a set of desirable qualities together comprising system state. Constraint nodes also have types indicating the temporal dependencies determining the order in which goals can be accomplished. Method descriptions are expressed in a high-level programming language such as C, SML, or Smalltalk.

A path through a goal tree, starting from the root, satisfying any visited-nodes' ordering constraints and including one or more leaf methods, can be thought of as a behavior trace. That is, the methods on the path form a sequence of operations that the system undergoes to accomplish one of its goals. Viewed conversely, behavior traces can be composed, and, when annotated with default ordering nodes and system goals, can be represented as a goal tree. . An example of a goal tree produced by a reflective-redesign tool we have developed [Murdock, 1998] is shown in Figure 1 in which rectangles denote goals, ovals denote methods, and circles denote constraints.

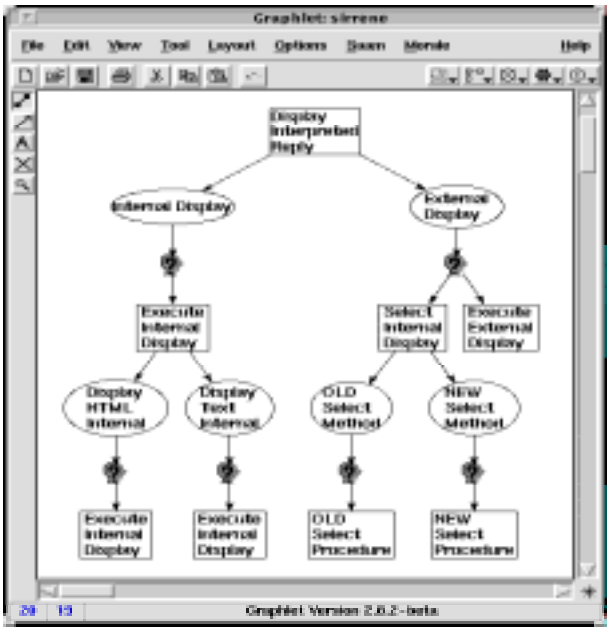

Figure 1: Feature-oriented goal tree with embedded method nodes

As indicated in the section on Conceptual Foundations behavior traces are common to many areas of software engineering, Below in Figure 2 we show an example of one form of behavior trace, a Message Sequence Chart, that is used as part of a dynamic program understanding tool we have developed [Jerding & Rugaber]. In the front panel, the horizontal axis displays system components and the vertical dimension indicates the passage of time. A horizontal line denotes a message transmission between the two components whose horizontal positions correspond to the line's endpoints. Thus the entire sequence of messages (shown in the panel on the extreme right) illustrates a trace of system behavior from which increased program understanding can be gained.

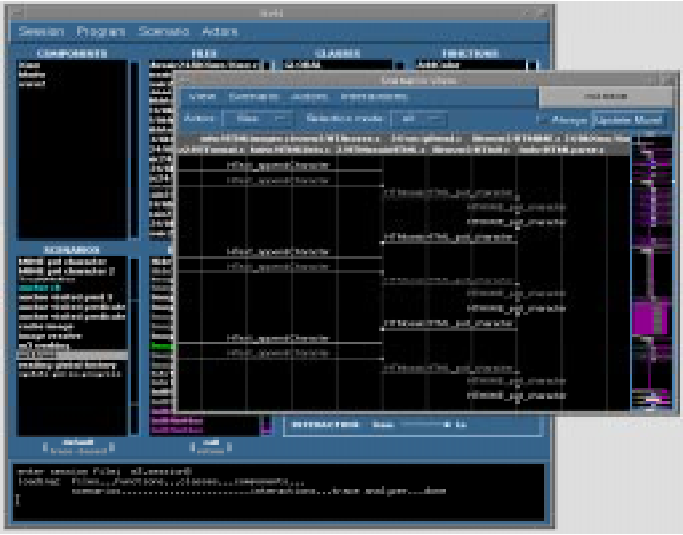

Figure 2: A message sequence chart portraying a trace of system behavior

Method

Given an existing software system, including its annotated goal tree, the ectropic design method for accomplishing the tasks listed above can be summarized as follows.

- Locate an existing software system containing needed functionality: the top levels of the goal tree provide information about the goals the system is capable of performing. The ectrospace provides indexing mechanisms so that search engines can readily locate systems satisfying given goal profiles.
- Determine the overall purpose of a system and how the purpose is realized in the code: the goal tree describes the overall system purpose; interpolated methods describe how the purpose is realized in the code.
- Locate relevant functionality in the retrieved system: the functionality of the system is described in terms of the goals it is capable of accomplishing. As goals are related to the methods used to accomplish them by the goal tree, it is possible to move from goals to code directly. Initial annotation of existing code is done using dynamic slicing [Korel]; extraction of the code is done using techniques developed from research on interleaving [Rugaber].
- Determine where and how to attach new functionality to the existing system in an ectropic fashion such that overall system structure is maintained or improved, assuming the detection and extraction mechanisms of the previous bullet. To do this, the goal tree must be extended to describe new functionality. This is accomplished by interpolating into the goal tree one or more behavior trace describing the new feature. Note that functionality is always described in terms of system goals and the methods for accomplishing them. Consequently, extending a goal tree really means merging two trees. Maintaining conceptual integrity means assuring that large parts of the trees being merged overlap.

Validation

Important ectropic design criteria are correctness, adaptability, and conceptual integrity. As far as correctness is concerned, ectropic design is compatible with a variety of formal specification techniques. For example, if system methods are specified with pre- and post-conditions, then a comprehensive specification can be built by conjoining the specifications for the methods at any particular level in the goal tree [Zave 1993]. To the extent that the programming language used to express the methods has a formal semantics, it can be used to derive the specifications of the system.

Adaptability is the extent to which a system can be easily extended to accomplish related goals. As there are no generally accepted measures of adaptability, we propose to build design critiquing tools capable of pointing out instances in which proposed new functionality is similar to existing functionality structurally, thereby suggesting opportunities to increase ectropy. We also propose to apply refactoring (from object-oriented design [Opdyke 1992])

and subsumption (from automated reasoning [Wos 1984]) to look for opportunities for further increasing ectropy.

There are also no agreed-on measures of conceptual integrity. Coherence, style, and consistency all bear on how integral a system is. Coherence relates to the extent of unification of system goals, which is inherently a domain-dependent measure. Nevertheless, having goals as a first-class design elements, allows developers to judge for themselves. Style is also difficult to measure and inherently qualitative. And style differences will no doubt proliferate under distributed development. But because the system design representation is functionally organized, design elements that are nearby in the goal tree are available for perusal, a necessary if not sufficient condition for style coherence. Finally, consistency expresses the desire that solutions to closely related design problems are themselves closely related.

5.1.3. Status

The vision of ectropic design can be thought of as the integration of emerging trends among software engineering researchers: make system goals first class design elements and describe system functionality in terms of examples of its behavior. The contribution of this proposal is to tie these trends together and to demonstrate how they can be used to support collaborative development. We will build on the conceptual foundations described above but apply them in an innovative manner.

5.2. Ectrospaces

5.2.1. Conceptual Foundations

The core of an Open Source effort is the communication mechanisms for collaboration between the participants. In current Open Source efforts, the communication mechanisms are email, web pages, FTP sites, and similar Internet-based transport mechanisms. The common theme of these mechanisms is generality and relative simplicity. None of these mechanisms are designed explicitly for Open Source, and all of the media are passive conduits for information represented in a few, simple, passive media, i.e., mostly text, graphics, and executable binaries.

An ectrospace differs in three ways from these other transport mechanisms:

- The ectrospace server is *intelligent* in that it is aware of the media present in the ectrospace. It can list them, maintain constraints on them, and track changes to them.
- The ectrospace server is *proactive* in that it looks for opportunities to make connections between the efforts participating in the ectrospace.
- The ectrospace supports *multiple* representations and media in *active* forms. Users in ectrospaces can manipulate computational elements that are active on the user's machine. Active media in the user's space facilitates development, but requires the ectrospace to support rapidly distributing changes.

The following subsections address some of the relevant research that impacts on the ectrospace work.

Most work in computer-supported collaborative work is focused on supporting current practice. The base technologies are powerful, but are mostly focused on distributing media in a synchronous and asynchronous fashion. While we know that CSCW efforts tuned to specific work practices are most effective, none are tuned for the work practices of Open Source efforts.

In our work in collaboration environments, we have explored techniques for providing maximum flexibility on the part of users [Guzdial, 1998][Guzdial & Kehoe, 1998]. We attempt to provide mechanisms for users to easily share media, in structures invented by teachers.

What we are proposing here is revolutionary in two ways:

We are trying to support a new kind of practice which is highly different from even the distributed organizations of the WORLDS project. In Open Source efforts, there is no organization. Any conceptual integrity in the system will come from the ectropic design process and its supporting space and tools.

In no CSCW system does the server really understand what is going on. In ectrospaces, the server understands the components, the manipulations of users, and the constraints that it must apply to insure that conceptual integrity is preserved.

Finally, the concept of a metaspace in which users can collaboratively define their own constraints and the conceptual integrity to be supported in the system is a completely new idea unlike anything existing today in any kind of CSCW software.

Below we present work related to ours in the areas of CSCW, external object systems, active servers, distributed

databases, agents, and constraint management.

CSCW – Computer Supported Collaborative Work

One of the goals of the CSCW community is to effectively support professional work in a social setting. Much of the work has emphasized the creation of network-based computational spaces where participants can communicate and share work (either synchronously or asynchronously). For example, the GroupKit effort [Goldberg & Roseman, 1998][Goldberg, 1997], now TeamWave (http://www.teamwave.com) provides a *room* metaphor for synchronous interaction and where data (such as transcripts of conversation and other media) can be left for asynchronous interaction. The WORLDS project [Fitzpatrick, Kaplan, & Parsowith, 1998][Fitzpatrick, Kaplan, & Mansfield, 1998, http://www.dstc.edu.au] seeks to support distributed organizations through:

- Effortless sharing of diverse media, from whiteboards to Microsoft Word documents
- High-bandwidth communications, including audio and video
- Automatic update of clients' views into the collaborative spaces

One of the big challenges in the CSCW research community is meshing with established work practices. CSCW works best when it is tuned to the dynamics of a given work group. CSCW tools that are meshed with specific work settings do improve efficiency and group performance (e.g., [Osterweil and Sutton, 1996]). However, building specific CSCW tools for each work setting is an enormous undertaking.

The ectrospace is specific in that it is aimed at Open Source style design activities, but it is designed to be flexible across that space of design activities. It can provide some of the same spatial metaphors and ease of exploring multiple media as in GroupKit and WORLDS, but in a cross-platform setting.

External Object Systems

For users to share more than just text, graphics, and indistinguishable binary files, a mechanism for transporting computational objects is required. The objects must be *serialized* (converted to a form that can exist outside of computer memory) and must be *portable* (capable of reuse on whatever client machine is available). Two of the most common mechanisms for supporting this are CORBA and XML.

CORBA (http://www.omg.org) is a standard architecture for accessing objects on other platforms. In CORBA, clients make requests for objects through an ORB (Object Request Broker). The client's request and the returned object are both specified through an interface definition language (IDL) so that language and platform characteristics can be abstracted away.

XML can also be used for serializing objects and transporting them in a portable manner (http://www.w3.org/TR/REC-xml). XML is a human-readable representation of objects. XML Remote Procedure Call mechanisms have been defined and implemented in a number of systems (http://www.scripting.com/frontier5/xml/).

We are currently using a variety of object serialization and distribution schemes in our exploration of ectropic design. We have XML object serialization and parsing schemes in place, as well as an idiosyncratic *ReferenceStream* format. We are also exploring a binary format that enables faster loading and saving.

Active servers

Active servers allow information to be generated using arbitrary programs, instead of just static files. One approach to active servers is to use *servlets*; a servlet is an object that can generate content according to a request (http://www.javasoft.com/products/servlet/index.html). Servlet-based active servers exist for Java, Smalltalk, and Python systems, among others (http://java.apache.org/ and http://perl.apache.org/). However, active servers need not be object-oriented. Common Gateway Interface (CGI) architectures generate content by executing programs, and server-side includes and DHTML extend HTML to include programmatic constructs.

Active server approaches allow much greater flexibility than static files. However, most current approaches are based on HTML and HTTP, which are limited in both display capabilities (HTML rendering) and in networking protocol (client-initiated requests).

Distributed and Replicated Databases

Distributed databases are a relatively well-known technology. Commercial tools such as Lotus Notes have made distributed, replicated databases a safe and consistent technology for a variety of purposes. In our vision for ectrospaces, the granularity of the database is low (e.g., small computational elements such as methods or buttons

are saved individually) and the frequency of the update is high.

Agents

The agent technology most relevant to ectrospaces is the Firefly project where individuals of similar interests are matched [Maes 1994]. In the ectrospaces, the goal is to match on goals and other attributes of the components to identify to developers similar projects where synergy or reuse might take place.

Constraint management

The work of Brad Myers in the GARNET [Myers, 1980] and AMULET projects is most relevant here in that they have developed a set of constraint satisfaction mechanisms that are general and work in real time. In ectrospaces, the constraints are applied to developers' work and their manipulations of the ectrospace.

5.2.2. Approach

An *ectrospace* is a shared multimedia space in which each user has their own replicated portion of the shared space. An ectrospace is composed of segments, which correspond to *pages* or *rooms* in other spaces. Each segment can store any kinds of objects along with their representations, including representations in a wide range of media.

The ectrospace is supported by a central server. The server coordinates publication and storage of ectrospace segments. The ectrospace server is intelligent in the sense that it unpacks serialized objects and applies constraints to them, so that it can provide intelligent updates and other kinds of processing on the components of the ectrospace. Since a single server can support many ectrospace segments, it's possible for the server to identify commonalities between different segment projects and thus serve to be proactive in providing access or information to groups working on similar problems.

Besides the ectrospace segments (pages or rooms), the server also provides access to *metaspaces* where constraints can be defined over the ectrospace. While many of these constraints simply provide for sharing of information and updating of shared representations, the metaspace constraints can also extend to individual operations. In this way, the metaspace allows for the specification of acceptable group behaviors and access privileges.

In this vision for ectrospaces, our goals are the following:

- A user manipulates graphical and computational elements. These elements exist on the user's system, for ease of access, but their change is automatically promulgated throughout all other users. At the same time, users who are dependent on older versions of the computational elements can continue using them until they are ready to upgrade.
- The server knows about all of the components in a segment of the ectrospace. The object serialization process is such that objects can easily be expanded for the sake of identifying them and their versions. The server knows how to replicate elements that are in use by others so that the new versions do not damage existing code, but instead informs developers that the new versions exist. It knows how to make new elements available widely. In Figure 3, two clients are manipulating a set of objects. Each has its own copy of each element. Elements that are used in two different forms (e.g., the colored and uncolored triangles) are replicated on the server. In addition, the server can maintain constraints that protect the conceptual integrity of the system.

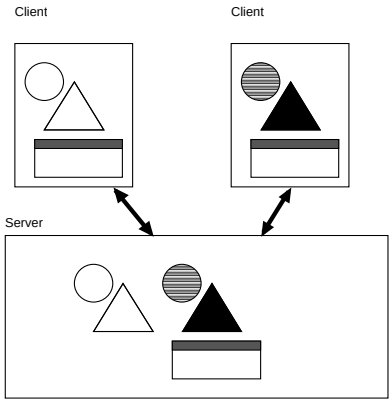

Figure 3: Replicating individual segments onto the ectrospace server

- The server is proactive. It seeks out common elements in different ectrospaces and identifies these commonalities to the developers, so that code may be reused, efforts can be leveraged, or new collaborations can be started. It oversees users' work and matches against constraints defined in the metaspace, so that conceptual integrity is preserved. Agents in the Server will continually review existing work and identify common themes.

These basic capabilities of ectrospaces are enough to enable a wide variety of Open Source efforts. Development tools such as editors and debuggers are simply one kind of computational element to be manipulated and shared. Design representations such as UML diagrams are another kind of representation, and relationships between design representations and other elements such as code are simply specific forms of the general constraint mechanism already described. Merged with the ectropic design elements, ectrospaces enable highly-distributed, Open Source projects where integration and persistent conceptual integrity is facilitated by the server.

We believe that the structure we describe allows for a wide variety of Open Source activities and helps to address the research questions identified earlier. The proactive server can help to connect users and resources across segments and projects. Through manipulation of metaspaces, it will be possible to distribute how users work in a space and how they can maintain appropriate group behavior while sharing group activity. At the same time, the tool is flexible enough to support a wide range of policy mechanisms, even on a project-specific basis.

Validation

Our plan for validation of this approach is to use the ectrospaces to support student group work in undergraduate computer science classes. Students actually find it quite difficult to find time to work at the same time (Hmelo, Guzdial, & Turns, 1998), so they are an accurate testbed for participants working at a distance. Students in these classes are not actually as organized as they might be, and they often need support for their process (Turns, Newstetter, Allen, & Mistree, 1997). The constraint system and other supports for conceptual integrity can serve as learning aids to these students.

5.2.3. Status

We have already built a simple kind of collaboration space, called a MuSwiki [Spoon Guzdial 1999], which supports the sharing of graphical and computational elements in real time with code replicated on the clients' machines. The proposed server's intelligence and proactiveness, however, is a revolutionary step that no one has yet taken. Providing these mechanisms, as well as tools for defining and controlling them in a collaborative setting among diverse users without a single lead, are radical advances in collaboration technologies.

6. Plan

Our research is planned for two years. The goal of work during Year One will be to solidify and document the ectropic design method so that a proof-of-concept prototype can be built. The prototype will include a proactive, real-time collaborative design environment with constraint enforcement and the use of a goal and trace-based design representation. Multiple users in the same collaboration space will create and compose code, graphics, and interface elements with real-time update. Code elements will be integrated with goal and constraint structures that can be directly created and manipulated by the user. Composing code elements will automatically merge goal trees and

maintain constraints across the merged tree. In this way, users will be able to work collaboratively with distributed self-management by defining reflective information that aids in understanding code and maintaining conceptual integrity.

During Year Two, we plan to begin testing in the classroom. We plan to use at least two classes: A Sophomore *Introduction to Design* course, and a Junior or Senior *Software Engineering* class. Both groups will be well-versed in the UML-based design tools we plan to build for ectropic design. The former group of students will serve as a testbed for the usability of the basic system, while the latter group of students will have the knowledge and experience necessary to actually change their metaspace. In both cases, students will be given a features-based definition of a project and an ectrospace with appropriate constraints already in place. We plan to use observation, surveys, and interviews to assess the usability of the system; the effectiveness of the conceptual integrity supports; and the quality of the resultant artifact. We also plan to make our tools available to the Squeak community for their development, as a test of the validity of our approach in an Open Source context.

6.1. Summary of Tasks

TASK (process/tools)	DESCRIPTION
Design Representation Definition	Create the design representations to be used by participants in ectropic design
Comprehensive Manual Exercise	We plan, by-hand, to walk through all of the interacting pieces of the vision we describe. The elaborated scenario will be the driving force for our development efforts
Server Prototype	Intelligent and proactive server design and implementation, including constraint system
Browser/Editor Environment Prototype	First version of ectrospace with connections to Server
Support Tools	UML and other design representation support in browser
Experiment (Validation Project) Design	Preparation for implementation of system in authentic setting
Comprehensive Prototype	Complete ectrospace and ectropic design implementation
Final Report	Description of process and deliverable tools

6.2. Schedule of Milestones and Deliverables

Milestones	Dates
Design Representation Definition	Y1Q2 (Year 1, end of Quarter 1)
Comprehensive Manual Exercise	Y1Q3
Server Prototype	Y1Q3
Browser/Editor Environment Prototype	Y1Q4
Support Tools	Y2Q2
Experiment (Validation Project) Design	Y2Q2
Comprehensive Prototype for Internet Release	Y2Q3
Final Report	Y2Q4

7. Summary

We are interested in the development of "quality, software-based systems" particularly in the context of students transitioning from concerns about programming issues to software engineering issues. We see an opportunity to explore this issue by seeing how Open-Source software development can be adapted and applied to situations in

which it is essential to maintain conceptual integrity, but where no single individual is available to fill the enforcement role.

We expect to learn several things from our investigations:

- The applicability of goal trees and behavior traces to the problem of documenting evolving programs;
- The interplay of feature orientation and object orientation in structuring software;
- The extent to which collaboration environments can replace individuals in enforcing conceptual integrity during distributed software development;
- How automated support environments can aid in the teaching of software engineering principles.

We believe that the proposed research will benefit both the software development community and educator teaching software engineering concepts to undergraduates.