Ectropic Software

Spencer Rugaber[*] and Mark Guzdial
College of Computing, Georgia Institute of Technology

I. Introduction

Imagine hundreds of 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 it in order to solve their own particular problems. Imagine each of them being able to contribute their fixes and enhancements to the program while sharing the same code base.

This is the vision of the Open Source movement, responsible for such popular software as the **Apache** web server and the **emacs** text editor. However, Open Source suffers from two limitations: the inherent nature of all evolving software to lose its design coherence over time and the specific reliance of Open Source on a central individual to coordinate the evolution of the software.

Ectropic software is an attempt to leverage the advantages of Open Source while overcoming the disadvantages. *Ectropy* is the inverse of entropy. Ectropic software evolves over time to become more highly structured and better able to accomplish the goals of its users. It consists of two key ideas: *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.

II. Background

Ectropic software shares a goal with traditional software development: to harness the efforts of large numbers of developers, working simultaneously on the same code base, and sharing a common, high-level vision of the purpose of the software. But what happens if those developers do not know each other, do not share a common understanding of how the vision is realized in the design, and do not even agree on the low-level features to include in the system?

There is a precedent for this situation, 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 best in those situations where a strong individual has been able to provide a unifying force that binds a community of developers together. **Linux** (Linus Torvalds), **gcc** (Richard Stallman), and **python** (Guido Von Rossum) are three examples of this phenomenon. The unifying force serves several roles: providing a home base where other developers can find the latest "official" version, moderating disputes, serving as a repository of collected knowledge, and, most significantly, enforcing conceptual integrity.

Frederick Brooks in *The Mythical Man-Month* [Brooks] describes *conceptual integrity* as follows:

> 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 is the gradual but inevitable long-term degradation of system structure [Belady]. This phenomenon is analogous to what happens in physical systems where the Second Law of Thermodynamics mandates an inevitable and 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.

---

[*] Point of contact: spencer@cc.gatech.edu

One way to combat 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. Unfortunately, there have also been numerous instances of project cancellations or customer dissatisfaction despite rigorous use of process.

Ectropic software leverages the advantages of Open Source while specifically addressing its main vulnerability, its dependence on a single individual to maintain a common vision. The term *ectropic* comes from the word *ectropy*. Ectropy is the natural phenomenon by which order is added to a system.

> Ectropic software departs from Open Source in its concern for maintaining conceptual integrity in the absence of an individual serving as a unifying force. In particular, it addresses the challenge of how to produce high-quality, large-scale, software generated by multiple, unrelated developers working concurrently without the presence of an individual moderator.

III. Issues Raised

The vision of ectropic software raises a variety of research questions.

- **Conceptual Integrity**: How can the conceptual integrity of a system be maintained when the system is being actively altered by programmers who are not only not in the same organization but who do not even know each other? How can design constraints be enforced? How can system documentation be kept up-to-date with respect to fast-changing programs? How can new developers come up-to-speed quickly on a system with which they have no experience? How, in fact, should conceptual integrity be measured in the first place?

- **Collaboration**: What informal, interpersonal communications mechanisms can fill the role provided by formal process in an industrial organization? How do collaborators come to share and grow a common vision for a software system? How can developers find willing collaboration partners? What happens when conflicts arise between developers? 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?

- **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 automated and distributed? How can security and integrity be enforced? How can large, fast-changing, multipartite software objects be efficiently accessed and updated by multiple, concurrent users? How can a potential developer rapidly determine whether or not an existing software system is relevant to his or her needs? How can relevant pieces of a software system be determined and extracted or enhanced?

IV. Solution Strategy

We 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 (*Ectrospace*) capable of proactively notifying developers when relevant opportunities arise or when design constraints are compromised.

A. Ectropic Design

Ectropic Design (feature-oriented design): Development, instead of proceeding from a predetermined and static design description, emerges from the continual accretion of new features. New features are defined in terms of the end-user goals they support and how the features interact, both statically and dynamically, with other features. Tools are provided to check feature for consistency, to merge features, to map between features and system goals, and to debug feature-specific code in terms of system goals.

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 solve 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 users of a software systems do not normally think in terms of classes of objects. Instead, they think 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 diffused throughout an OO system.

Ectropic Design addresses the tension between the flexible and modular structure of well-designed OO software and the perspicuity of functionally organized software. It does this by combining two key ideas from current software engineering research: goal trees and behavior traces. A *goal tree* is a hierarchical representation of a program's teleology (purpose structure). 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 include user-interface task modeling [Dix], goal-based requirements representations [Dardenne], and reflective redesign [Murdock].

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 selective examples rather than a comprehensive view. Uses of behavior traces in computer science research include scenario-based requirements elicitation [Potts], message sequence charts [ITU], architectural impact assessment [Kazman], dynamic program slicing [Korel], and interleaving [Rugaber].

2. Approach

As with any other software design method, 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: system goals or subgoals, design constraints, and method descriptions. All three types of nodes are expressed with specialized vocabularies. Goals are expressed in terms of a limited set of environmental actors and a set of non-functional qualities. Constraint nodes also have types indicating the logical dependencies among the goals. Method are operational descriptions of program behavior expressed in a high-level programming language.

A path through a goal tree, starting from the root, satisfying any visited-nodes' constraints and including one or more leaf methods, can be thought of as a behavior trace. That is, the methods contained in the path form a sequence of operations which the system undergoes to accomplish one of its goals. Viewed conversely, behavior traces can be composed, and, when augmented with constraint nodes and system goals, can be synthesized into a goal tree.

In Ectropic Design, a behavior trace serves to define a feature, and a goal tree acts as a road map by which a new feature can be added to an existing system while retaining its conceptual integrity. Because a single execution of a system typically is intended to accomplish a single goal, a behavioral trace of that execution identifies the part of the system directly related to accomplishing the goal. Overlaying this trace on the goal tree then relates the goal to the its implementation.

Method: Given an existing software system, including its annotated goal tree and selected behavior traces, Ectropic Design addresses the major issues listed in the Issues Raised section as follows.

- To locate an existing software system containing needed functionality: the top levels of a goal tree provide information about the tasks the corresponding system is capable of performing. The Ectrospace (described below) provides indexing mechanisms so that search engines can readily locate systems satisfying given goal profiles.

- To determine the overall purpose of a system and how the purpose is realized in the code: a system's goal tree describes its overall purpose; interpolated methods describe how the purpose is realized in the code.

- To 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.

- To determine where and how to attach new functionality to the existing system such that overall system structure is maintained or improved: To do this the goal tree is extending to describe new functionality. This is accomplished by interpolating into the goal tree one or more additional behavior traces describing the new feature. Note that new functionality is always described in terms of system goals and the methods for accomplishing them. Consequently, extending a goal tree requires merging a new behavior trace into it. Maintaining conceptual integrity means assuring that a large part of the trace overlaps existing nodes in the tree.

Validation: Important Ectropic Design validation 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].

Adaptability is the extent to which a system can be easily extended to accomplish new goals. As there are no generally accepted measures of adaptability, design critiquing tools are required that are capable of pointing out instances in which proposed new functionality is structurally similar to existing functionality, thereby suggesting opportunities to increase 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 element, 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 system behavior conforms to design constraints. Constraint nodes explicitly direct designer attention to those constraints.

B. Ectrospaces

1. Conceptual Foundations

The core of Open Source development is the communication mechanisms for collaboration among the participants. In current Open Source efforts, the communication mechanisms are e-mail, web pages, FTP sites, news groups, and similar Internet-based transport mechanisms. The common themes of these mechanisms are generality and simplicity. None of these mechanisms are designed explicitly for Open Source, and all of the mechanisms are passive conduits for information represented in a few simple media, such as text, graphics, and executable binaries.

Ectrospaces are intelligent, proactive collaboration spaces. Software development is performed in a distributed fashion by multiple developers accessing an intelligent and proactive, multi-representational, collaboration space. The collaboration space is intelligent in that its server is aware of the Ectrospace's contents and can manipulate them. The space is proactive in that it can take initiative to alert a developer to both relevant resources and conflicting constraints. An Ectrospace supports multiple representations and tools, all of which are active and manipulatable by the developer.

An Ectrospace differs in three ways from traditional transport mechanisms:

- The Ectrospace server is intelligent in that it is aware of the design present in the Ectrospace. It can display representations of it, maintain constraints on it, and track changes to it.
- The Ectrospace server is proactive in that it looks for opportunities to identify redundancy within the software developed by the users of the Ectrospace.
- The Ectrospace supports multiple representations and media in active forms. A user of an Ectrospace can manipulate computational elements which are active and executing on the user's machine. Active media in the user's space facilitates development, but requires the Ectrospace to rapidly distribute changes.

Ectrospaces build upon related areas of computer science research include computer supported collaborative work [Greenberg], [Fitzpatrick], external object systems such as CORBA, active servers, distributed and replicated databases, computational agents, and constraint management [Myers].

2. Approach

In our vision for Ectrospaces, our goals are:

- Users manipulate graphical and computational elements. These elements exist on the users' machines, for ease of access, but their changes are automatically propagated to 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 it currently serves. 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 new versions do not damage other users' code, but rather to inform developers that new versions exist. It knows how to make new elements available

widely. Elements that are used in different forms are replicated on the server. In addition, the server can maintain constraints that protect the conceptual integrity of the system.

- The server is proactive. It seeks out common elements from different developers and identifies these commonalities, 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 Ectrospace, so that conceptual integrity is preserved.

These basic capabilities of Ectrospaces are enough to support a wide variety of Open Source efforts. Combined with Ectropic Design, Ectrospaces enable highly-distributed Open Source projects where integration and persistent conceptual integrity are facilitated by the space.

V. Status

The constituent methods of Ectropic Design (goal trees and behavior traces) are mature technologies although they have never before been combined. We have developed and released several different tools for building and displaying goal trees and for simulating the execution of a behavior trace on a goal tree. We have also built and released a tool for analyzing behavior traces in order to detect and extract code segments responsible for implementing particular goals. The next step is to experiment with more closely integrating the two methods.

We are currently implementing Ectrospaces on Squeak, a new, free, cross-platform implementation of Smalltalk that strongly supports multimedia applications. We have a limited prototype of Ectrospaces working now. The current prototype allows sharing of graphical and computational elements, with turn-taking support for synchronous use. The Ectrospace server does know all of the objects in the Ectrospace, though it does not yet do anything with that knowledge. We have several constraint systems now running in Squeak, but they have not yet been integrated into the Ectrospace. We plan to do some field testing in a classroom setting of the current prototype in the next six months.

VI. References

[Belady] L. A. Belady and M. M. Lehman. *Program Evolution. Processes of Software Change.* Academic Press, 1985.

[Brooks] Frederick P. Brooks. *The Mythical Man-Month, Essays on Software Engineering*. Addison-Wesley Publishing Company, 1982.

[Dardenne] A. Dardenne, A. van Lamsweerde, and S. Fickas. "Goal-directed Requirements Acquistion." *Science of Computer Programming*, 20(1-2), April 1993.

[Dix] Alan Dix, Janet Finlay, Gregory Abowd and Russell Beale. *Human-Computer Interaction, 2nd Edition.* Prentice Hall, 1998.

[Fitzpatrick] Geraldine Fitzpatrick, Simon Kaplan, and Tim Mansfield. "Applying the Locales Framework to Understanding and Designing." *In Proceedings Australasian Computer Human Interaction Conference (OzCHI'98),* pp. 122-129, Adelaide Australia, IEEE, 1998.

[ITU] International Telecommunication Union. "Criteria for the Use and Applicability of Formal Description Techniques / Message Sequence Charts (MSC)." ITU-T Recommendation Z.120, March 1993.

[Greenberg] S. Greenberg. "Collaborative Interfaces for the Web." In *Human Factors and Web Development,* C. Forsythe, E. Grose and J. Ratner (eds.), Chapter 18, pp.241-254, LEA Press, 1997.

[Kazman] Rick Kazman, Len Bass, Gregory Abowd and Mike Webb. "SAAM: A Method for Analysing the Properties of Software Architectures." *Proceedings of the 16th International Conference on Software Engineering,* 1994.

[Korel] B. Korel and J. Laski. "Dynamic program slicing." *Information Processing Letters,* 29(3):155-163, October 1988.

[Murdock] J. W. Murdock and Ashok Goel. "A Functional Modeling Architecture for Reflective Agents." *AAAI-98 Workshop on Functional Modeling and Teleological Reasoning*, Madison, WI, July, 1998.

[Myers] B. A. Myers. "The Garnet compendium: Collected papers 1989-1990." Carnegie-Mellon University, 1998.

[Potts] Colin Potts, Kenji Takahashi, and Annie I. Anton. "Inquiry-Based Requirements Analysis." *IEEE Software,* 2(11):21-32, March 1994.

[Raymond] Eric Raymond. "Introduction to Open Source." http://www.opensource.org/ intro.html.

[Rugaber] Spencer Rugaber, Kurt Stirewalt and Linda Wills. "Detecting Interleaving." *International Conference on Software Maintenance,* 265-274, October 16-20, 1995.

[Zave] Pamela Zave and Michael Jackson. "Conjunction as Composition." *ACM Transactions on Software Engineering and Methodology,* 4(2):379-411. October 1993.