Technical Report #85-09

MAINTENANCE AND PORTING OF SOFTWARE BY DESIGN RECOVERY

Guillermo Arango
Ira Baxter
Peter Freeman
Christopher Pidgeon

Reusable Software Engineering Project
Department of Information and Computer Science
University of California at Irvine
Irvine, CA 92717

ABSTRACT

Enormous resources are invested in the construction of software. As needs change we would like to be able to preserve these investments. In this paper we outline a unified model for addressing the problem of change in software systems for which the original specifications are not available. Our approach is derived from a much broader paradigm for software construction. The approach is based on recovery of abstractions and design decisions made during implementation of the current version of a system. The paper consists of two major parts. We begin with a discussion of the Draco paradigm for software construction and how it can be applied to maintenance and porting problems. We follow with a discussion of our experience with a porting project using prototype implementations of these ideas.

Support for this research has been provided by National Science Foundation grant MCS-83-04439.

1. INTRODUCTION

Enormous resources are invested in the construction of software [Lyon81]. As needs change we would like to be able to preserve these investments. Changes may be caused by the desire of users to move the software to different environments, or by the need to alter the functionality or performance of software.

Why does changing software prove to be so difficult? For most software the original design is inaccessible: the original requirements analysis and specifications, if recorded, are out of date. The existing realization of the software systems usually contain implicit assumptions about their environments. Idioms corresponding to these environmental idiosyncracies are scattered throughout the code of the program.

Those parts of the design and environment that are recorded in documents cannot be automatically processed because they are not in machine-processable form. Manual processing is expensive and unreliable. There is also a popular perception that "small" changes in a system will require correspondingly small, incremental efforts. Because of our inability to estimate the cost of changes we optimistically assume that they can be easily done. When inappropriately "simple" changes are introduced, inconsistencies occur that then require extensive testing and further analysis to locate and debug. Changes made to a program in

the past leave "scar tissue": code not easily changed due the ripple effect on the rest of the software. Along its life time, a system is taken apart, twisted and sewn back together beyond recognition (*). Horror stories about these software Frankensteins are well known to the software practitioners.

We see three major reasons for change: enhanced performance is needed, the program must be made to operate in a different environment, or different functionality is required. Many authors [Hague76] [Tanenbaum78] [Poole73] treat these changes as requiring different solutions, as evidenced by their varying definitions of portability, transportability, adaptability, etc. We propose a model that unifies the management of change regardless of its cause. In this sense, we share Boehm's general definition of maintenance as "the process of modifying existing operational software while leaving its primary functions intact" [Boehm81].

Our capacity to make changes to software systems is limited if we must rely on manual systems. We are limited by the sum of the maintenance team's ability, and only "better design" can be proposed as a means to amplify its capacity. The alternative is to attack the fixed-change capacity assumption by use of tools not limited by those abilities.

(*) This sentence has been sewn back together beyond recognition.

We propose a method for the re-implementation of programs by recovering the design of such programs, and using the recovered design, to re-implement the program in new environments (porting), with different functionality (maintenance), or with different performance (performance enhancement). This method is based upon an emerging paradigm for the automatic construction of software for which a prototype, the Draco system, has been developed [Neighbors84].

The paper has two major parts. We begin with a discussion of the Draco paradigm for software construction and how it can be applied to maintenance and porting problems. We follow with a discussion of our experience with a porting project using prototype implementations of these ideas.

2. THE DRACO PARADIGM

The Draco paradigm for the generation of software assumes that an organization wishes to construct a number of similar software programs. These programs will share the property that they operate, in an abstract sense, on objects from one or more domains. Before program construction begins, the domain areas of interest are formalized by specifying, for each domain:

- an (informal) set of concepts composed of objects, operators and relations
- a formal external notation for specifying an instance of

the above: a domain language

- a recognizer for the notation (a "parser")
- a formal internal representation for the notation (an abstract graph constructed from the parsing process)
- a set of transformations, which map internal representations in a domain to equivalent internal representations in that same domain, generally used to effect optimizations
- a set of refinements, which map individual concepts in the domain to one (or usually more) concepts in another, domain.

The domains required to develop software for a given application area can be viewed as constituting a "domain structure graph" in which the nodes are domains and the set of refinements between them are represented as arcs. Such a network must provide for refinement paths to map high-level specifications into low-level implementation. Usually there are multiple paths through the domain network from an abstract domain node to an implementation domain node.

Software development starts with an abstract specification written using a combination of existing domain languages. The implementation process traverses a path through a space of possible implementations of progressively lower abstraction until a concrete implementation is reached

(Figure 1). The space forms a(n enormous) directed acyclic graph (DAG) called a "possible refinement DAG", with nodes in the graph representing specifications for the program written with notations from multiple domains. The single root of the DAG is represented by initial specification. Leaves of the DAG are concrete, executable specifications. Arcs represent individual possible design choices (refinements); the domains used by the specification at a node limit the type of arcs which exit that node to precisely those arcs emanating from the same domains found in the domain structure graph. Usually, an individual node is reachable by many paths, representing differing orders of choice of the same set of design decisions. A path from the root to a leaf represents a particular choice of a set of implementation design decisions, and constitutes what is generally called "the design". Navigation through the graph may be controlled by an implementation-style enforcing mechanism called "tactics". Separate tactics can co-exist for different purposes: implementation for speed, for minimal space, for rapid prototyping, etc.

The refinement DAG is never constructed in its entirety. Only the path needed to reach a desired leaf from the root is explored. Once an implementation design path is chosen, it is not kept as such, but the design decisions that define the path are generally retained. A prototype tool to handle domain specifications, and to construct implementation paths from abstract program specifications

has been constructed.

3. A MODEL OF MAINTENANCE USING THE DRACO PARADIGM

We believe that one can use the Draco paradigm to accomplish maintenance activities. In this section, we assume that a program has been derived from a specification using the Draco paradigm, and that the specification, the refinement DAG, and the implemented program are all available to a would-be maintainer. We will discuss the maintenance problem in the the absence of the specification and the refinement DAG in a later section.

Should a program need change, there are two methods for accomplishing it. One possibility is to choose an entirely new path through the refinement DAG from the initial specification to a different implementation. This method is generally not preferred, as many of the design decisions made for the current implementation can be reused in the desired implementation.

The other alternative is to start with the concrete implementation chosen, reverse some of the design decisions, moving up the refinement DAG towards the root, until a node is reached which is the least common abstraction (LCA) of the current implementation and the desired implementation. The least common abstraction is the top node of an embedded sub-DAG, and can be reached by any of several paths (as the design decisions need not be reversed in the order origi-

nally made). A new path must then be chosen from the LCA to the desired implementation (Figure 2). This method preserves all of the implementation design decisions made above the LCA and thus minimizes work required to accomplish change.

Performance enhancement is generally accomplished by changing the underlying representations used by a program, and using more efficient procedures made possible with the changed representation. We assume that the revised representations and corresponding procedures are already contained as refinements in the domains used to generate the current program (if this is not the case, then the domains must be augmented accordingly), but were simply not used. Some set of nodes in the refinement DAG are LCAs that allow re-implementation of the currently low-performance abstractions. Design decisions are reversed to travel from the current implementation back to one of those LCAs. New decisions are applied to arrive at a different implementation. The change in refinement direction is accomplished by a change in tactics.

Changes of-environment can be handled in a similar fashion. The domains are first augmented with new refinements specifying how the abstractions used in those domains can be implemented by the new environment; this effectively produces an implementation DAG which entirely contains the original DAG (Figure 3). A suitable LCA is found and re-refined using the revised refinements.

Different functionality is accomplished by changing the specification. It is then straightforward, but possibly inefficient, to re-refine the specification to a particular implementation. Refining the new specification creates a new refinement DAG, different than the original.

A perhaps more efficient method for producing the revised program requires several steps (Figure 4):

- determine a substitution S that converts the original specification to the revised specification (this can be constructed automatically as the original specification is revised).

- determine the largest subgraph G', of the new refinement DAG, starting from the top node, that is isomorphic with a subgraph G, of the old refinement DAG under the substitution S. Each node n in G has a corresponding node n' in G', obtainable by applying the substitution S to n. Note that G' must include at least the root node (i.e., the revised specification).

- find an LCA of P in G. The corresponding node in G' can be refined to a concrete implementation P' which realizes the revised specification.

To determine the isomorphism, and therefore the candidate LCAs, the refinement DAGs need not be constructed in their entirety. The work accomplished in the original refinement history up to the chosen LCA in G can be reused at great

savings. Refinements from the LCA in G' to the concrete implementation P' must be applied; this constitutes the bulk of the work. Design decisions used in the path from the LCA in G to P can perhaps be reapplied, reusing analysis done for the original program.

If the specification is modular, then there will be a refinement DAG for each part of the specification. The implementation will consist of a set of leaves, one taken from each DAG. A change to the specification will then then affect only some of the specification modules, and so affect only some of the refinement DAGs. Leaves nodes from DAGs which do not change may be used in the new implementation unchanged. The procedure outlined above can be used to generate new leaves for the changed DAGs. Modularity is then seen simply as a method for making trivial the determination of the isomorphism on portions (the unchanged DAGs) of the what would otherwise be a single, large refinement DAG.

4. THE PROCESS OF DESIGN RECOVERY

In Figure 5 we present a view of the "conventional" approach to maintenance. Arcs are represented by broken lines to indicate that the refinement history, and thus the original abstract specification are not available. What is to guide the maintainer when going from program P to P'?

The Draco paradigm offers a model of maintenance activities provided that the program specification and

design are available. If we do not have these, we can recover them from the code, and then use the Draco paradigm as the guide. The design recovery paradigm we propose provides a systematic way of carrying out the process that we think maintenance programmers apply informally: before performing changes in a program to adapt them to new requirements, a higher-level plausible "ancestor" specification equivalent to the original program is informally developed.

Such an ancestral specification can be developed by repeatedly performing a "design recovery step". Each step consists of inspecting the specification recovered from the previous step, proposing a set of possible abstractions of the portion of interest, choosing the "most suitable" abstraction, and constructing a specification containing the new abstraction. Each abstraction proposed implicitly selects some domains and refinements which must produce the existing code when applied to the ancestor containing the proposed abstraction. Design recovery steps are repeated until a useful LCA is reached.

The design recovery process is illustrated in Figure 6. Starting with program P its plausible immediate ancestors (broken-circles) are postulated. Selection of an appropriate ancestor (solid circle) is based upon conjecture that the node is on the path from P to a suitable LCA.

Good choices of abstraction will use domains and refinements recovered in earlier steps, or will augment them

minimally. The iterative process induces learning in the maintainer which can be captured in the resulting domains. The choice of the appropriate ancestor is the result of a generalization process based on the specification under consideration. The implementation provides a very limited sample on which to base a generalization step. This has also been recognized by Boyle et al. [Boyle84] "Transformations that codify implementation decisions, ... , are frequently irreversible, ... (furthermore) ... it is impossible to tell from the program alone." In other words, unrefinements are possible only by using additional knowledge: we must rely on the maintainer's knowledge of the application domain, intelligence, experience and educated guesses, on common knowledge and on any additional information available on the current implementation (e.g., inputs from original designer, existing documentation, environmental specifications, etc.).

Since quite often the maintainers are not the original authors, and are usually distant in time from the original implementation, maintainers are likely only to regenerate APPROXIMATIONS of the original domains that were used. This mismatch between the maintenance DAG obtained by design recovery and an "ideal" DAG (Figure 7) reveals the crux of the maintenance problem. Avoiding approximations is very hard, and the approximation errors are typically amplified by repeated maintenance steps. The magnitude of the errors is increased when the recovery process is done informally.

The errors, generated by the limited sample used for the abstraction step, can be substantially reduced by performing domain analysis.

Through domain analysis a more adequate, complete and reusable set of abstractions of a knowledge domain can be produced, thus enhancing the power of the design recovery paradigm. Domain analysis is a the fundamental components of the Draco technology and one of the central research concerns of our Project.

5. ADVANTAGES AND LIMITATIONS OF A DESIGN RECOVERY APPROACH

A discussion of the advantages and limitations of the proposed approach should be based on some considerations on economy of scale and reusability of software engineering workproducts.

It could be argued that a systematic application of the design recovery approach to small programs is like killing gnats with a sledge-hammer. Recovering the design of a large application using the method outlined will require that the source for the application program be read and processed. Since these programs are written in conventional computer languages, capture of this information may require an effort comparable to writing the semantic analysis of a compiler. While this would seem to limit the utility of the process to only very big programs, such programs do exist and the payoff is large. In large organizations, the

recovery of the design of one program will probably lead to the discovery of many domains and domain relationships that can be used to recover the the designs of others, thus amortizing the recovery costs.

A key notion supporting the economy of scale is that of reusability of analysis and design information [Freeman83]. In the framework of the Draco technology, the analysis and design knowledge is formalized through networks of domain specific languages. These languages enable software developers and maintainers to reuse the expensive processes of analysis and design and to avoid a costly learning experience. Once the (archeological) analysis is performed on an application, new modifications and ports are easier, and resource requirements to accomplish these actions are more predictable. Given that the system is formal, we can make explicit predictions about the effects of certain kinds of changes. Consider, for example, attempting to reimplement a system using linked lists structures instead of arrays. For most current programs, we cannot even predict whether it is possible to re-implement such programs this way. With the design recovery paradigm, all we need do is redefine the implementation of data accesses to use lists, and re-refine to ground.

We believe that we can still benefit from the the economic and intellectual advantages derived from the reuse of the analysis and design processes, even if we have to

start by recovering them from a concrete implementation. Our conjecturee coincides with that of Boyle et al. [Boyle84]: "The fact that concrete programs have a plethora of such POSSIBLY, but not PROVABLY irrelevant properties makes them difficult to modify, extend, adapt, and transport. ... abstract programs contain ... only such information as is necessary to show that they solve the problem for which they were written. Therefore, modifying, extending, adapting and transporting is much easier that it is for concrete programs."

The model proposed has the attractive property of providing a unified solution to a variety of problems: the need to implement changes in the functionality or performance of a system or its interfaces with the environment. Once the infrastructural domain network has been defined for an application area, the updates to programs in that area are easy to accomplish and re-deliver to all customers, including the original. This approach allows for an economic and practical configuration control and distribution system for software applications.

6. RELATION TO OTHER WORK

The power of this approach is based in the formal manipulation of problem domain knowledge and software design knowledge. Hague in [Hague76] proposes the idea of a "super" language customized for the application domain which would map down onto a real language. He claims that gain in

flexibility may be lost because that language may not compile on a real machine, and thus rejects the idea. The Draco paradigm shows how to perform the mapping, and thus makes the idea appropriate to re-consider.

[Boyle84], following an approach similar to that of Hague, employs an extended version of FORTRAN as a single "base" language. By extending FORTRAN, new classes of abstractions can be expressed. In our paradigm these would qualify for separate notations. We will not dwell here on the limitations implicit in wide spectrum languages, but we must acknowledge the fact that the use of a single base language limits the transformation (refinement) alternatives. In any case FORTRAN is a poor base for a wide spectrum approach.

7. THE EXPERIENCE

The Reusable Software Engineering Project at UCI is conducting research on the reuse of Software Engineering artifacts and knowledge. One of our principal research tools is the Draco prototype system outlined here. This tool was coded in UCI Lisp [Meehan79] running on DEC 2020s.

The need to port this system arose as a result of the Computer Science Department's decision to migrate from Decsystem-20 computers running TOPS-20, to DEC VAXen under Unix (Berkeley 4.2). We needed to change the environment of our program. As UCI Lisp is not available on this new confi-

guration, we were faced with the issue of porting Draco or UCI Lisp.

At the time, spring '84, DEC was rumored to be ready to announce general availability of Common Lisp [Steele84]. As one of our industry sponsors desired a VAX-VMS-Common Lisp version of Draco, we considered Common Lisp as a potential target Lisp in addition to Franz Lisp available for Unix. We quickly dropped any idea of porting UCI Lisp, leaving us with the problem of porting Draco. We chose Franz because it was available, stable, and wide-spread in the research world. It appeared that Franz was a reasonable stepping stone on the way to a Common Lisp version of Draco. The implications were that we were to produce more than one new implementation of Draco, making the idea of hand-conversion particularly repugnant.

The version of Draco to be moved has a complex kernel coded in Lisp. The balance of Draco consists primarily of some specialized domains (not coded in Lisp!) used to construct domains; it was obvious (to us, anyway) that these could be easily moved using the Draco paradigm, so we will not further discuss converting the balance of Draco.

Being a research prototype, Draco was constructed single-handedly, and the author has since moved on and was generally unavailable. The other members of the research team were inexperienced with the mechanims used by Draco. We had the makings of a classic maintenance problem.

It was decided to apply the Draco paradigm to accomplish the port (this was before the ideas on design recovery had become more clear). To minimize the impact of new code on the porting process, we imposed the following iron-clad rule: there would be NO changes to the UCI Lisp source for Draco. The side effect is that we were forced to treat the kernal as a specification.

Our first discovery (and in retrospect, very obvious) was that Draco used only parts of the UCI Lisp dialect. This enabled us to design a limited domain which was specific to the Draco functionality. Thus we could capture the meaning of Draco-specific UCI Lisp idioms effectively in an abstract form and discard the concrete syntax. The recovery of the meaning from the concrete syntax is an example of reversing the design decisions to implement those abstractions with the particular UCI Lisp incantation. The captured abstraction corresponds to the LCA described in previous sections.

The Lisp idioms captured fell into 3 classes:

- classic LISP functions and S-Expresssions
- UCI-Lisp idioms (generally related to enviromental interface, such as I/O)
- Draco-specific abstractions implemented as procedures (Initialize, etc.)

To re-implement the Draco kernal in Franz Lisp, we coded new refinements for the abstractions captured in the previous step (this corresponds to moving down from the LCA to a new implementation in Figure 3). A typical example is shown in Figure 8.

The abstractions for the Input/Output used by Draco in UCI-Lisp turned out to be very difficult to refine directly to Franz Lisp. We concluded that the semantic gap was too large, and were forced to defined a "bridging" domain to implement these abstractions by a virtual machine technique, with some consequent inefficiencies (not expected when using the general Draco paradigm) in the final implementation. We expect this problem to re-appear when we retarget for Common Lisp. In retrospect, we feel that we did not capture these abstractions at a high enough level. The implications are that one should should capture the design at as an abstract level as possible to make re-implementation easier.

8. SOME QUANTITATIVE RESULTS

We include here some statistics about the conversion effort. The converted code consists of approximately 2400 lines of UCI Lisp code divided among some 170 functions. Approximately 280 abstractions were identified in four domains; refinements were implemented for each. About 45% of the abstractions refined directly (most of these were generic Lisp); 14% of the abstractions were implemented by simulation in the target environment. The balance of the

abstractions were not complex. The entire 2400 lines of the Draco kernal were automatically converted by the process, using 19 hours of CPU. Code expansion is estimated to be 10%. About 8 man-months were expended. We expect that much of the human effort expended can be re-used if we decide to proceed with a Common Lisp implementation.

9. CONCLUSION

We have outlined a model of the maintenance process based on the construction of software by components philosophy. This process requires the program specifications at some abstract level and the set of design decisions that were made to implement the program. We believe that the model can be formalized, and that the formalized version can be used, with human aid, to recover the design of concrete code, and can then be applied to make changes to the resulting design.

REFERENCES

[Boehm81] B. W. Boehm, Software Engineering Economics, pp. 54-55, Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1981.

[Boyle84] J. M. Boyle and M. N. Muralidharan, Program Reusable through Program Transformation, Transactions on Software Engineering, Vol. SE-10(5), pp. 574-588, Sep. 1984.

[Freeman83] P. Freeman, Reusable Software Engineering: Concepts and Research Directions, Proc. ITT Workshop on Reusability in Programming, pp. 2-16, Stanford, CT., 1983.

[Hague76] S. Hague and B. Ford, Portability -- Prediction and Correction, Software Practice and Experience, Vol. 6(1), pp. 61-69, 1976.

[Lyon81] M. J. Lyon, Salvaging your Software Asset (tools based nmaintenance), AFIPS Conference Proc., Vol. 50, pp. 337-341, 1981.

[Meehan79] J. R. Meehan, The New UCI Lisp Manual, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1979.

[Neighbors84] J. M. Neighbors, The Draco Approach to Constructing Software from Reusable Components Transactions on Software Engineering, Vol. SE-10(5), pp. 564-573, Sep. 1984.

[Poole73] P. C. Poole and W. M. Waite, Portability and Adapability, in Advanced Course on Software Engineering, F. L. Bauer Ed., Springer-Verlag, Berlin, 1973.

[Steele84] G. L. Steele, Jr., Common Lisp, Digital Press, Burlington, Mass., 1984.

[Tanenbaum78]

A. S. Tanenbaum, P. Kling and W. Bohm, Guidelines for Software Portability, Software-Practice and Experience, Vol. 8(6), pp. 681-698, 1978.

[Wilensky84]
R. Wilensky, LISPcraft, W.W. Norton and Co., New York, N.Y., 1984.

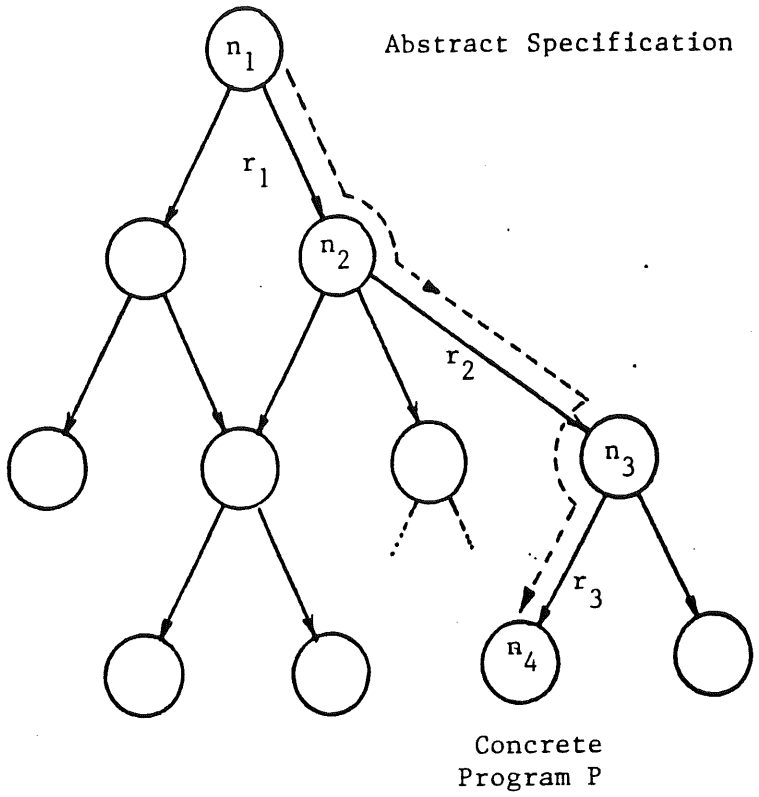


Figure 1. Construction of program from specification
Refinement decisions r_1, r_2 and r_3

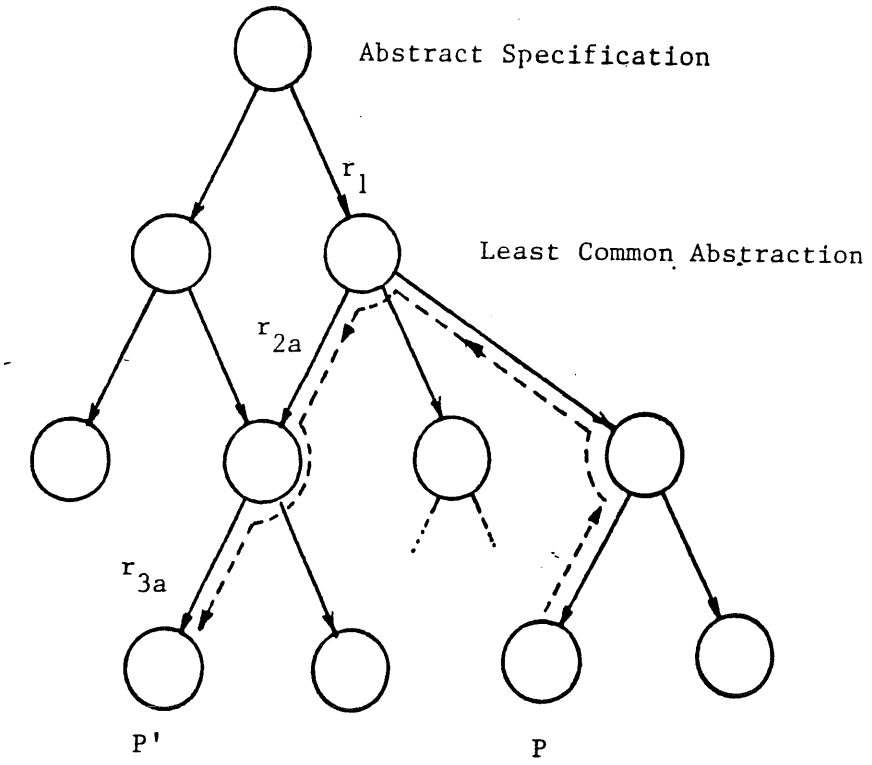


Figure 2. Maintenance
General choice r_1 is preserved

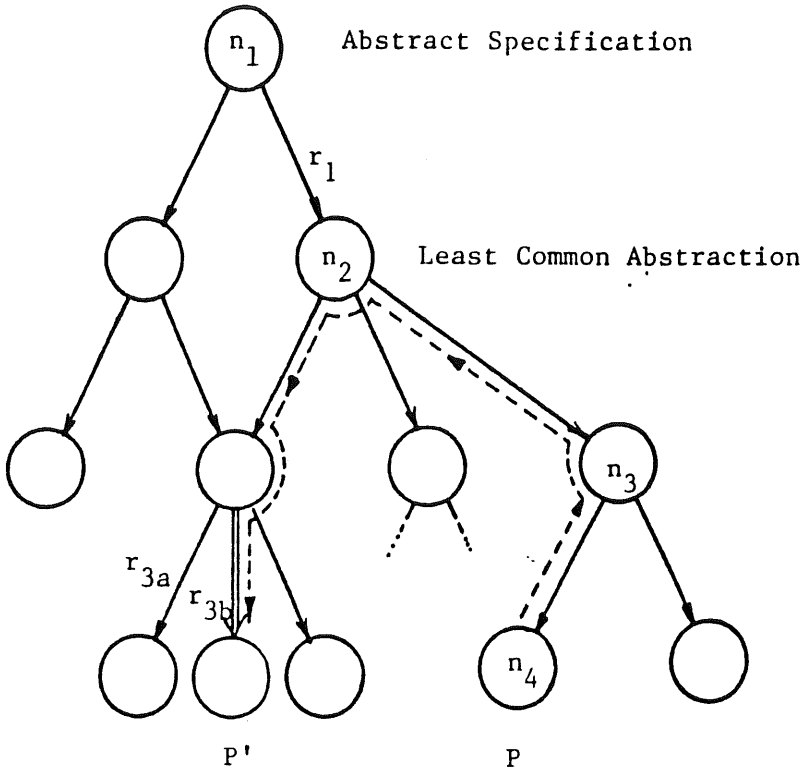


Figure 3. Changing Environment:
r_{3b} is a new refinement

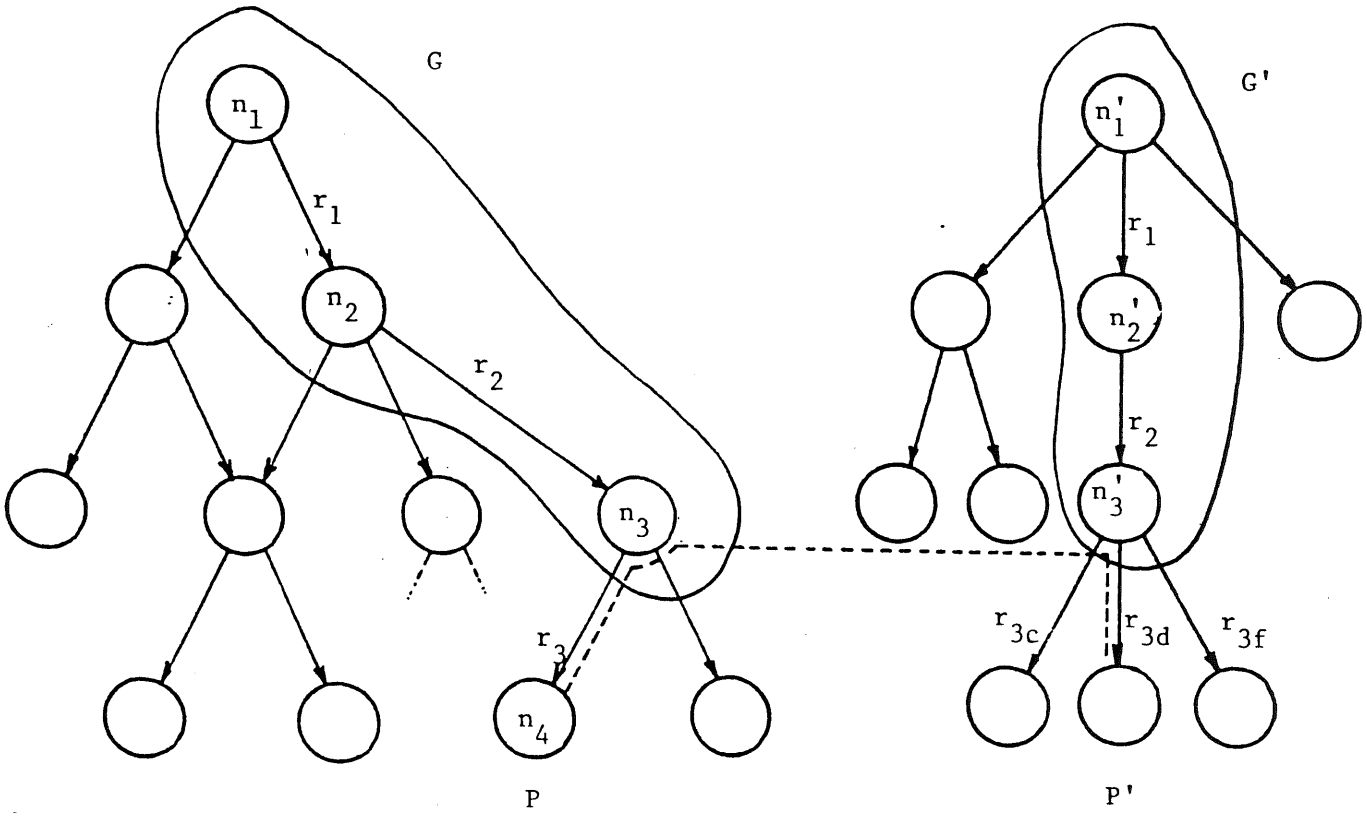


Figure 4. Changing Specification:
G' is isomorphic to G under substitution S

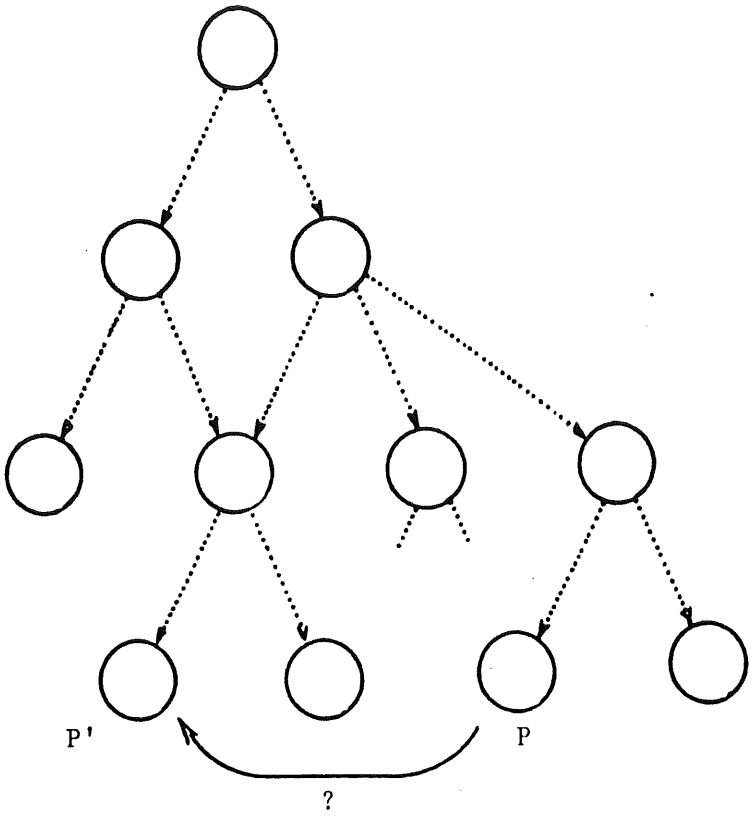


Figure 5. Conventional Maintenance:
With no background, What guides the changes?

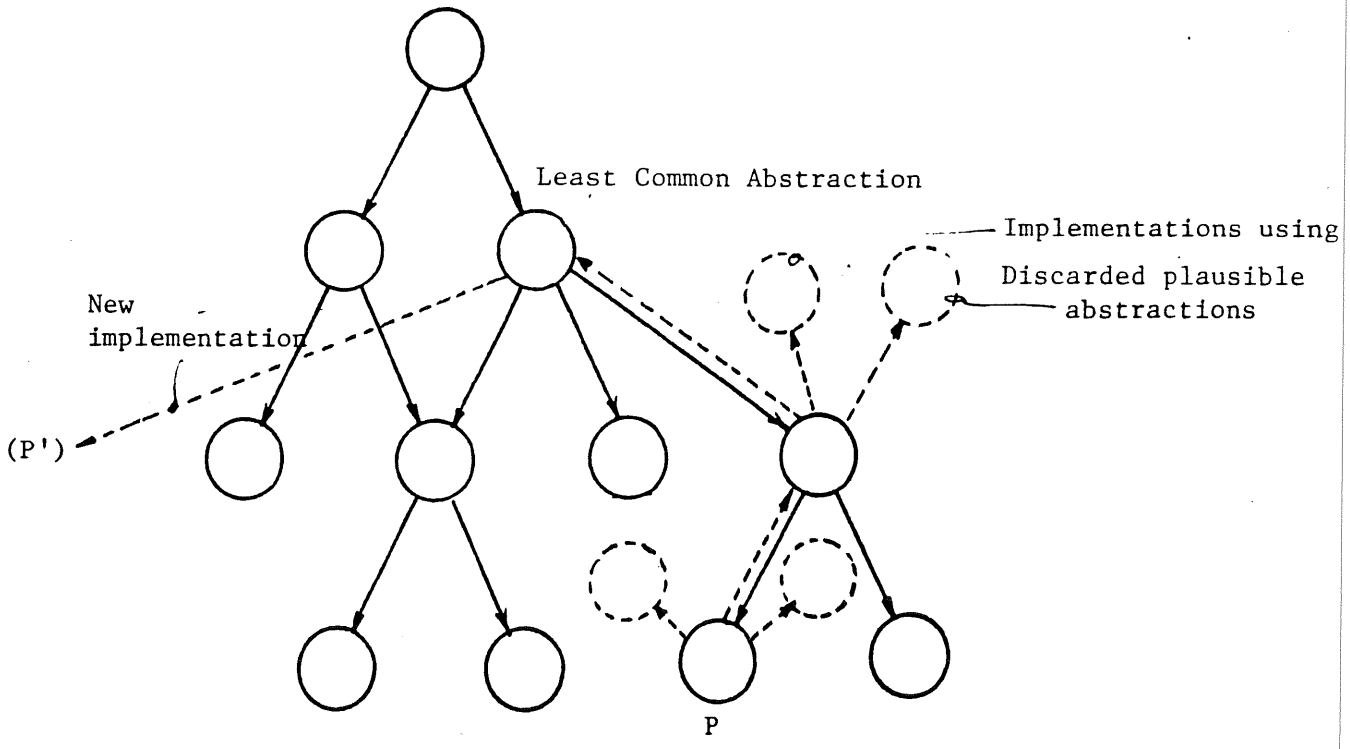


Figure 6. The Process of Design Recovery

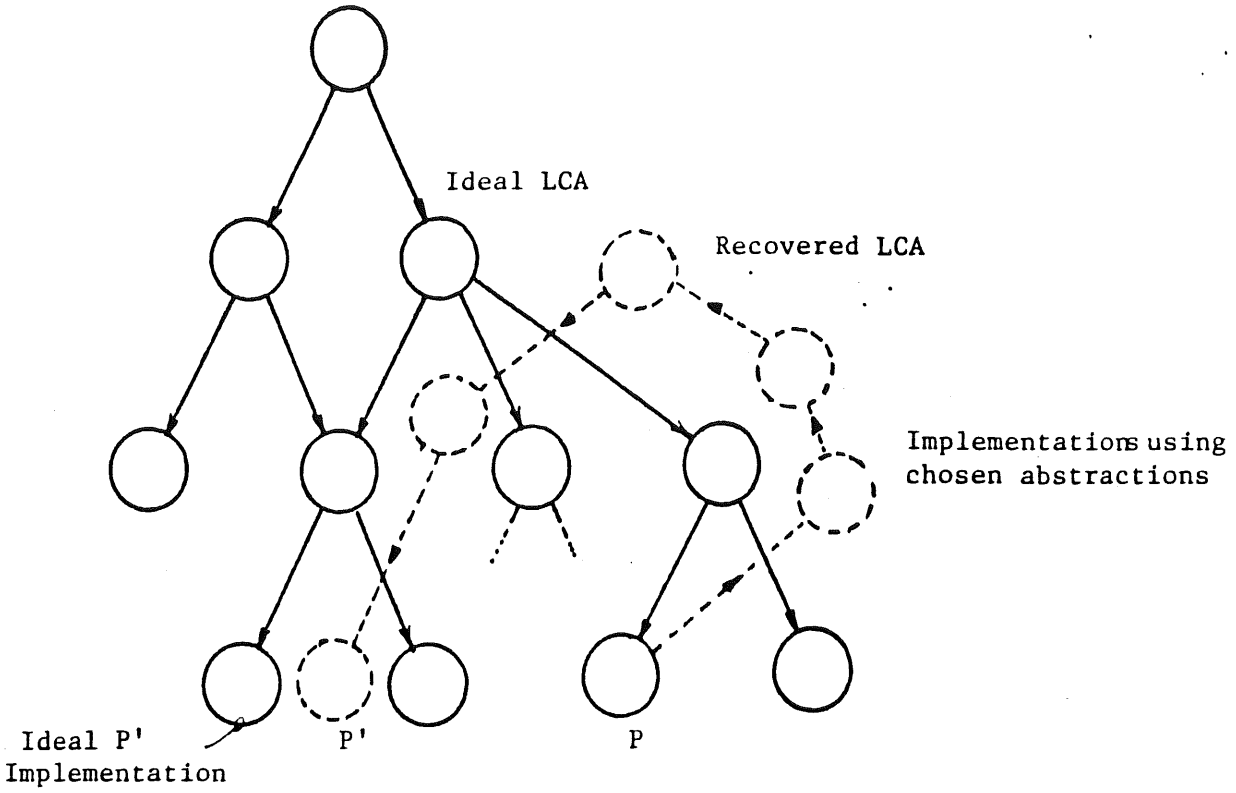
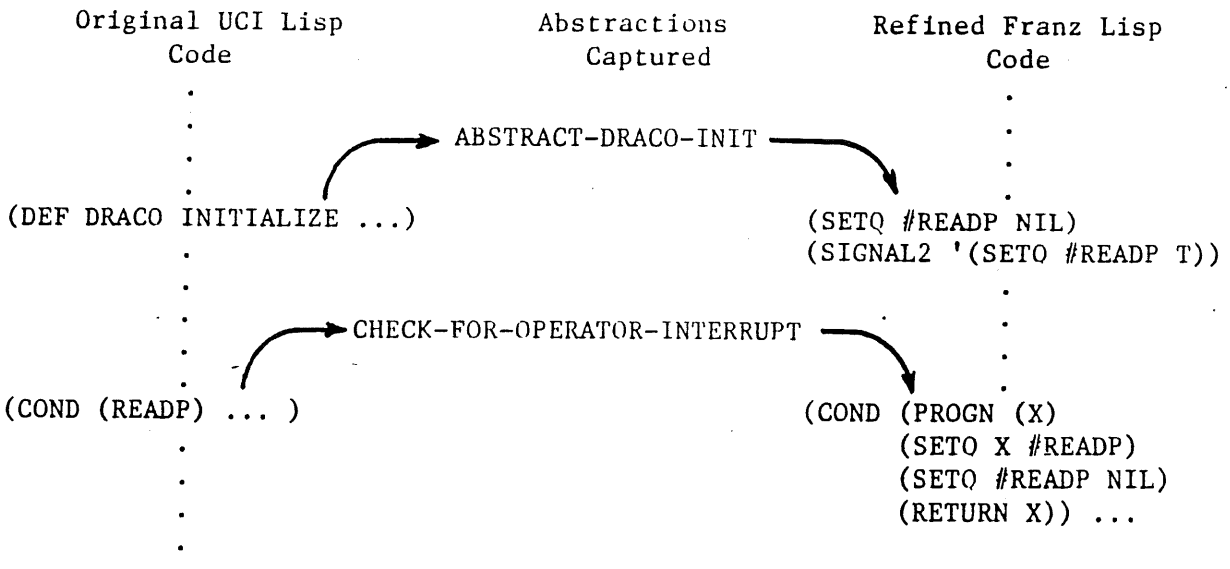


Figure 7. Recovered design vs. "ideal" design: the approximation error.



DRACO-INITIALIZE was concrete code to initialize DRACO.

READP is a predicate which determines if any keys have been struck by operator.

SIGNAL2 causes its argument to be executed when CTRL-C is typed.

Figure 8. Example of an Abstraction