Software Change Through Design Maintenance

Ira D. Baxter
idbaxter@semdesigns.com

Semantic Designs, Inc.
12636 Research Blvd., Suite C-214
Austin, TX 78759 USA
(512) 250-1018
http://www.semdesigns.com

Christopher W. Pidgeon
cwpidgeon@semdesigns.com

ABSTRACT

Conventional software engineering tends to focus on a small part of the software life cycle: the design and implementation of a product. The bulk of the lifetime cost is in the maintenance phase, where one must live with the product previously developed. Presently, we have little theory and fewer tools to help us manage the maintenance activity. We contend that a fundamental cause of the difficulty is the failure to preserve design information. This results from an over preoccupation with the synthesis and maintenance of code. We offer an alternative paradigm:

- make the design the central focus of the *construction* process—get code as a byproduct;
- make the design the central focus of the *maintenance* process—preserve revised designs and get code as a byproduct.

A transformational scheme for accomplishing this is presented. We call it the *Design Maintenance System*. The programming roles change radically from coding instances for ill-defined specifications to specifiers of functionality and (compiler-like) implementation methods. Specification and implementation method debugging would then become prominent activities. The design scheme and change management procedures are illustrated with a simple data processing application. We sketch an ongoing implementation.

Keywords: Design, Maintenance, Transformation System, Domain Engineering, Automation

INTRODUCTION

The average lifetime of software is about 10 years, according to a study [TAM92] of 95 systems drawn from a variety of application domains. Consequently, is hardly surprising that most of the lifecycle costs for software occur during the so-called *maintenance phase* [GRA87], since its duration dwarfs the development phase.

Incomplete or nonexistent system documentation was ranked in the top four problems in software maintenance according to a Delphi study run with senior software maintenance managers [DEK92]. (The other top-ranked technical problem was *inadequate testing*). This is consistent with the perception that maintainers face two major obstacles: understanding the program to be modified, and validating the modification while assuring that the remainder of the program is not accidentally affected.

It seems clear that better processes for producing and maintaining system documentation (as well as testing) for generated programs would reduce maintenance costs. However, this approach relegates documentation to secondary status. As such, it is likely to get short shrift. *We recommend that the software development process should treat the design as the major product*, with the implementation (code) being merely a useful byproduct. Implementation decisions and their rationale are captured as they are made, not after the fact.

There is considerable value in capturing designs, decisions and their rationales even informally. However, informal designs are subject to wide interpretation. This variability limits their value. Moreover, the burden of details to be managed makes this so unappetizing that informal capture just is not performed. Thus, we are confronted by the present state of affairs: design information simply is not preserved.

So we turn our attention to formal development methodologies. This paper will briefly outline a vision for a *Design Maintenance System* (DMS) [BAX92] that transformationally constructs and records the design of software. We then sketch how the design may be incrementally modified by the DMS to produce revised versions of the software. We must necessarily be brief; details of the procedures are found in Baxter's dissertation [BAX90].

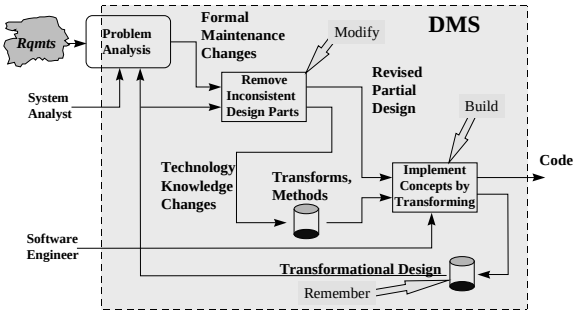

Figure 1. Design Maintenance System Concept

Our intention is to exhibit a system that maintains designs, in an attempt to persuade the software engineering community of the value of our point of view. Commercial tools may not yet be available, but paradigm shifts must occur before such tools can be used effectively.

WHAT'S IN A DESIGN?

If we are to capture a design, we must know what it is that we are capturing. Most design notations can be considered as projections of the completed artifact, under which some chosen aspect of the artifact is displayed. Examples include, call graphs (structure charts), data flow diagrams, state machines, interface specifications, etc. The design process then consists of choosing sets of projections for which the designers believe that a final artifact can be constructed, and acquiring construction hints from those projections. Questions about the artifact are answered by inspecting the projections.

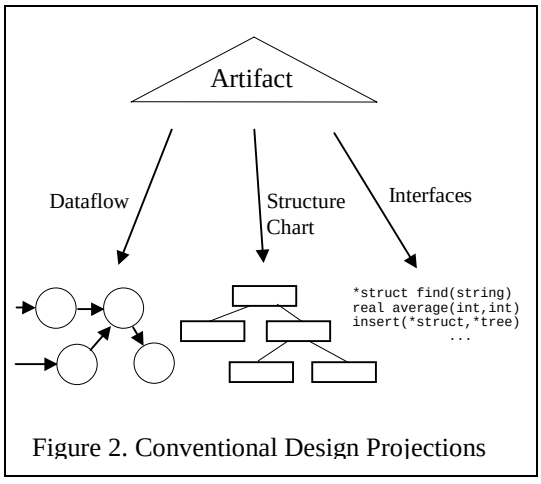

Figure 2. Conventional Design Projections

The flaw with this notion of design is the *absence of rationale*; projections do not explain *why* the artifact organized the way that it is. Without such rationale, one can hardly hope to explain the artifact. Worse yet, how do we decide what part of an artifact is worth preserving in the face of some proposed change?

One way to capture a rationale is to understand how the artifact was constructed, and why the construction works. We turn to transformational implementation methods to provide this information. Such methods can allow us to capture the design rationale as:

- A specification of the desired task, both functionality and performance;
- A derivation of the implementation from the specification, explaining the final program; and
- A justification of the derivation steps.

TRANSFORMATION SYSTEMS

Transformation systems convert abstract program specifications into concrete programs by applying *semantics-preserving* transforms to produce new specifications [FEA86, PAR90]. Each system usually has a large repertoire of available transforms, and can choose which ones to use semi-automatically. Compilers are simply transformation systems with fixed specification languages, predefined transform libraries and fully automatic choice of transformations.

In general, transforms may be arbitrary procedures (such as YACC, which transforms BNF specifications into LALR parser code). That is, transforms are functions from specifications to specifications, *t: S→S*. In this paper we will show only examples in the subclass that can be represented as *string rewrites*, although our ideas apply to the general case. Many transforms are familiar to readers as optimizations, such as the *eliminate-additive-identity* transform:

$$x + \mathbf{0} \rightarrow x$$

The italicized names are parameters of the transform, and are consistently substituted where the transform is used. When a *transform* is applied at a particular location in a specification, we obtain a *transformation* of the specification; the place where the transform (rewrite) has been applied is called the *locator*. The value of the locator depends on the underlying representation of the specification, e.g., a path for a tree. We will use

@line-number:token number

for paragraph shaped specifications. So the transformation

eliminate-additive-identity@3:1

applied to the 3rd line, 1st token of the (nonsensical) specification

1	**do j = 1 to 10**	1	**do j = 1 to 10**
2	**s=s+0**	2	**s=s+0**
3	**p=p+0**	3	**p=p**
4	**end**	4	**end**

changes that specification by binding *x* to **p** and rewriting **p+0** as **p**.

Some transforms change levels of abstraction by implementing high-level concepts in terms of lower level concepts. The *implement-sum* transform does this by

mapping the logical concept of a SUM over an array into the loop that implements it:

```
1   sum(var,limit,vector) →
2   begin    local s = 0, var
3        do var = 1 to limit
4             s = s + vector(var)
5        enddo
6        return s
7   end
```

A full specification has two conceptual parts: a *functional* specification (what the desired program should do), and a *performance* specification (how well it should do it). Functional specifications may be written as abstract programs, as input-output constraints, or in problem-domain specific notations. Performance specifications are often stated in terms of desired target languages (PROLOG or BASIC), speed (.5 sec response time per transaction), complexity ($O(N^2)$). *Transformations are applied to the functional part only to achieve some specified level of performance* [BAX90].

A program specification may be very abstract or describe a very complex system. In this case, a large number of transformations may need to be applied to implement the specification at the desired level of performance. Since manually commanding a transformation system to apply large numbers of transformations is impractical, many transformation systems allow *metaprograms* to control the selection and application of the transforms. The primitive operations in such languages apply transformations to locations. If a metaprogram cannot decide locally what to do, then it may backtrack to try alternatives. Such metaprograms may be purely procedural [WIL83] or may be planning-style languages [ALL90] with post conditions describing effects in terms of performance specifications [MCC87, BAX90]. The post condition style allows *methods*, metaprogram components, to be selected (**achieve**d) non-procedurally according to performance specifications. Such methods also allow explicit statement of required sequencing of transformation steps. Consider for example

name: *lower-abstraction-level*
post: **abstractionlevel**(*spec*)=low
action: foreach sum**@***locator* **in** *spec*
do apply *implement-sum***@***locator*

The **foreach** operator implies that there is no particular order to multiple applications of *implement-sum*.

The following method invokes *lower-abstraction-level* non-procedurally.

name: *implement-in-BASIC*
post: **targetlanguage**(*spec*)=BASIC
action: achieve abstractionlevel(*spec*)=low
then call
implement-using-BASIC (*spec)*

The **then** operator requires that transformations that lower the abstraction level be performed before any transformations that implement any operations in BASIC.

Complex performance specifications must be achieved by breaking them down into simpler performance specifications over smaller regions (*locales*) of the program.

CAPTURING A TRANFORMATIONAL DESIGN

A design should provide a rationale for the construction of an artifact. Clearly, any rationale must include a specification of the artifact. (Many present software designs do not include any usable specification.) Theoretically, a specification is sufficient. We should be able to work forward from the specification to rediscover the purpose of each part of the artifact. In practice, we wish to cache the connection between the specification and the artifact, because we do not want to effectively redesign it each time we need an explanation. We presume that either the programmer or the transformation system worked hard:

- to discover which transforms to apply,
- to determine exactly where to apply them,
- to achieve the desired level of performance.

The explanation of a transformationally derived program is straightforward: the *derivation history*, the sequence of transformations applied to the functional specification, explains the final program. The choice of the individual transformations is explained by the beneficial effect the individual transformation has on achieving the performance specification. If we record how the overall performance specification is broken into subspecifications over smaller locales, we obtain a *design history*. The design history includes a *derivation history*—the complete explanation of how the performance and functional specifications are met.

Figure 4 shows an abstract design history. The initial functional specification, $f_{0,}$ was transformed by application of transformations c_1, c_2, etc. until the final implementation, f_G was obtained. The performance specification, G_{rest}, was recursively partitioned by choosing methods (boxes) that achieve portions of the individual performance levels; the sequencing constraints one the order of method execution and transformation application is also recorded. Eventually, low-level methods apply particular transformations.

DESIGN MODIFICATION

Such a design history provides a complete explanation of the final artifact, f_G. *It is possible to incrementally revise the design history, driven by the desired change, to produce a new design history.* Since the new design history contains a derivation, the revised artifact is easily extracted from the new design history.

Figure 4 also shows, in gray boxes, the various kinds of changes Δ_{Type} that can affect the final artifact. *In each case, the desired change is made explicit in the form of a delta*. Some changes affect the functionality of the product. Some changes affect the performance but not the functionality. The DMS model also handles changes to the software engineering infrastructure caused by domain engineering errors, such as errors in the library of transforms or methods, or even changes in the definitions of performance. A separate change procedure for each type of change is required, but all these procedures share considerable commonality due to their need to revise the design history.

Each change can cause complex ripples in the structure of the design history. We do not have space here to illustrate most of the procedures, nor even space to illustrate how the performance goal decomposition is updated. We will focus instead on showing how a functionality delta can be used to revise just the derivation history portion.

The key to revising the derivation history is to take advantage of the ability to commute the order in which the transformations were originally applied. In essence, we wish to preserve transformations when possible, and remove transformations that are no longer useful, inspecting the interaction between the functional delta and any proposed transformation to guide our decision making process. We start the process with a functionality delta applicable to the initial specification.

For each intermediate specification (including the initial spec), we have a transformation leading to the next intermediate specification, and a delta describing the change required. To determine if a transformation *t* can be preserved in the face of a delta, we determine if

$$\Delta(t(spec))=t(\Delta'(spec))$$

This essentially tells us that the implementation step accomplished by the transformation is not affected by the change we wish to make. If there is no effect, the transformation can be preserved and is copied to a new derivation history. The delta might change in form (but not intent) because the implementation transform may rearrange the specification somewhat, and consequently the locator for the delta can change. If the implementation transform lowers the abstraction level, the delta may also shift levels, to express the change at the lower level of abstraction.

If the transformation and the delta interfere or, by conservative assumption if we are unable to decide, then we *banish* the transformation. Banishment is accomplished by commuting, if possible, the offending transformation with its immediate follower in the derivation history. If the offending transformation will not commute with its follower, then the follower must be dependent on the offender, and is banished by the same method.

DMS walks down the derivation history, deciding whether it must preserve or banish the implementation transformation at each intermediate specification. When a transform is reached this cannot be preserved, and cannot be banished because the rest of the transforms depend on it, then no more transforms can be preserved, and DMS stops the walk. The already-preserved transformations form a legitimate prefix of a complete derivation for the revised specification. DMS then switches over to a more conventional transformation implementation style to complete the new derivation, requiring possibly more "design" to choose new transformations. Of course, the DMS can propose trying to use the remaining transformations by analogy, or it may simply discard them.

The completed derivation history has the desired revised program at the end, and can be revised again by the same process for yet another delta.

AN EXAMPLE

The previous section sketched the abstract derivation history revision scheme. In this section, we sketch a concrete software maintenance scenario, and show how the DMS theory guides the derivation history revision process. Because of space constraints, we must necessarily demonstrate a formal mechanism rather informally, and we have consequently taken liberties with the details and the notation. Since our purpose is to change the reader's perspective about how maintenance might be done, rather than give him the exact mechanisms, we hope we are forgiven.

We start with an initial program that accumulates the total price of a set of order records kept in a file, where each order record contains an item quantity and a price. To keep the problem small, we omit other typical record attributes such as item name.

In Figure 5, we see the abstract functional specification for the original problem in the top left-hand box (circles in Figure 4 appear as boxes in Figure 5). To keep the example simple, we leave out performance specifications although the reader should see how they implicitly drive the implementation. Down the left-hand column of boxes

in the figure, we see a transformational derivation from the abstract specification into a practical program in a BASIC-like language. Each box represents an intermediate functional specification derived from the one above it; each intermediate step has exactly the same functionality as the one preceding it. No conventional programmer would implement the specification by going through these individual steps. Yet, they are necessary for a machine if one needs to be able to explain exactly how the resulting program implements the specification. Each downward arrow connecting boxes represents the application of a single transformation. Since we do not have room to write the exact formal transforms, we have settled for simply naming (t1, t2, ...). We show in italics, the nature of the transformation corresponding to the arrow immediately above it at the top each box. The individual transformations are justified by the performance enhancement each makes. The box at the lower left is the efficient final program that would be used in practice, and is what the traditional maintenance programmer would see as his starting point.

Now, a new need arises: our manager wishes to keep order quantities in separate files from the price per item. This is reflected by the revised abstract functional specification in the upper right box. One way to handle such a change request is to simply re-implement the program. Starting from the revised functional specification proceeding through the derivation history down the right column of boxes, the figure can be viewed as exactly that. This culminates in the changed program at the lower right.

However, we assumed that the discovery of transformations used in the original implementation was hard; we do not expect the discovery to be any easier in a new implementation. DMS shows how and when transformations used in a prior implementation can be reused in the new implementation, avoiding the discovery costs.

The arrows in Figure 5 crossing from left to right show how formal deltas tie the original and new derivation histories together (Figure 3). Our manager provides the delta **Δ0: order→price@1:17**, i.e., change the **order** in line 1, token 17 of the specification to **price**. The DMS determines that *t1* can be reused as *t1'*, because *t1* does not affect anything related to the **order@1:17.** However,

Δ0	**order→price@1:17**
Δ1	**order→price@3:12**
Δ2	**order→price@6:7**
Δ3	**order→price@6:11,7:7**
Δ4	**order→price@5:11,6:7**

Figure 3: Functional Δs for Figure 5

the delta must change to reflect the "movement" of the code caused by implementing the loop, giving Δ1. Similarly, *t2* and *t3* can be reused, changing the locators on the delta, giving Δ2 and Δ3 respectively. Transform *t3'* has not really changed; our liberties have caused us to write t3' with the variable part bound to the entity *@6:7*. (**price**)

Now, Δ3 conflicts with *t5*, and since the purpose of *t4* is to enable *t5* (information captured in the design history but not shown here), then both *t4* and *t5* , and their dependent, *t7* cannot be preserved. The DMS effectively rearranges the order of the original derivation history to delay the application of the failing transformations until last. This does not affect the original implementation, but moves *t6* and *t8* upward (not shown). A *nonlinear* representation of the derivation history should be used to avoid actually commuting the failed transformations with their followers. Transformations *t6* and *t8* can be preserved as *t4'* and *t5'*, leaving Δ3 alone and producing Δ4, respectively. Finally, DMS operates as a conventional transformation system to generate *t6'* and *t7'*, ultimately deriving the final maintained program in the lower right box.

The traditional maintainer does not have access to all of this information. He must start with the program in the lower left, and using some informal description of the change from his manager, somehow decide to produce the program in the lower right. While that may look easy for this small example, in practice it is very hard because of the scale of the code. The missing information, made explicit in this diagram, gives some indication why this is such a hard task, and why tools like DMS should enhance the productivity of the maintenance programmer.

REVERSE ENGINEERING

It is a grim fact that the typical organization desiring to make change has only has the system code, with possibly some informal, inaccurate documentation and some understanding of the code distributed across the maintainers. Consequently, the typical organization could not carry out our method for maintenance by design modification directly. Worse yet, since the paradigm calls for a program formally derived from a specification, and the conventional program was constructed by informal means, it is not clear that a design history can be constructed by any means.

Reverse engineering is one means to recover lost design information. Program understanding methods [NING], [RIC90], [QUILICI] represent one approach by which reverse engineering may be accomplished. Such methods use a library of program clichés, and match the clichés against the code. Where matches occur, the cliché abstraction becomes a plausible explanation for the code.

It is assumed that with sufficient clichés, a complete (possibly overlapped) tiling of the code can be obtained—consider each cliché to be a tile covering some portion of the code—thereby, providing a complete description of the code. There are a number of flaws to this approach:

1. There is an assumption that one can get a complete set of widely acceptable clichés. We subscribe to the notion that there are a large number of problem domains, and that clichés are needed for each.
2. There is the potential of huge computational demands if one attempts to tile a large system at once. Present cliché recognition research is attempting to understand the scaling issues for just 10,000 lines of code and a small number of clichés [QUILICI]
3. A complete tiling of the code only raises the abstraction level somewhat. For a large system, it would seem that one should tile the tiles repeatedly to get to the highest level of abstraction possible.
4. The purpose of Reverse Engineering (RE) is generally to aid informal understanding of the code (not mechanical modification). RE usually results in the production of informal documents under the implicit assumption that the code will be constant (often by virtue of being thrown away).
5. Since code maintenance always changes the code, the RE activity must be repeated for each maintenance event (aggravating problem 2). No knowledge is accumulated in a directly tool-reusable form. Yet, the one luxury we have with maintenance is the long period of time over which related activities recur.
6. Implemented code has all kinds of optimizations that entangle the implementation of abstractions, which disables recognition of clichés. As an example, the *file-data-access* abstraction in the data processing program code can only be found once, although the specification indicates two separate data accesses.

One approach for obtaining a design history is to generalize cliché recognition in a way that solves these problems. The key observation is that every cliché is a **<abstraction, code template>** pair, which can be treated as a transformation rule. We can use the power of a transformation engine to recognize clichés and abstract them. In essence, we are analyzing the code by synthesizing a plausible explanation of it as a derivation from a plausible specification.

It is not necessary to recover the entire design to make changes. One need only recover the part that will be impacted by the change. Further, it is not necessary to maximally raise the level of abstraction at the change site; one merely needs to raise it enough so the maintainer gets advantage over doing the task manually. The change itself and the informal understand of the software maintainers can be used to focus the recovery activity to the crucial part of the design.

We have applied the reverse-engineering method by hand to assembly code in an operating system, to recover the semaphore abstraction it implemented. The final specification is the domain-specific term, P(x: semaphore) [DIJKSTRA]. The derivation history is the set of transformations that map the abstraction into the highly optimized assembly code. The flavor of this is easily obtained by considering the final code in Figure 5, and discovering the transformations bottom up.

SUPPORTING TECHNOLOGY

There are two challenges to implementing the DMS vision. First one must have sufficient integrated infrastructure to carry out the steps. Second, it must scale reasonably well. Required infrastructure includes the following.

A means for representing the program to be maintained. A hyper-graph substrate is used to encode language-specific graph representations. For procedural languages under DMS, we will be using advanced compiler representations [CYT91] enabling data and control flow analysis, with extensions that encode memory access and synchronization conditions. This aids cliché recognition [RIC90] and allows more powerful transforms than abstract-syntax tree representations.

A graph rewrite engine to apply individual transformations to the program representation enhanced with the ability to record, view and revise design histories.

Tools to manage a database of notations, abstractions, and transforms that might be used in the application. The application domain language to which they apply categorizes these.

Reverse-engineering tools, which use the rewrite engine to recognize clichés taken from domains, and propose them to a maintainer, or allow unrecognized code fragments to be added to a domain as an instance of a domain abstraction.

A domain-notation driven structure editor to allow maintainers to inspect and point at portions of partially derived applications in the appropriate domain notation.

SCALE

Scale management for DMS occurs at two levels: the size of the application system, and the number of engineers who maintain it. DMS is arguably inappropriate for applications having only several thousand lines or less; maintenance programmers are able to handle these on an

individual basis. DMS should be more effective for systems with hundreds of thousands of lines, which are common, but cannot be handled by individual maintainers. Here the captured domain knowledge and the relationship between the specification and the actual code should be of great value to the maintainers. Since such systems are always evolving, DMS must support the simultaneous efforts of large teams of maintainers. Organizations with 5 million lines of code often have hundreds of software engineers working on the system every day. DMS supports large-scale applications by several means.

DMS is implemented in a parallel processing language, ***Parlanse***, [Bax96] running on easily obtained Windows/NT multiprocessor workstations. Our experience with transformational code synthesis systems such as SINAPSE [KAN91], [BAX93], implemented on symbolic manipulation systems (e.g., Mathematica) showed that code synthesis alone sometimes required hours to generate some 5,000 lines, well short of our target ceiling of 10 million lines. Symbolic manipulation is expensive. ***Parlanse*** provides the DMS implementers with efficient support for compiler-managed forking and synchronization of fine-grain parallel processes on (symbolic) data structures, as well as software engineering support such as modules and robust exception management. (We note that we expect to maintain the DMS system, implemented in ***Parlanse***, using DMS with a ***Parlanse*** domain!).

DMS will not generate all 10 million lines of code when a change is made, but it might have to inspect a significant fraction of the 10 million transformations it has stored as the design of that system. We will use a nonlinear design history representation to capture dependencies between transform effects, rather than dependencies between transformation states.

Since DMS revises histories, it is not a requirement that DMS be able to automatically generate and apply transformations by itself. It should be sufficient that maintainers choose the transformations one by one; re-implementation repair efforts should be small. However, DMS will have a built-in implementation of a programmable Transformation Control Language (TCL), to provide some automation.

We are implementing DMS as a client-server architecture in which the program design database will be held on a server, and the DMS interface will run on maintainer's workstations. An individual maintainer will implicitly lock the part of the design history he may be changing. On large systems, this should be a small portion of the design base, and multiple maintainers should be able to work with little interference.

Large application suites are never coded in a single language. The DMS notion of domain and domain networks will allow DMS to handle multi-language applications.

DOMAINS

The abstractions and notations useful for characterizing an application problem are often poorly expressed in the actual application program run on the computer. DMS will store and use multiple domains [NEI89] to specify and implement a single application at its various levels of abstraction. Such domains include but are not limited to target languages such as "C", Entity-Relationship diagrams, and graphical languages, as required for the application at hand. Each DMS domain has several parts:

External Form	string or graph view suitable for user
Internal Form	representation suitable for DMS
Semantics	meaning of the Internal Form
Parser	converts External to Internal Form
Unparser	converts Internal Form to External
Optimizations	transforms within a domain
Refinements	transforms between domains
Analyzers	how to analyze in the domain

This database is the repository of *reusable* knowledge for this domain. It is collected by explicit domain engineering episodes, and augmented by repeated reverse engineering activities during the maintenance process. DMS provides a Domain-Definition Domain in which other domains are defined.

NEW SOFTWARE ENGINEERING ROLES

If we have design capture and modification tools the tasks of the software engineer changes. In present practice, few software engineers specify; most spend their time designing algorithms, coding, and chasing bugs. With DMS, we see two classes of effort: *Domain* analysis and engineering versus *Application* analysis and engineering.

The Domain Analyst defines problem areas, formal notations describing problems, and the notation semantics. The Domain Engineer determines how one domain can be implemented in terms of existing domains already known to the system. The Application Analyst operates traditionally, determining the specific problem a customer needs solved, but unconventionally produces a specification in one or more domain notations established by the Domain Engineer. The Application Analyst is also the source of maintenance deltas related to functionality and performance of the individual product. Finally, the Application Engineer guides the DMS in the implementation of the specification, or those parts that must be re-implemented, but he does not program in the usual sense.

RELATED WORK

Wile [WIL83] proposed that transformational metaprograms, not code, be the major software product. [ARA86, WAT88] both describe an approach for porting software by recovering a plausible derivation leading to specification, and then re-implementing the specification, but the derivation is lost. *The Programmer's Apprentice* [RIC90] system recognized code idioms, converted them to abstractions by an inverse transformation process, and allowed a programmer to change the abstract program. Based on a model of transformational maintenance by reverse engineering [WAR89], a system for modifying COBOL programs has been implemented. Capture and incremental reuse of informal design plans is being pursued by [Jacq94] whose work may lead to a practical tool useful for handling conventional programs.

CONCLUSION

We have shown a scheme for capturing the design of transformationally synthesized code. Given the design, incremental changes can be installed by use of mechanical procedures and some additional transformational synthesis. Such capabilities should decrease the cost of maintenance, and therefore significantly lower the cost of software. *This payoff strongly suggests the value of treating design as the primary product of the software process, rather than code.*

Semantic Designs is funded by National Institute of Standards and Technology (NIST), Advanced Technology Program (ATP) under the Component-Based Software Initiative to build a scaleable prototype DMS system. We expect to have demonstrable capabilities in early 1998.

Bibliography

[All90] J. Allen and J. Hendler and A. Tate, eds., Readings in Planning, Morgan-Kaufmann, 1990.

[ARA86] G. Arango, I. Baxter, C. Pidgeon, P. Freeman, "TMM: Software Maintenance by Transformation", *IEEE Software* 3(3), May 1986, pp. 27-39

[BAX90] I. Baxter, "Transformational Maintenance by Reuse of Design Histories" Ph.D. Thesis, Information and Computer Science Department, University of California at Irvine, Nov. 1990, TR 90-36.

[BAX92] I. Baxter, "Design Maintenance Systems", *CACM* 35(4), Apr. 1992, pp. 73-89.

[Bax93] I. Baxter, "Practical Issues in Building Knowledge-Based Code Synthesis Systems", *Proceedings, Sixth Annual Workshop on Software Reusability*, Owego, New York, Nov. 1993.

[BAX96] I. Baxter, "***Parlanse***, A Parallel Language for Symbolic Expression", Semantic Designs, 1996 n.b. to be made available at www.semdesigns.com.

[CYT91] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph", *Transactions on Programming Languages and Systems* 13(4), Oct. 1991, pp. 451-490.

[DEK92] S. Dekleva, Delphi Study of Software Maintenance Problems, *Proceedings of 1992 Conference on Software Maintenance*, Nov. 1992.

[FEA86] M. Feather, "A Survey and Classification of some Program Transformation Approaches and Techniques", IFIP WG21 Working Conference on Program Specification and Transformation, Bad Toelz, Germany, Apr. 1986.

[GRA87] R. Grady, *Software Metrics: Establishing a Company-Wide Program*, Prentice-Hall, 1987.

[Jacq94] J.-P. Jacquot, "Programming through Disciplined Modification", *Proceedings of 1994 Conference on Software Maintenance*, Sep. 1994.

[KAN91] E. Kant, F. Daube, E. MacGregor, and J. Wald, "Scientific Programming by Automated Synthesis," *Automating Software Design*, Michael R. Lowery and Robert D. McCartney, eds., MIT Press, 1991.

[MCC87] R. McCartney, "Synthesizing Algorithms with Performance Constraints," Ph.D. Thesis, Brown University 1988.

[NEI89] J. Neighbors, "Draco: A Method for Engineering Reusable Software Systems", *Software Reusability*, T. Biggerstaff and A. Perlis, ACM Press 1989.

[PAR90] H. Partsch, *Specification and Transformation of Programs*, Springer-Verlag, 1990.

[RIC90] C. Rich and R. Waters, *The Programmer's Apprentice*, ACM Press, 1990

[TAM92] T. Tamai and Y. Torimitsu, "Software Lifetime and its Evolution Process over Generations", *Proceedings of 1992 Conference on Software Maintenance*, Nov. 1992.

[WAR89] M. Ward and F. W. Calliss and M. Munro, "The Maintainer's Assistant", *Proceedings of the 1989 Conference on Software Maintenance*.

[WAT88] R. C. Waters, "Program Translation via Abstraction and Reimplementation", *IEEE TSE* 14(8), Aug. 1988.

[WIL83] D. Wile, "Program Developments: Formal Explanations of Implementations", *CACM* 26(11), Nov. 1983.

[YANG94] H. Yang and K. Bennet, "Extension of A Transformation System for Maintenance -- Dealing with Data -Intensive Programs", *Proceedings of 1994 Conference on Software Maintenance*, Sep. 1994.

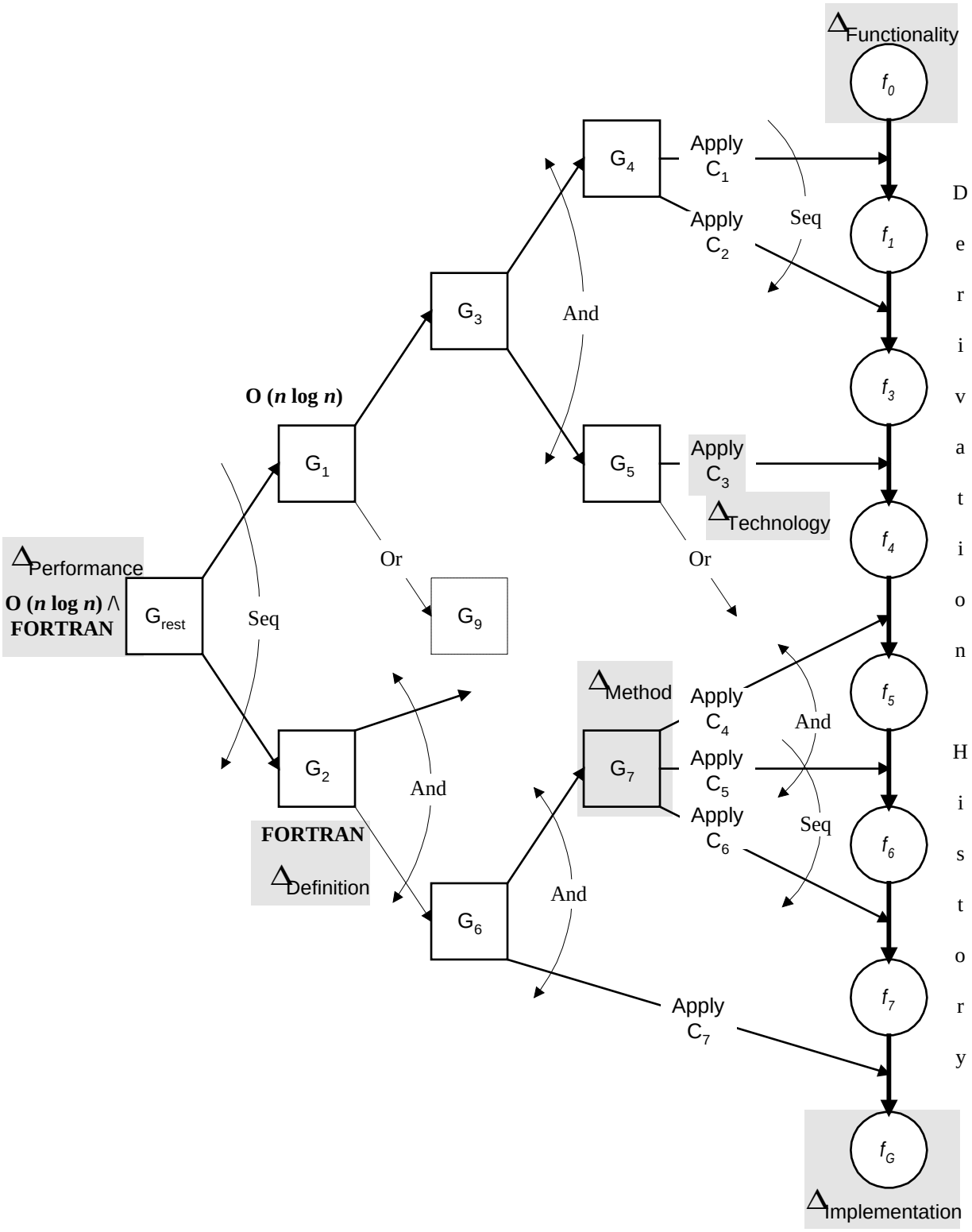

Figure 4: A Transformational Design Rationale
for an Implementation in terms of a Specification and applied Transformations

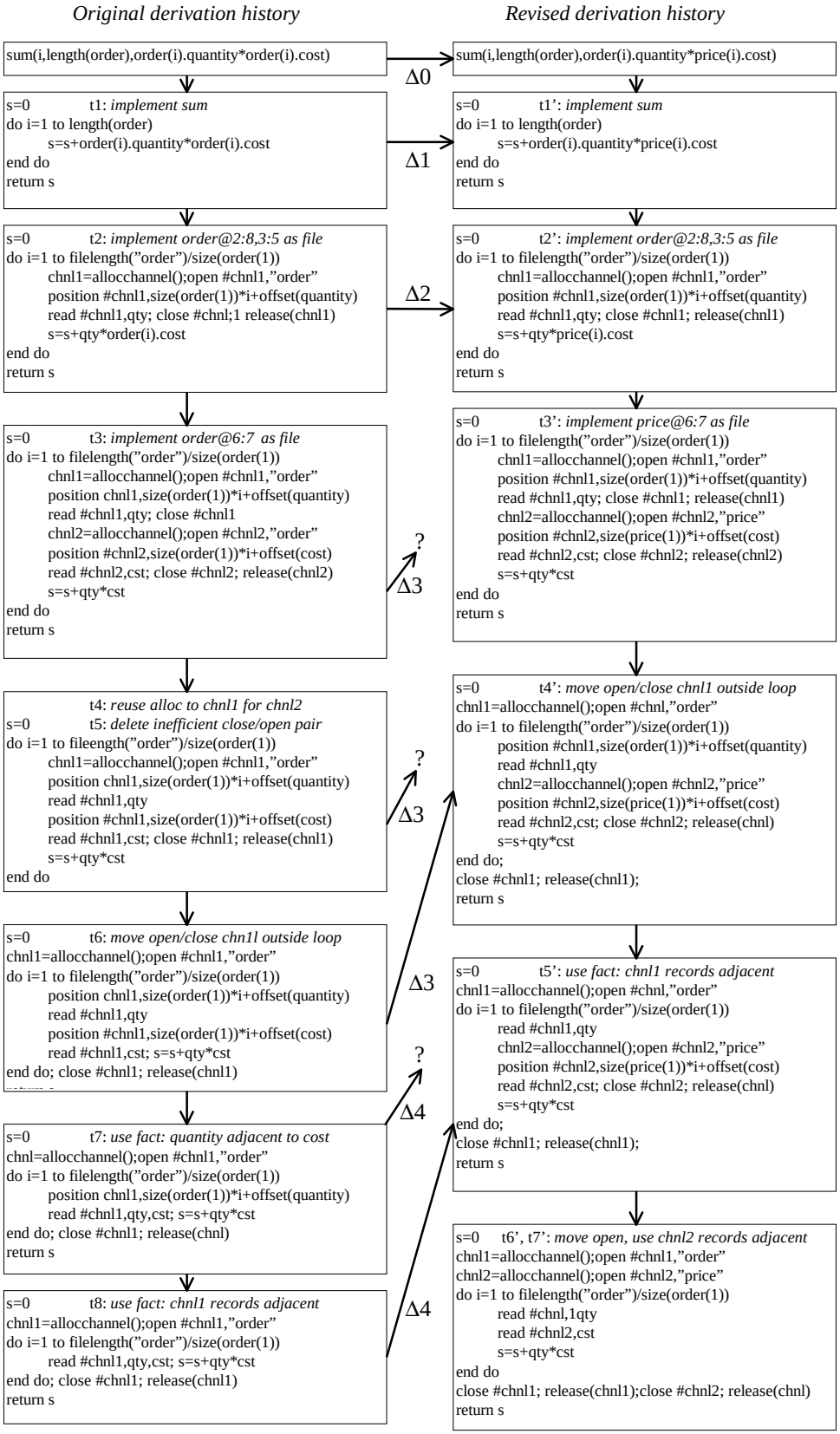

Figure 5: Preserving implementation steps using Δs as guidance