Design Maintenance Systems

Traditional maintenance has proven difficult in the absence of design information [19]; maintainers often spend considerable energy trying to recover this design information before making changes [15]. Capture and reuse of certain kinds of design information should consequently improve the maintenance process. Updating the design information is as important as revising the software itself, for the revised design information is necessary for further maintenance. Four major software engineering efforts that ended in disaster precisely because trying to maintain their designs was perceived as impractical are described by [23]. Consequently, each design was abandoned and the projects spiraled into chaos.

Rather than emphasize software maintenance, we suggest that *design maintenance* become the key focus, with programs being easily derived from a complete design. The details of our work lay foundations for such *design maintenance systems* (DMS). This article sketches such a system. In particular, it defines the kinds of information such a system must retain and a particular approach to capture and modify that information. For more details, the reader is referred to [6].

We believe that productivity advances in software construction and maintenance depend on automation, which, in turn, depends on formalization. Our approach focuses on theories and mechanisms necessary to allow formal *maintenance deltas* to be integrated into software systems constructed by rigorous, if not completely automatic, means. Such deltas express desired changes in program functionality, performance, and implementation technology. Figure 1 shows a life cycle model establishing the context for a DMS. Most of the mechanisms are concerned with comparing the existing artifact with some ideal, and producing a maintenance delta describing the error.

Ira D. Baxter

If one had a design maintenance system based on integrating formal deltas, then the life cycle notions of design/coding phases and maintenance phases become indistinguishable. Software would be constructed by incrementally revising a current design according to a continuous stream of maintenance deltas generated by comparing expected consequences of the current design with customer desires and available implementation technologies.

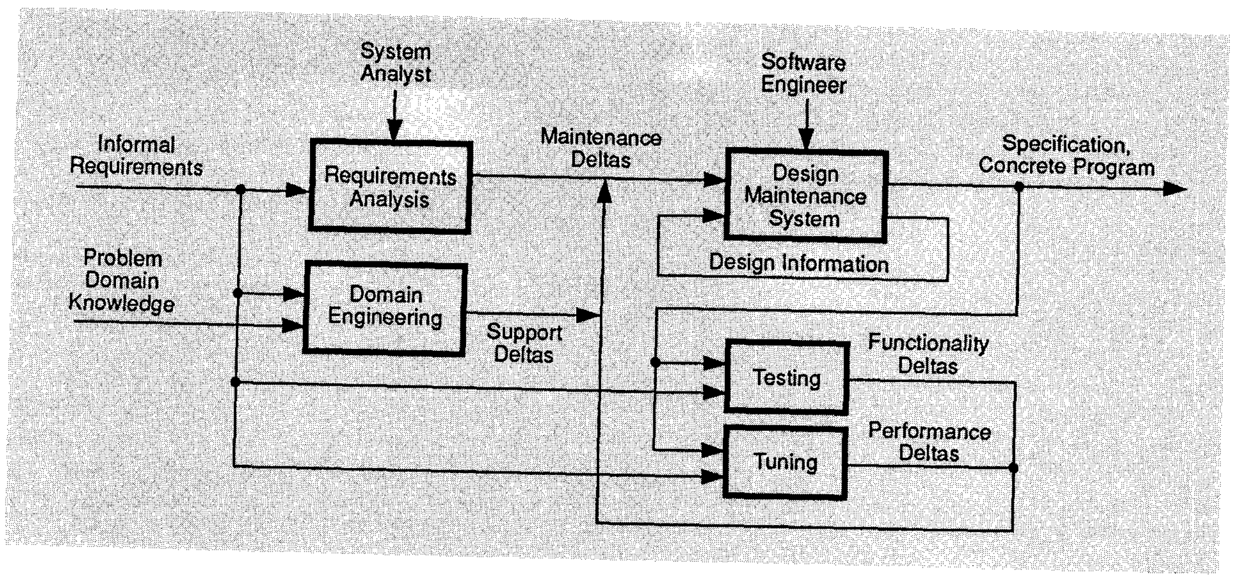


FIGURE 1. Incremental evolution by delta integration

Our particular approach to a DMS requires that:

- the software system be formally specified;
- its implementation be derived transformationally;
- the justification for the implementation be captured;
- desired maintenance deltas be formally specified;
- suitable tools for design justification modification exist.

Given this background, a DMS requires:

- a representation for programs
- a transformation engine
- an agenda-oriented metaprogramming language
- a representation for the justification
- justification revision mechanisms, which we call delta integration procedures

We address these in turn, with the bulk of this article being devoted to delta integration procedures.

Software Construction Model

Conventional software construction almost invariably loses two critical classes of design knowledge: the problem specification, and the design justification. The justification demonstrates that the implementation truly solves the problem stated by the specification. This knowledge is lost because of the informal nature of the construction process, and its dependence on smart agents (programmers) whose design knowledge and activities are largely contained in the agents, rather than in a formal database.

We choose formal transformation systems as a basis for a formal construction methodology. Such systems accept formal specifications (predicates defining functionality and performance goals to be achieved) and apply transformations to the specification to construct the final program. We use a library of heuristic methods coded in a Transformation Control Language (TCL) to control the application of implementing transforms. Each method formally relates a design plan, consisting of some set of activities ("how"), to a design purpose which states the effect ("what") the plan achieves. "How" is stated in the form of plans for applying specific transformations or other methods in the form of nonlinear plans (sets of actions and explicit sequencing information about their order of execution [8]). "What" is stated as predicates on arbitrarily defined performance measures over program regions. A set of TCL methods is used to nonprocedurally decompose the specification into solvable subproblems with their own specifications. The plans from methods chosen for a particular decomposition are executed to actually solve the subproblems.

Thus a set of TCL methods capture generative design knowledge. It is not design knowledge for a particular problem, but for a class of similar problems. Such knowledge is not only crucial to initial implementation, but is also necessary to regenerate portions of a related program resulting from a desired change.

Transformation System Model

Our particular transformational model is summarized in Figure 2. This model covers many of the existing transformation systems (CIP [5], MEDUSA [16], TAMPR [7], Draco [18], TI [14], REFINE [20, 22]). Practical specifications have two parts, $\langle f_0, G_0\rangle$, which we shall motivate shortly. A number of support libraries provide the necessary technology to implement the specification. Members c_i of a library of transforms $C_{library}$ are applied to convert an abstract operational specification into a concrete program. Performance measures are used to determine what properties a partially transformed program has, and produce results from a range specified by performance values. Subsumption relations define the notion of "at least as good as" over the measured performances. Library predicates combine performance measures and subsumption relations to define interesting levels of performance. The method library contains plans for applying transforms to achieve known performance levels for program regions.

Specifications

A transformation system requires that we somehow specify what program we desire. Such specification capabilities are also needed for describing the effects of transformational plans. The most general form of a specification is a predicate over programs, which we denote G.

Suppose we have two kinds of performance measures used to specify programs, functionality and computational complexity. A full specification of a sorting program p might then be:

$$G \equiv sorted(p(x)) \wedge complexity(p(x)) = O(length(x) \cdot \log(length(x)))$$

As a practical matter, transformation systems often avoid specification of functionality as a predicate by requiring, instead, an operational description (i.e., an abstract program) whose semantics satisfy the desired functionality. Thus one might specify the functionality f_0 of p by the operational program *sort*(X), whose semantics are defined as *sorted*(*sort*(x)), with another predicate G_0 containing the balance of the predicate G. The mixed specification $\langle f_0, G_0 \rangle$ for p is then

$$\langle sort(X),\ complexity(p(x)) = O(length(x) \cdot \log(length(x)))\rangle$$

The transformation system then need only apply correctness-preserving transformations to the program f_0, such that the other performance goals G_0 are achieved.

Transforms, Locators and Transformations

We define a *transform* to be any function which maps programs into programs. Often, a transform is applied to a particular place of a larger program; we designate such places with *locators*. We call the application of a transform t at place ℓ a *transformation* and denote it as t^{ℓ}. If we use a correctness-preserving transform, we generally use c rather than t. Any particular program representation will determine how the transforms and the locators are represented. A common program representation is an abstract syntax tree; tree transforms are represented by pairs of trees containing pattern variables, and locators are represented by paths, a possibly empty sequence of integers that describe how to navigate from the root of the tree to the root of a subtree where the tree transform will actually be applied. Each element of a path selects a numbered branch, counting from left to right. We denote tree transformations by writing

$$pattern \Longrightarrow pattern@\langle path\rangle$$

Figure 9 shows several such tree transformations, written as linear strings with *?letter* as pattern variables, and the effect of applying them to particular trees. A different type of transformation is a *refinement*, which atomically maps entire programs at one level of abstraction to another, and is shown vertically in Figure 10. Being applied everywhere, refinements do not need a locator.

Transformational Planning Components

Multiple transforms may apply to a program; in the case of *sort*, it may be implemented by refining it into a bubblesort or a mergesort (Figure 3). TCL provides ways of choosing transformations that produce desired effects, by providing a language for stating both procedural and nonprocedural actions for achieving such effects. A TCL method is a plan for achieving an explicit performance property, stated as a plan postcondition, over some portion of a program. Procedural actions in plans are those to be taken by the transformation system, usually the application of a transform (*APPLY*) at a designated place in the program, the invocation of a named set of actions (*CALL*), or the determination of a

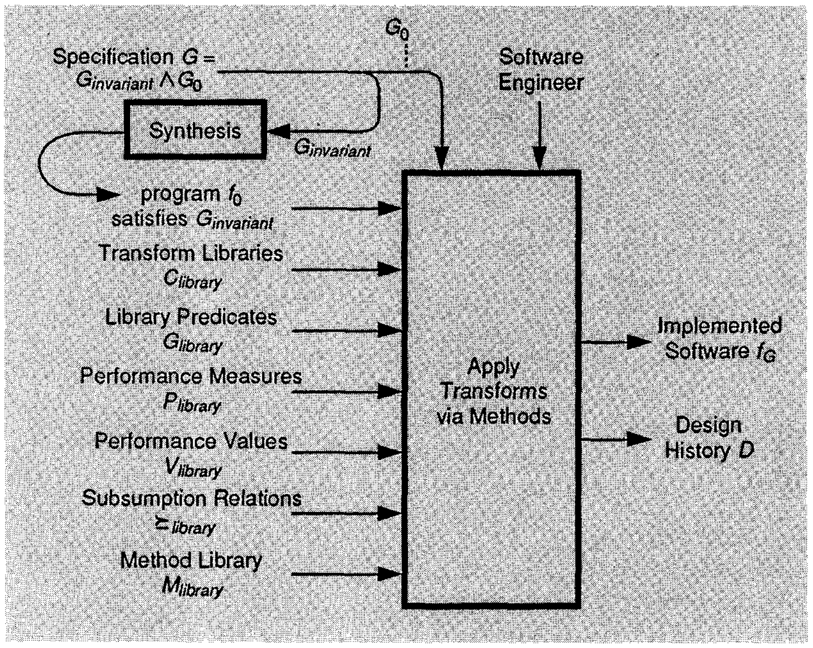

G_0
Specification $G = G_{invariant} \wedge G_0$
Software Engineer
Synthesis
$G_{invariant}$
program f_0 satisfies $G_{invariant}$
Transform Libraries $C_{library}$
Library Predicates $G_{library}$
Performance Measures $P_{library}$
Performance Values $V_{library}$
Subsumption Relations $\succeq_{library}$
Method Library $M_{library}$
Apply Transforms via Methods
Implemented Software f_G
Design History D

FIGURE 2. Transformation system controlled by Methods.

place or region relative to another (*LOCALE*). Nonprocedural actions are goal-oriented statements of purpose, to be achieved by consulting a library of existing TCL methods for relevant plans. The postcondition states an expected property of the program after the method's actions have been applied. Figure 4 shows a TCL method $m_{mergesort}$ for implementing a provably "n log n" procedure for *sort* by refining it into a merge sort, with performance constraints (*ACHIEVE*) on the implementations of the components of the merge sort. The method plan is to sequentially (*SEQ*) apply the refinement transform, producing a locator lv_0 for the refined code, and then to ensure (*PLAN*) that the various parts of the refined code are implemented correctly. A partial order on *PLAN* steps is generally needed to specify the essential sequencing requirements on the transformation process; *SEQ* is just a specialized type of plan in which all steps must be executed in a particular linear order. In the example, sequencing of the subplan steps is not needed, as long as all plan steps succeed. Failure of any part of the plan causes the method to fail; backtracking allows the transformation system to try other methods.

```
c_bubblesort: sort(X) ⇒ . . . codeforbubblesort . . .

and

c_mergesort: sort(X) ⇒
        split(X)to(A, B);
        sort(A);
        sort(B);
        merge(A, B)to(R);
        return(R)
```

FIGURE 3. Transforms that implement *sort*

```
m_mergesort(lv) = ⟨ postcondition : p_complexity(lv) = O(n log n) ⟩
                  ⟨ action : a_mergesort(lv)                    ⟩
 with

 a_mergesort(lv) =
    SEQ(LET(lv_0, APPLY(c_mergesort, lv)),
         PLAN(LOCALE(lv_1, 1thson(lv_0)), ACHIEVE(p_complexity ≥ O(n), lv_1))
                LOCALE(lv_2, 2thson(lv_0)), ACHIEVE(p_complexity ≥ O(n log n), lv_2))
                LOCALE(lv_3, 3thson(lv_0)), ACHIEVE(p_complexity ≥ O(n log n), lv_3))
                LOCALE(lv_4, 4thson(lv_0)), ACHIEVE(p_complexity ≥ O(n), lv_4))
                LOCALE(lv_5, 5thson(lv_0)), ACHIEVE(p_complexity ≥ O(1), lv_5))
                ∅) % no ordering on plan steps
         )
```

FIGURE 4. A TCL method implementing *sort* with *O(n ln n)* performance

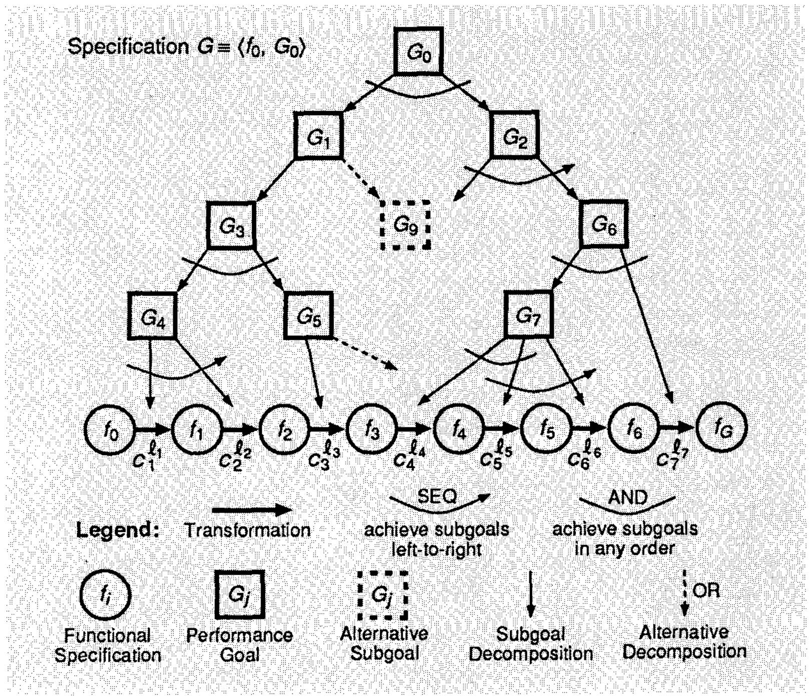


FIGURE 5. Transformational design history including unfolded design plans

Design Capture

Design capture requires that we know:

- What was desired—specification
- How it was achieved—implementation actions
- Why the implementation works—justification

Use of a transformation system, plus a transformational planning language such as TCL, allows us to capture this information. The formal specification tells us what is intended and is trivial to capture. The sequence of transformations actually applied tell us how the generated program was constructed, and is captured as a linear *derivation* history[1]. The motivation for application of each transformation is captured by storing a trace of the nonprocedural unfolding of goals during the course of execution of the TCL methods. We call such a trace a *design* history. Encoding alternative decompositions for (sub)specifications as TCL methods captures design alternatives.

Figure 5 shows an abstract design history (DH). The conceptual specification G is written as a mixed specification $\langle f_0, G_0\rangle$. The functional specification portion f_0 is transformed repeatedly, producing intermediate specifications f_i, and

[1]The derivation history can be stored nonlinearly, but that has little effect on our theory.

eventually the final program f_G. The performance goal G_0 is decomposed by selection of subgoals achievable by TCL methods, that in turn eventually provide the needed transformations. Goal nodes in the design history are called *agenda* items, because those on the frontier of an incomplete design history represent uncompleted actions or further goals to be achieved. Arrows between specification nodes represent the transformations which were applied. The leaf nodes plus transformations constitute a part of the design history called a derivation history. A crucial detail of the derivation history is not only which transform were applied (c_i), but the locators ℓ_i indicating where they were applied. The design history also records dependencies on the order of execution of agenda items.

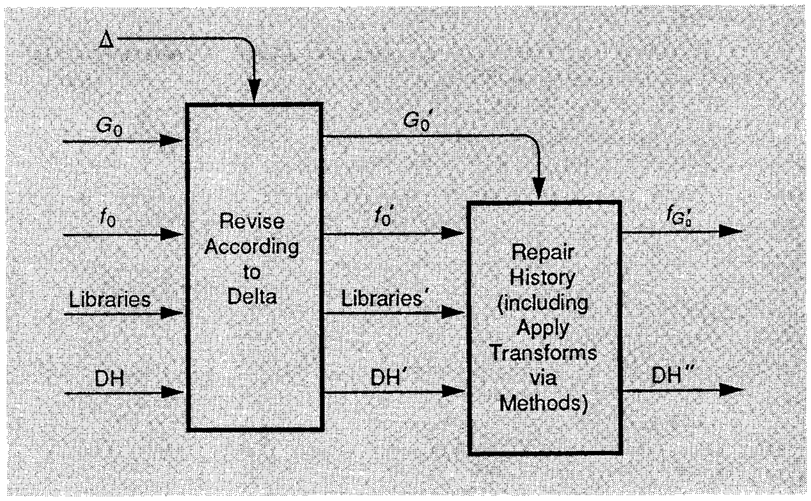

Δ
G_0
G_0'
f_0
Revise According to Delta
f_0'
Libraries
Libraries'
DH
DH'
Repair History (including Apply Transforms via Methods)
$f_{G_0'}'$
DH''

FIGURE 6. Updating design according to maintenance delta

Design Maintenance

One reason conventional software maintenance is difficult is that the kind of information present in a design history is missing. The only information the conventional maintainer possesses with certainty is the final program f_G. Much of a maintainer's work consists of informally reconstructing the relevant portion of the design information, even before a change is made. To make a change, one modifies f_G directly to produce f'_G. The next maintainer must repeat this laborious process with f'_G. Such reconstruction also brings with it the frequent problem that the rediscovered design information is incorrect, which introduces faulty assumptions and consequently, bugs into the maintained program. Conventional software-engineering practices can help alleviate this problem, but since most design documentation is informal, not directly or causally coupled to the code, and not of primary interest to the end customer or corporate management, the presence of such documentation only slows the slide into code-only maintenance. Such design information is abandoned when the maintainers perceive the code to be more accurate than the documentation. It is our contention that focus on maintaining the code eventually leads to loss of the design.

Rather than focus the maintenance process on the end code, our approach is to *maintain* the design, and derive the program from the design. This ensures that the design stays up to date, because it is the primary artifact.

In a transformational context, the problem of design maintenance is one of updating the specification, the derivation history, and the design history in a way consistent with any new desire, stated as a maintenance delta (Figure 6). The implementation is easily generated by sequentially applying all the transformations in the revised derivation history to f'_0.

Maintenance Deltas

Before making a change, the desired change must somehow be codified. Traditional maintenance classifies change types into *perfective, adaptive,* and *corrective* [14, 25]. Although this classification may be useful to management, it is of little use to the maintainer in guiding the modification process.

By enumerating the inputs of a formal transformation system, we can exhaustively and formally define the types of formal maintenance deltas that can affect the outputs (i.e., the implemented program). We can then use knowledge of how a transformation system operates to define procedures for handling each type of delta.

Deltas come in two fundamental varieties: specification deltas (those that affect the problem definition) and support deltas (those that affect how the solution is implemented). Specification deltas model understanding of dissatisfaction with the presently implemented program. Support deltas model improvements over time of the expertise of the developers. We denote the type of a delta by Δ_{type}, and a particular instance delta by δ_{type}. For our particular transformation system model we found the types of maintenance deltas shown in Figure 7. (See box for explanation of all the delta types covered by our model.)

We believe that the majority of deltas are specification deltas: Δ_f, those that describe changes in desired system function, and Δ_G, describing change in desired performance.

Each type of maintenance delta has a specific representation. A nice synergy with transformation systems is that a functionality delta δ_f can be represented directly as a (noncorrectness-preserving) transformation, t^ℓ. A performance delta

δ_G can often be represented as a pair of sets of performance predicates, $G_\ominus$, removing performance constraints, and $G_\oplus$, adding new performance constraints. We show procedures for integrating such deltas into a design in the following section; we have procedures for integrating all the delta types.

Integrating a Delta into a Design History

Design maintenance must start with a design history. We can construct one from scratch by running a transformational implementation on a chosen specification, or, as a last resort, by reverse-engineering such a history from an existing system [3]. Since we believe that generating such a history is expensive, because of the need for the transformation system to consider many implementation alternatives, we almost always use a revised history from a previous design maintenance step. Given a particular maintenance delta, the problem then becomes one of revising the existing history, using the transformation system as little as possible.

We do not address the question of how a maintenance delta is chosen, or how to handle justifications for the particular delta. Because the rate at which problems are detected with implemented software can exceed the rate at which such problems are resolved, a number of individual maintenance deltas are frequently bundled together as a single composite delta. While such composite delta may be processed by breaking it into its individual parts and processing each sequentially, there is economy in the design maintenance process if the composite delta is handled as a batch, in that many interacting effects of the changes are handled at once. However, in this article we will consider only individual maintenance deltas.

Revising the specification component of design information is simple: apply a specification delta to the current specification. Support deltas are applied to the transformation system components; they do not affect the statement of the specification, but can affect its meaning or implementation.

What remains is to revise the design history to be consistent with whatever delta has been applied. This is accomplished by:

1. rearranging and pruning the derivation history part of the design history according to the delta,
2. pruning away parts of the design history which no longer serve a useful purpose,
3. using the stored TCL methods to regenerate any incomplete part of the design history.

The detailed mechanics of the first two steps are determined by the type of the delta. The general mechanisms are similar. The regeneration process uses the transformation system.

Revising and Pruning Derivation Histories

A derivation history represents a particular trajectory through an implementation space from a functional specification to a solution. Often many other equivalent tra-

Types of Maintenance Deltas

Performance deltas (Δ_C) to C are the essential specification changes. Such changes affect the performance aspects of the desired program. They are generated when a customer compares an implementation against reality, and discovers points of difference between what was specified and the current requirements. An example of a performance delta is a change of desired implementation language from $C_{FORTRAN}$ to C_{PROLOG}. The practice of using mixed specifications effectively limits such changes to C_0.

Performance bound deltas (Δ_V) are a special kind of performance delta. These arise when some performance bound is either too loose, so the final artifact is unsuited for its ultimate application, or too tight, and the desired artifact cannot be built at a reasonable cost. A typical performance bound delta might be to change a $p_{complexity}$ performance bound from $O(n^2)$ to $O(n)$. Many times, revising one performance bound specification will require adjusting another performance bound specification; as an example, a tighter time bound usually requires a looser space bound.

Functional deltas (Δ_F) occur because of the practice of providing mixed specifications (f_0, C_0). Such changes are generated whenever the actual semantics of f_0 are discovered to not meet the requirements; in a conventional software development environment, these would correspond to changes made to a functional specification. An example functional change would be modifying the functionality $f_0 \equiv \sin^2(x-3)$ to be $\sin(x+1)$. The evolution transforms of Johnson [12] are examples of functional deltas; initial specifications (f_0's) in the form of GIST programs are modified by applying a series of built-in evolution transforms (Δ_F's) to convert an initial f_0 into functional specifications believed to be better suited to the problem at hand. In practical systems, we would expect the bulk of changes made to be functional changes; performance tuning usually comes after achieved functionality.

Technology deltas (Δ_C) occur when software engineers realize that a desired transform does not exist in the library of transforms available to the transformation system, or when an existing one is discovered to be incorrect. An example of such a delta is the change from an incorrect *LISP* transform (*cons ?x t*) $\Longrightarrow$ (*list* ?x) to the correct version (*cons ?x nil*) $\Longrightarrow$ (list ?x). Incompleteness of the library is expected because of the impracticality

jectories from the same specification to the same solution may be obtained by rearranging the sequence of the transformations. Given a particular delta, an existing derivation history can be rearranged into two parts, much like separating oil and water. The first part consists of those transformations that are still legitimate in the face of the particular delta; the second part consists of transformations (and their dependents) that conflict with the delta.

We determine the preservability of an individual transformation of the derivation history by attempting to prove that it "commutes" with the delta. If it does, application of the delta can be delayed past the preservable transformation. Such proofs often succeed, and are even very efficient, because the transformations and deltas in programs of moderate or larger size tend to be applied far "apart" in the program, in terms of purpose or effect. This is one reason that most code in maintained programs does not change. Special techniques are used on transformations and deltas that are applied near one another; sometimes both the preserved transformation and the delta must be adjusted to account for their mutual interference, extending the notion of "commutes." An interesting example of this is pushing a delta through a transformation that changes abstraction levels (e.g., a transformation from abstract lambda calculus into machine code can map a lambda calculus delta into a machine code delta). If our conservative proof fails, we banish the offending transformation and its dependents to the second part of the derivation history.

The second part of the derivation history is *reuseless* with respect to the delta, and must be removed. The first part is a legitimate derivation history for a partially implemented program and can be preserved intact. Additional transformations may be needed to fill out the derivation history and obtain a complete implementation; but first, the remaining transformations must be revalidated. We will return to this topic later. The remainder of this section is devoted to showing how a derivation history can be revised given a particular functional delta, δ_f. The mechanism can be used as a component of other delta propagation procedures.

Understanding the relationship of a presently existing derivation history to its revised version helps clarify our procedures. Applying a change δ_f (shown as δ_0) to the functional part of a specification completely changes the implementation space from that of f_0 to f'_0, in which the new implementation must be found (Figure 8). In one sense, the original derivation history is entirely irrelevant, requiring an entirely new derivation to be constructed in the new space. In another sense, there should be a close analog of the original path in the new space.

From the beginning of the new path (f'_0), the transformations from the original implementation space can be tried sequentially. Each applicable transformation can still be

of engineering a complete transform library before using the system [4, 9]. Errors in existing transforms will occur simply because human designers are fallible; many transformation systems (DRACO, REFINE, TI) allow domain engineers to install and use transforms without verification of correctness. Even correct transforms can be invalidated if the problem domain to which they apply changes, as is expected in the domain-engineering process [2]. We consequently expect that technology changes will be necessary during both program construction and maintenance.

Method deltas (Δ_M) capture knowledge of new implementation techniques, or fix errors in existing techniques. For example, an error in the method implementing mergesort (Figure 4), a complexity goal of *O*(n^2) performance in locale *lv*$_4$, would require a method delta to correct it. Such changes occur for the same kinds of reasons that technology changes occur. Some methods will be applicable over a broad range of programs. Unlike technology changes, however, our expectation is that for each program implemented, some new methods will be generated. This is primarily due to our inability to encode effective heuristics for implementing every possible program [4], and because of limits on the completeness of the control mechanisms.

Performance predicate library deltas (Δ_C) change the vocabulary available to the transformation system to express performance specifications and/or postconditions of methods. Such changes occur when new performance predicates are added, or existing ones are deleted due to lack of space or utility, or revised due to error. An example of a performance predicate delta is changing the definition for "reasonable-size" of modules.

Other deltas: Deltas relating to performance measurement functions Δ_P, changing a subsumption ordering over a set of performance values $\Delta_{\geq}$, or to the range of performance values Δ_v are possible, but expected to be rare. Such changes indicate an error on the part of the domain analyst defining these entities, or their incorrect implementation in the transformation system proper.

Deltas not covered by the model: Because our performance model only includes measures of the resulting artifact, there is another class of changes that we do not address: changes to *process* performance predicates, or constraints over resources consumed by the transformation system while constructing an artifact.

- Specification deltas:
  - Δ_G : Change of performance goal (G_0)
  - Δ_V : Change of performance bound measure
  - Δ_f : Change of functionality (f_0)
- Support deltas:
  - Δ_C : Change of technology (transform library)
  - $\Delta_{\mathcal{G}}$: Change of performance predicate library
  - $\Delta_{\mathcal{P}}$: Change of performance measurement functions
  - $\Delta_{\succeq}$: Change of orderings $\succeq_i$
  - $\Delta_{\mathcal{V}}$: Change of range of performance values
  - $\Delta_{\mathcal{M}}$: Change of method library

FIGURE 7. Types of delta induced by transformation system model

legally applied (as exemplified by $c_1^{\ell_1}$ and $c_2^{\ell_2}$). Transformations that no longer apply (such as $c_3^{\ell_3}$) could simply be dropped (naive replay), leaving the applicability of their successors ($c_4^{\ell_4}$) in doubt. We depart from naive replay by using a more sophisticated technique to preserve many successor transformations: If an inapplicable transformation commutes with its successor, we can delay application of the inapplicable one and attempt to preserve its successor (note that $c_4^{\ell_4}(c_3^{\ell_3}(f_2)) = f_G = c_3^{\ell_6}(c_4^{\ell_5}(f_2))$; this allows us to propose $c_4^{\ell_5}$ when $c_3^{\ell_3}$ fails to be preservable). Since the commutativity of transformations depends on their semantic properties, such commutation fails only if there is a true order dependency on transformations.

We revise a derivation history by repeatedly applying either a *delay* or *preserve* action to each individual transformation it contains.

Delaying a transformation: We can often determine that a particular transformation in a derivation history cannot be preserved. We can revise the derivation history to delay its application as long as possible. A primitive for accomplishing delay is the notion of swapping two sequential transformations, in order to delay the application of the first one. Given a swap mechanism, we may repeatedly swap a troublesome transformation down a derivation history. Delaying the application of a transformation as far as possible is called banishment.

Swapping transformations often requires a generalized kind of commutativity. The effect we desire is given a derivation history fragment $c_2^{\ell_2}(c_1^{\ell_1}(p))$, we need to find a new fragment $c_1^{\ell_1'}(c_2^{\ell_2'}(p)) = c_2^{\ell_2}(c_1^{\ell_1}(p))$, such that $c_2^{\ell_2'}$ is known to be preservable. We can then replace the original fragment with the new fragment, effecting a unit delay of c_1 by changing its locator. We accomplish this by computing candidate ℓ''s and testing for equivalence of the composition of the original pair with the composition of the swapped pair. An example for tree transformations is shown in Figure 9, demonstrating that application of the distributive law can be delayed past application of the commutative law by revising the locators on both, without changing the effect. When such a swap cannot be computed, then the transformation has been delayed as far as practical, and no further attempt is made to delay its application further down the derivation history.

Preserving a transformation: Given a functionality delta δ, we often want to determine if a transformation c_i^{ℓ} from a derivation history can be preserved. If we can find another pair of transformations δ' and $c_i^{\ell'}$ that preserve the effect of δ, then we can reuse the application of c_i. This is formalized as $c_i^{\ell'}(\delta(f_i)) = \delta'(c_i^{\ell}(f_i))$.

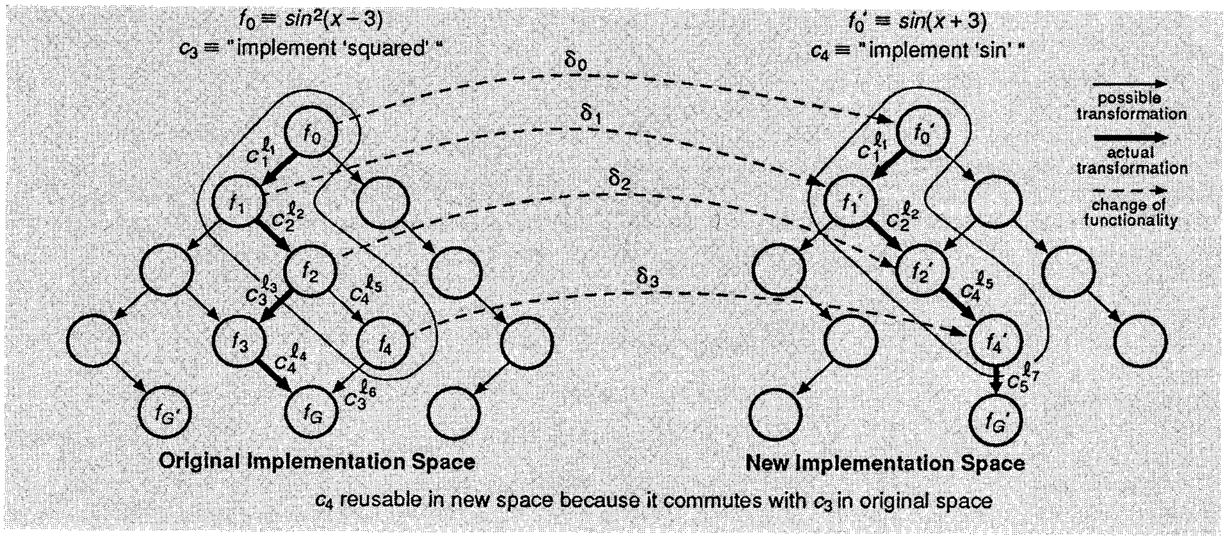


FIGURE 8. Changing functionality: preservation of path across implementation

This is also practical to compute for certain kinds of transformations. In Figure 10, we see that the step of refining a stack-computation program into *LISP* can be preserved by computing $\delta' = c(\delta')$. The revised δ' is computed by applying the refinement to the original delta.

We have so far seen how to revise a derivation history when we have been told, directly or indirectly, which transformations simply cannot be kept. Functionality deltas provide us with the opportunity to directly inspect interactions between individual transformations in the derivation history and some desired functionality change. What we attempt to do is to preserve as much of the derivation history as possible. The essential idea is to "push" the delta through the derivation history, from beginning to end, either preserving or banishing as we go. We scan the original derivation history from beginning to end, checking the delta for interference with each transformation. When the existing transformation interferes with the desired change, we simply banish the offending transformation; otherwise we can preserve that transformation. The scan stops when an existing step can be neither preserved nor banished; the derivation history is truncated at the offending transformation. The truncated derivation history is compatible with the revised delta. All we need to do is actually apply the delta and finish the implementation, using the transformation system itself. A detailed example of derivation history revision is provided in the Appendix.

Revising Design Histories

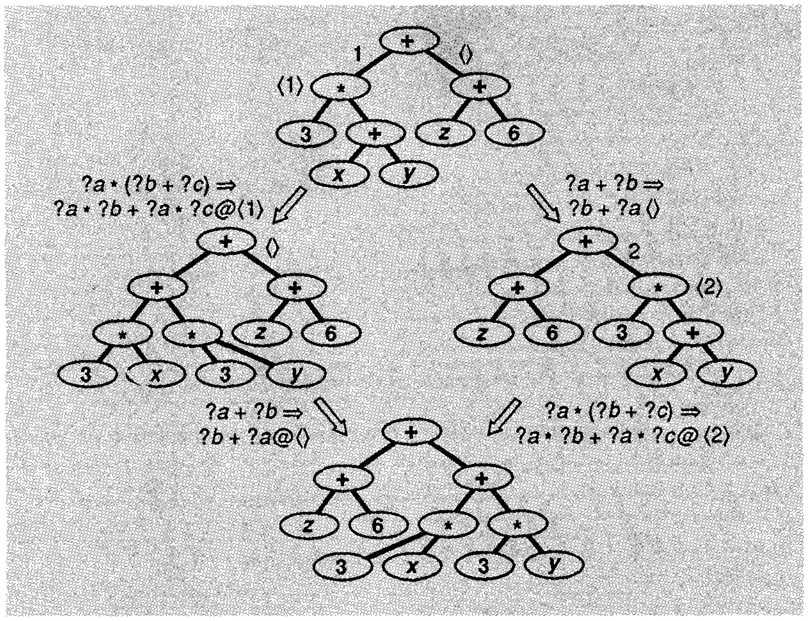
Different maintenance deltas affect different aspects of the transformation system, and consequently require different procedures for revising the design history. Each such procedure must determine the impact of the delta on the design history and adjust it accordingly. As an example, functional deltas affect the derivation history portion directly. Indirect effects occur as a side effect of modifying the derivation history.

All of our design history delta-integration methods follow the same general sequence:

Revise: Adjust structures in delta-specific fashion;

Mark: Identify agenda items in the design history inconsistent with the delta;

Prune: Prune away inconsistent agenda items and all the forced choices dependent on those already



?a * (?b + ?c) ⇒ ?a * ?b + ?a * ?c@⟨1⟩
?a + ?b ⇒ ?b + ?a ⟨⟩
?a + ?b ⇒ ?b + ?a@⟨⟩
?a * (?b + ?c) ⇒ ?a * ?b + ?a * ?c@⟨2⟩

FIGURE 9. Swapping order of two overlapping sequential transformations

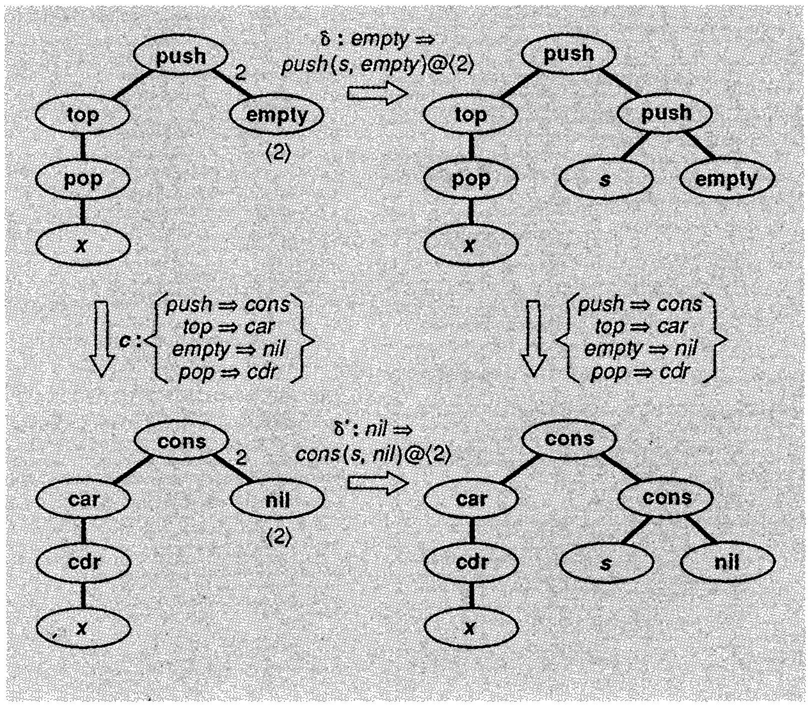


FIGURE 10. Preserving a refinement translates the delta

pruned; and
Repair: Complete the pruned design history, perhaps by using new information supplied by a delta.

As with modifying a derivation history, these steps are interleaved in practice, for both individual deltas, and for the multiple deltas that make up a composite delta.

The revision process usually inspects the design history relative to a particular delta, and, while revising it, marks those parts (agenda items) which conflict with the delta as undesirable. How the conflicts are detected depends on the type of the delta. Many delta types indirectly cause transformations to be eventually banished, revising the derivation history portion of the design history, and marking the agenda items that generated the banished transformations.

Undesirable agenda items are then pruned away, leaving an incomplete design history. If the pruned agenda item is a forced (i.e., only) choice on which some other agenda item is dependent, then it is only sensible that the depending agenda item must also be pruned. Pruning an agenda item leaves its parent incomplete. This part of the integration process is independent of the type of delta. The pruning of agenda items often invalidates transformations in the derivation history, which must eventually be removed.

Having finished the pruning process, the incomplete design history must be repaired by processing incomplete agenda items. We accomplish this by simply passing the incomplete design history to the TCL execution engine, which chooses and executes incomplete agenda items. We must of course ensure that pruning a design history always leaves it in a legal state as far as the TCL execution engine is concerned. The execution engine records the relation between agenda items and their progeny, thus filling out the design history. In this fashion we avoid the need for a special replay mechanism. Completion for delta integration and completion for initial implementation are identical. Repairing may cause further revision, marking, and pruning.

Pruning and Repairing the Design History

The purpose of pruning a design history is to remove those parts which are either simply invalid or no longer serve any purpose relevant to the final performance specification, G_0'. We assume we have a design history in which some agenda items have been marked undesirable. We must prune away:

- all portions of the design history that are directly marked,
- every agenda item that depends uniquely on some pruned agenda item, and
- agenda items generated as descendants of those marked.

What remains after pruning is a design history containing incomplete agenda items having alternative completions.

To prune an undesirable agenda item, the design history is traversed from that item upward until some parent agenda is found that provides an alternative (*OR* or *ACHIEVE* nodes). That item is then marked as *incomplete,* and all agenda items below that point are removed from the design history. When pruning a leaf agenda item that *APPLY*s a transformation, we additionally mark the transformation in the derivation history as undesirable; during the plan repair process, an eventual banish will remove the marked transformation, which may eventually cause additional pruning.

In Figure 11, we show a typical pruning process. A sequence of activities is numbered:

1. Mark agenda item G_n undesirable
2. Prune the undesirable agenda item and its dependents
3. Mark dependent transformations as undesirable
4. Banish an indirectly dependent transformation G_7
5. Mark, as undesirable, the agenda item generating the transformation

Some delta-specific marking process first marks G_7 as undesirable. At the pruning step, traversal moves up the design history from undesirable G_7 to the first parent having an alternative G_9. That item is marked as *incomplete*, and all of its descendents (the outlined region containing G_{10}, G_8, G_7, and the unshown nodes that *APPLY* transformations $c_4^{\ell_4}$, $c_5^{\ell_5}$ and $c_6^{\ell_6}$) are removed from the design history. All the transformations under the pruned region are also marked as undesirable. Eventually, but not as part of the pruning process for G_7, some derivation history banishing activity, triggered by the need to remove $c_4^{\ell_4}$, $c_5^{\ell_5}$, $c_6^{\ell_6}$, will encounter $c_7^{\ell_7}$; should this transformation itself also need banishing, then its immediate parent (*APPLY* under G_6) will be marked undesirable and the pruning process repeated.

Repairing the pruned plan consists of executing actions for incomplete agenda items, perhaps generating additional agenda items in the process, and recording the new items in the design history. Since such agenda items can be produced by the pruning process in the middle of the logical transformational implementation process (according to the sequencing constraints in the design history), to repair a design history we must have:

- out-of-order execution of TCL methods and fragments
- the ability to insert transformations in the middle of the derivation history

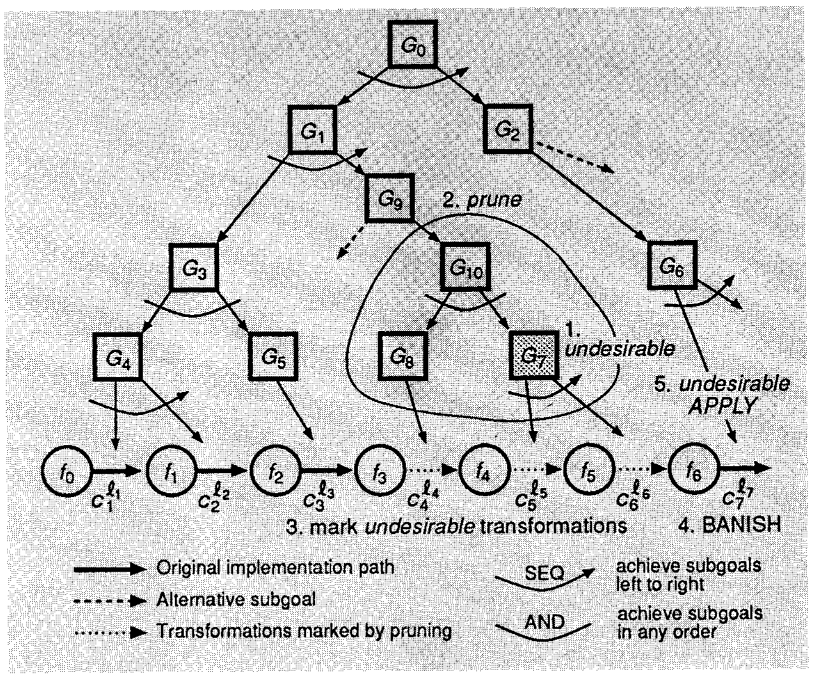
The execution order for a metaprogramming language like PADDLE [26] or the tactics language of Goldberg [11] is totally determined, and very difficult to restart at arbitrary points. This is why such metaprograms are replayed in their entirety from the beginning. Rather than be saddled with a purely linear execution model for the metaprogram, we designed TCL execution in such a way that an agenda-oriented execution process is possible. Agenda items are produced by TCL language constructs when encountered, and processed in the order determined only by the sequencing constraints defined by *PLAN*s.

Agenda-oriented execution process chooses any incomplete item and executes it (possibly causing the addition of more agenda items) and then marks it complete. We choose to process the earliest incomplete agenda item, as determined by the ordering constraints in the design history. This performs those actions with the most potential ripple effect on the remainder of the design history as early as possible. The earliest-first heuristic minimizes the amount of later revision required, by doing early actions while the design history is as small as possible. It also spreads damage from invalidated earlier decisions to later points in the design history before we attempt to build on top of them.

Each agenda item specifies a TCL action taken from some TCL method which determines what occurs when the agenda item is executed. Most typical is the execution of a *PLAN* action, and the most potentially complicated is the execution of an *APPLY* action; we will discuss these shortly.

An incomplete agenda item must be executed according to its action and produce a satisfactory subagenda. Some of the agenda item types allow only a single way to obtain satisfactory completion (*PLAN, CALL, REQUIRE*), and some allow many alternatives (*ACHIEVE, APPLY, OR*).

For alternative generators, execution generates the next possible alternative, based on internal state of the agenda item, and records it in the design history. If there are no remaining alternatives, then the agenda item is unsatisfiable, and is simply pruned, exposing an even higher-level agenda item that provides an alternative. Pruning the design history back to an agenda item resets its alternative generator.



G_0
G_1
G_2
G_3
G_4
G_5
G_6
G_7
G_8
G_9
G_{10}
2. prune
1. undesirable
5. undesirable APPLY
f_0 f_1 f_2 f_3 f_4 f_5 f_6
$c_1^{\ell_1}$ $c_2^{\ell_2}$ $c_3^{\ell_3}$ $c_4^{\ell_4}$ $c_5^{\ell_5}$ $c_6^{\ell_6}$ $c_7^{\ell_7}$
3. mark undesirable transformations
4. BANISH
Original implementation path
Alternative subgoal
Transformations marked by pruning
SEQ
achieve subgoals left to right
AND
achieve subgoals in any order

FIGURE 11. Pruning a design history back to an alternative

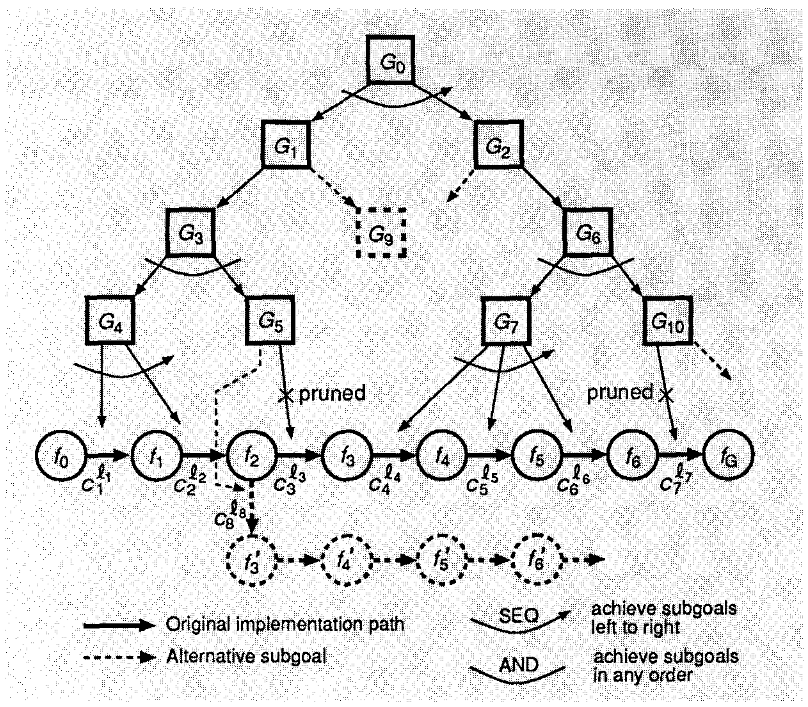
Execution of *PLAN* agenda items occurs in a way similar to expanding a node in a hierarchical nonlinear planning process [8, 13]. The design history is augmented by creating new, incomplete agenda items for each element of the *PLAN*. An execution-ordering relation is placed on these newly created agenda items according to the ordering given by the *PLAN* action. It is the lack of order present in many plans that provides much of the freedom to reorder execution of actions in the design history.

Execution of *APPLY* agenda items requires the application of a transformation to a state. If we had a simple linear execution model, we could simply apply the transformation to the last state of the derivation history, but with plan repair we can have out-of-order execution. The transformation might be applied to any state in the derivation history.

When choosing agenda items to execute, we want to choose the earliest to maximize the appearance of any possible downstream effects. When applying a transformation, we want to apply it to the latest state to which it can legitimately apply, because we will have to revise the derivation history from that point on, and we wish to minimize the effort to do so. The information necessary to determine this is present in the ordering information in the design history.

When incomplete *APPLY* agenda items occur late enough according to the ordering information in the design history, the point of application turns out to be the end of the derivation history. This scheme thus conveniently subsumes the simple linear execution model.

An interesting case is shown in Figure 12, in which $c_3^{\ell_3}$ has been marked as undesirable, and the design history has then been pruned back to G_5, which offers the alternative $c_8^{\ell_8}$, shown by the dashed arrow. According to the sequencing information in the design history, this alternative transformation must be applied just prior to the earliest sons of G_7 and G_{10} (i.e., it should replace the third transformation). This is accomplished by



G_0
G_1
G_2
G_3
G_9
G_6
G_4
G_5
G_7
G_{10}
pruned
pruned
f_0 f_1 f_2 f_3 f_4 f_5 f_6 f_G
$c_1^{\ell_1}$ $c_2^{\ell_2}$ $c_3^{\ell_3}$ $c_4^{\ell_4}$ $c_5^{\ell_5}$ $c_6^{\ell_6}$ $c_7^{\ell_7}$ $c_8^{\ell_8}$
f_3' f_4' f_5' f_6'
Original implementation path
Alternative subgoal
SEQ
achieve subgoals left to right
AND
achieve subgoals in any order

FIGURE 12. Repair by inserting a replacement transformation

treating the new transformation to be applied as a functional delta δ_f to f_2 and using a variation of the Δ_f-integration procedure to construct a new derivation history. Such a new derivation history is shown in dotted outlines, growing horizontally in the figure, with the new transformation inserted in the middle.

The revised derivation history must have all retained transformations revalidated to ensure that they continue to serve the purpose for which they were originally intended. This is accomplished by checking that each transformation continues to play its designated role as specified by the design history; postconditions of parent agenda items are checked as transformations are preserved. An apparent preservable transformation which fails to achieve a desired effect must be banished; this is accomplished by marking its parent, which will cause a later cycle of prune and repair.

Delta-Specific Revise and Mark

Having sketched how a design history is pruned and repaired, we now outline revision and marking procedures for the common delta types. Each such procedure revises the design history in a delta-specific way, and marks the part of the design history that is no longer relevant.

Integration of functional deltas Δ_f into a design history is easy because plan repair already does most of the work. There is no need to initially mark any part of the design history at all. It is only necessary to *APPLY* the functional delta δ_f to f_0. The derivation history revision mechanism mentioned earlier will propagate the changes into the design history appropriately.

Integration of performance deltas Δ_G: Performance deltas change the specification of the desired artifact (performance bound deltas δ_v are simply a special case of Δ_G, and so essentially the same integration procedures can be used). We have seen that typical specifications are often given as mixed specifications $\langle f_0, G_0\rangle$. Since we have a mechanism to handle changes to f_0, we limit performance deltas to changes of G_0. Changing G_0 requires that a different path through the *same* implementation space (left half of Figure 8) be chosen to an alternative implementation $f_{G'}$. The choice of path through the implementation space is controlled by decomposition of the performance goals allowed by TCL methods in $M_{library}$. The decomposition of the performance goal for the current artifact is stored in the structure of the design history. Integration of Δ_G requires that we revise this decomposition.

We revise the design history by propagating the specific δ_G in a top-down fashion, paralleling the design history construction method by goal decomposition. As we walk down the design history, at each *ACHIEVE(G,e)* agenda item (see Figure 13), we must do the following:

- Revise the *ACHIEVE(G,e)* agenda item to be *ACHIEVE*($\delta_G(G)$,*e*)
- Decide if *CALL*ing method *i* is still useful as a means of decomposing the revised goal
  —If still useful, determine the changes induced by δ_G and propagate those into both subplans.
  —If not, prune the non-procedurally generated subplan.

Determination of the continued utility of the method $m_k = \langle G_k, action_k\rangle$ requires that we revalidate the goal decomposition for the agenda item. This can often be done by simple substitution. The revised subgoals must be propagated into the subplans; propagation should be into earlier subplans first to ensure that damage caused to early plans is propagated to later plans by the pruning process, ensuring that effort to update such later plans is only applied to their useful parts.

If we cannot revalidate the goal decomposition, then the existing plan to implement *ACHIEVE(G,e)* is not valid for the new

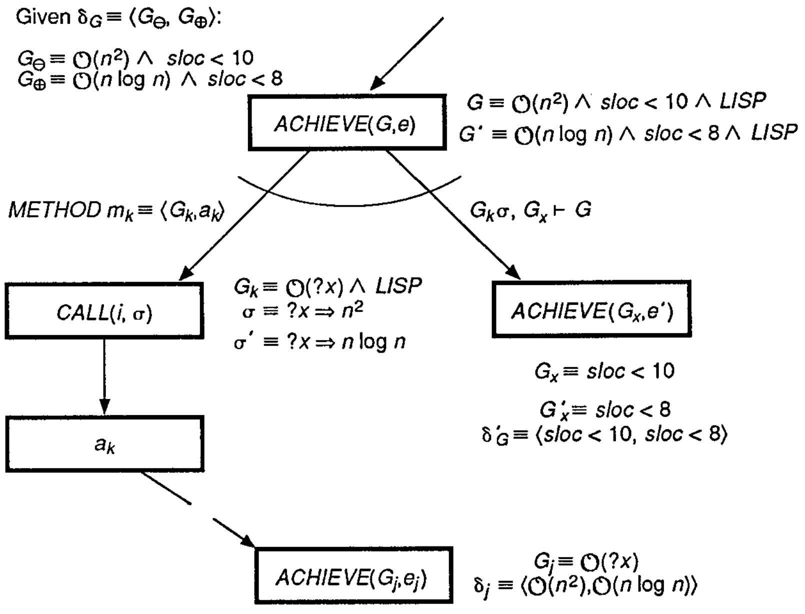
$ACHIEVE(\delta_G(G),e)$. We force the eventual pruning of the generated plan by marking both the *CALL* and the $ACHIEVE(G_x,e')$ nodes as undesirable, and terminate the propagation of δ_G into this subagenda. An eventual pruning process will prune the plan back up to the revised $ACHIEVE(\delta_G(G),e)$ node, and the repair process will attempt to find a new replacement.

Figure 13 shows how this occurs for a specific case. We start with an original performance goal G_0 of $O(n^2)$ with source lines (*sloc*) less than 10 and target language *LISP*. We want to apply a δ_G having two parts, $G_\ominus$, removing constraints, and $G_\oplus$, adding new constraints. The design history shows that the original performance goal was achieved by decomposing the problem into an action *CALL*ing a known method m_k whose postcondition ensures an arbitrary complexity and a specific target language, and then *ACHIEVE*ing non-procedurally the remaining part of the original performance goal, the desired *sloc*.

The design history revision requires that the new performance goal be propagated into the subagenda items. This requires that the former goal $p_{complexity}$ condition of $O(n^2)$ be revised to $O(n \log n)$ and propagated into the body of the called method. This is just another performance delta δ_j and is propagated by similar techniques. Similarly, a revised performance delta δ'_G for *sloc* is constructed and propagated. Should there be no way to revise the postconditions to match a performance delta, then the corresponding agenda item is marked as undesirable.

DMS Usage

The obvious use of a DMS is the construction of an incremental maintenance system. System analysts compare needs against existing system specifications, and produce maintenance deltas. The deltas are integrated into the existing design history for the existing software artifact, producing a revised history



Given $\delta_G \equiv \langle G_\ominus, G_\oplus \rangle$:
$G_\ominus \equiv O(n^2) \wedge sloc < 10$
$G_\oplus \equiv O(n \log n) \wedge sloc < 8$
$ACHIEVE(G,e)$
$G \equiv O(n^2) \wedge sloc < 10 \wedge LISP$
$G' \equiv O(n \log n) \wedge sloc < 8 \wedge LISP$
METHOD $m_k \equiv \langle G_k, a_k \rangle$
$G_k\sigma,\ G_x \vdash G$
$CALL(i,\sigma)$
$G_k \equiv O(?x) \wedge LISP$
$\sigma \equiv ?x \Rightarrow n^2$
$\sigma' \equiv ?x \Rightarrow n \log n$
$ACHIEVE(G_x,e')$
$G_x \equiv sloc < 10$
$G'_x \equiv sloc < 8$
$\delta'_G \equiv \langle sloc < 10, sloc < 8 \rangle$
a_k
$ACHIEVE(G_j,e_j)$
$G_j \equiv O(?x)$
$\delta_j \equiv \langle O(n^2), O(n \log n) \rangle$

FIGURE 13. Propagating δ_G from root to leaves of design history

and a revised artifact.

Since the procedures are insensitive to whether the design history is complete or not, one can apply deltas using a DMS to a partially completed implementation. This allows a DMS to be used during the initial implementation phase. The repair process can be stopped after any agenda item is processed, and a new delta applied. Thus deltas can be applied at will. Such a DMS would allow us to perform transformational maintenance relatively cheaply and lead toward the goal of an Incremental Evolution lifecycle.

A DMS can also be used as a foundation for a reusability system. Implemented components are stored with their specifications and design histories in a library. A reuser chooses a component whose specification is near his desires; the difference is a delta and is applied to the stored history to help revise the component.

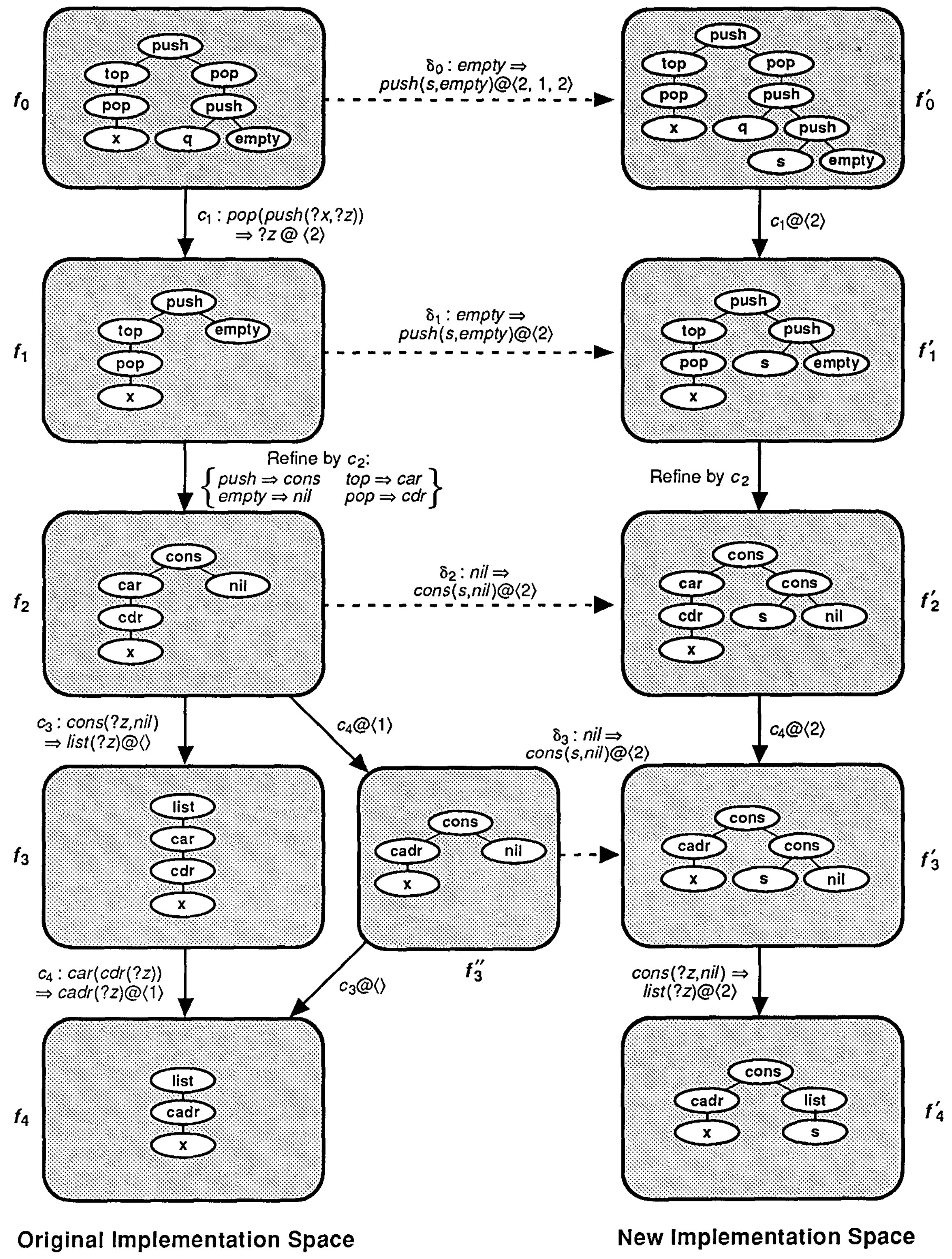
Status: A prototype system that takes a derivation history and a functional delta, using conditional tree transformations and theory morphisms has been tested. It generated the example in Appendix B, as well as a number of similar examples. Work on an implementation of the design history delta integration procedures has started.

The other components of a DMS are relatively well understood, making it practical to consider building an experimental DMS. Application of a practical DMS is still probably several years away, because of the need to build considerable transformational infrastructure, the need to encode considerable implementation knowledge, and to test the ideas on systems with real scale.

Related Work

A short survey of the state of software maintenance is provided by [21]. A fundamental observation is that for conventional maintenance, it is difficult to determine what code is affected by a change. A DMS acquires this traceability from a design history.

PDS [9] is a transformation system that retains derivation histories, and rederives components dependent on changed components. This is a kind of transforma-



f_0
δ_0 : empty ⇒ push(s,empty)@⟨2, 1, 2⟩
f'_0
c_1 : pop(push(?x,?z)) ⇒ ?z @ ⟨2⟩
c_1@⟨2⟩
f_1
δ_1 : empty ⇒ push(s,empty)@⟨2⟩
f'_1
Refine by c_2: {push ⇒ cons, top ⇒ car, empty ⇒ nil, pop ⇒ cdr}
Refine by c_2
f_2
δ_2 : nil ⇒ cons(s,nil)@⟨2⟩
f'_2
c_3 : cons(?z,nil) ⇒ list(?z)@⟨⟩
c_4@⟨1⟩
δ_3 : nil ⇒ cons(s,nil)@⟨2⟩
c_4@⟨2⟩
f_3
f'_3
f''_3
c_4 : car(cdr(?z)) ⇒ cadr(?z)@⟨1⟩
c_3@⟨⟩
cons(?z,nil) ⇒ list(?z)@⟨2⟩
f_4
f'_4
Original Implementation Space
New Implementation Space

FIGURE 14. Δ_f-Integration (replay) using a derivation history tional MAKE, in which there is no explicit delta to help guide the revision process [10].

A system for maintaining existing software by reverse-engineering an existing concrete program into an abstract program, applying an explicit functional delta to the abstract program, and reimplementing the abstract program is described by [24]. An earlier view of maintenance with explicit deltas, and how those deltas affect navigation of the implementation space, is provided in [3]. Neither of these consider the rationale needed to achieve desired performance properties.

Others have suggested enhancing the transformational maintenance process by reusing generative design information (goal-directed plans) [11, 26]. What distinguishes our work is the use of target-program-specific design information and the use of formal deltas to guide revision of this information. The problem of constructing a design history is very close to that of planning [1], with the difference being primarily the requirement to use correctness-preserving transformations. Many of the general ideas on what parts of a plan need repair are summarized by [17]. Work on plan repair and reuse [13] is often limited to sets of facts rather complex structures such as programs with indirectly derived properties, although the mechanisms bear considerable similarity.

Conclusions

We have sketched a DMS based on capture and reuse of design information from a transformational implementation process. The DMS updates the specification ("what"), a derivation history ("how"), and the design history ("justification") using a formal maintenance delta to guide the revision process. A system demonstrating some of the concepts has been built and tried on small examples.

Such a DMS could radically change software life cycles by unifying design and maintenance into a process of integrating a stream of formal deltas into the design, rather than focusing solely on the production of code. It is, perhaps, possible to translate software implemented conventionally into a form compatible with DMS by using transformational design recovery techniques [3, 24].

An incremental system for design revision need not be fast, if the amount of change per increment is small. This suggests that formal software construction methodologies may be justifiable solely because the design knowledge can be formally captured and reused. This in itself may justify their use even if one ignores any other positive benefits such as removing ambiguity from specifications, raising the quality of implementations, and the potential for automation.

Acknowledgments
Special thanks go to former members of the Advanced Software Engineering project at the University of California at Irvine for moral and technical support, including Peter Freeman, Y.V. Srinivas, and D.L. Rector.

References

1. Allen J., Hendler, J., and Tate A., Eds. *Readings in Planning.* Morgan Kaufmann, San Mateo, Calif., 1990. ISBN 1-55860-130-9.
2. Arango, G. Domain engineering for software reuse. Ph.D. thesis, Department of Information and Computer Science, University of California at Irvine, July 1988. ICS-RTP-88-27.
3. Arango, G., Baxter, I., Freeman, P., and Pidgeon, C. TMM: Software maintenance by transformation. *IEEE Softw. 3,* 3, 27–39, May 1986.
4. Balzer, R. A 15-year perspective on automatic programming. *IEEE Trans. Softw. Eng.* SE-11, 11 (Nov. 1985), 1257–1268.
5. Bauer, F.L., Moller, B., Partsch, H., and Pepper, P. Formal program construction by transformations—Computer-aided, intuition-guided programming. *IEEE Trans. Softw. Eng. 15,* 2 (Feb. 1989), 165–180.
6. Baxter, I.D. Transformational maintenance by reuse of design histories. Ph.D. thesis, University of California at Irvine, Nov. 1990. Tech. Rep. 90–36. Also available from University Microfilms International, Catalog #9109634.
7. Boyle, J.M. and Muralidharan, M.N. Program reuseability through program transformation. *IEEE Trans. Softw. Eng. SE-10,* 5 (1984), 575–588.
8. Charniak, E. and McDermott, D. *Introduction to Artificial Intelligence.* Addison-Wesley, Reading, Mass., 1985. ISBN 0-201-11945-5.
9. Cheatham, T.E., Jr., Holloway, G.H., and Townley, J.A. Program refinement by transformation. In *Proceedings of the Fifth International Conference on Software Engineering,* (San Diego, Calif., Mar. 1981), pp. 430–437. Reprinted in *New Paradigms for Software Development,* William W. Agresti, Ed., IEEE, 1986, ISBN 0-8186-0707-6.
10. Feldman, S.I. Make—A program for maintaining computer programs. *Softw.—Prac. Exp. 9* (Apr. 1979), 255–265.
11. Goldberg, A. Reusing software developments. Tech. Rep., Kestrel Institute, Aug. 1989. 3260 Hillview Avenue, Palo Alto, Calif., 94304.
12. Johnson, W.L. and Feather, M. Building an evolution transformation library. In *Proceedings of the Twelfth International Conference on Software Engineering.* IEEE Computer Society Press, Mar. 1990.
13. Kambhampati, S. Flexible reuse and modification in hierarchical planning: A validation structure based approach. Ph.D. thesis, University of Maryland, Oct. 1989. Tech. Rep. CAR-TR-469, CS-TR-2334, Computer Vision Laboratory, Center for Automation Research, College Park Maryland, 20742-3411.
14. Lientz, B.P. and Swanson, E.B. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations.* Addison-Wesley, Menlo Park, Calif., 1980.
15. Littman, D.C., Pinto, J., Letovsky, S., and Soloway, E. Mental models and software maintenance. In *Empirical Studies of Programmers,* E. Soloway and S. Iyengar, Eds., Ablex, Norwood, N.J., 1986, pp. 80–98.
16. McCartney, R. Synthesizing algorithms with performance constraints. Ph.D. thesis, Brown University, 1988. Department of

Computer Science Tech. Rep. CS-87–28, Dec. 1987.

17. Mostow, J. Some requirements for effective replay of derivations. In *Proceedings of the Third International Machine Learning Workshop,* (Skytop, Pa., June 1985), pp. 129–132.

18. Neighbors, J. The Draco approach to constructing software from reusable components. *IEEE Trans. Softw. Eng. SE-10,* 5 (Sept. 1984).

19. Ning, J.Q. A knowledge-based approach to automatic program analysis. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1989. UIUCDCS-R-89-1548.

20. Reasoning Systems Inc. *REFINE User's Guide.* Reasoning Systems, Inc., Palo Alto, Calif., 1986.

21. Schneidewind, N.F. The state of software maintenance. *IEEE Trans. Softw. Eng. SE-13,* 3 1987, 303–310.

22. Smith, D.R., Kotik, G.B., and Westfold, S.J. Research on knowledge-based software environments at Kestrel Institute. *IEEE Trans. Softw. Eng. SE-11,* 11 (Nov. 1985), 1278–1295.

23. Sneed, H.M. The myth of top-down software development and its consequences for software maintenance. In *Proceedings of Conference on Software Maintenance,* (Miami, Fla., Oct. 1989), IEEE Computer Society Press, pp. 22–29. ISBN 0-8186-1965-1, IEEE Catalog Number 89CH2744-1.

24. Ward, M., Calliss, F.W., and Munro, M. The maintainer's assistant. In *Proceedings of Conference on Software Maintenance 1989,* (Miami, Fla., Oct. 1989), IEEE Computer Society Press, pp. 307–315. ISBN 0-8186-1965-1, IEEE Catalog Number 89CH27441-1.

25. Wedo, J.D. Structured program analysis applied to software maintenance. In *Proceedings of Conference on Software Maintenance* (Wash. DC, 1985), pp. 28–34. IEEE. ISBN 0-8186-0648-7.

26. Wile, D. Program developments: Formal explanations of implementations. *Commun. ACM 26,* 11 (Nov. 1983), 902–911. Also available from University of Southern Calif., Information Sciences Institutes as

APPENDIX. An Example of Δ_f Integration

This appendix provides a concrete example (Figure 14) effectively reusing a derivation history by integrating a functional delta.

We have chosen an example problem domain consisting of stack computations. The key ideas are stacks-as-values, and operations that push and pop entities onto stacks, producing new stack values. A particular expression from the stack domain is provided as the functional specification, shown in tree form inside the box labeled f_0 in the figure. To keep the example uncluttered, we leave the balance of the specification G_0 implicit, but we assume it includes $p_{language}(f) = LISP$ and some unstated computational efficiency goal.

The leftmost column of the diagram shows a derivation history. The boxes labeled f_0 through f_4 down the left side are a series of design states traversed, with f_4 being an implementation of f_0. The arcs form a derivation history with mixed types of transformations. Transformations c_1, c_3, and c_4 are tree transformations with path locaters; c_2 is a refinement mapping stack expressions into Lisp. Transform c_1 is a simplification in the stack domain. Transforms c_3 and c_4 are simplifications possible in the Lisp domain. They are not possible in the stack domain. These simplifying transforms are obtained from algebraic specifications of the domains, and are part of the library of transforms available to the transformation system. The transformational implementation process follows that of the Draco system [18] in its style of repeatedly performing optimize-within-domain then refine-to-new-domain.

Similarly, the rightmost column is a derivation history, starting with a different specification f'_0, and carrying through various transformations and refinements. The horizontal dashed lines show how one derivation history maps into another via application of the deltas. The reader may wish to compare this figure with the enclosed regions of Figure 8; the only difference is that this figure is more detailed.

A problem to be solved by transformational maintenance is, given:

- f_0
- the leftmost derivation history
- the functional delta $\delta_0 \equiv empty \Longrightarrow push(s, empty)@\langle 2, 1, 2\rangle$

how can f'_4 and the rightmost derivation history be generated, running as little of the transformation system control process as possible? Intermediate states f_1, f_2, and f_3 are presumed unavailable because of the expected high cost of storing every state. Since this is a maintenance situation, we can assume we also have the implemented program f_4, but it will turn out to be unnecessary. All we really need is the derivation history.

We start with state f_0, with existing transformation $c_1@\langle 2\rangle$ and desired $\delta_0 \equiv t_0@\langle 2, 1, 2\rangle$, $t_0 \equiv empty \Longrightarrow push(s, empty)$. Set the new derivation history to empty. We compute $f'_0 = \delta_0(f_0)$, and save it.

Step 1. Computing $PRESERVE(f_0, c_1@\langle 2\rangle, \delta_0@\langle 2, 1, 2\rangle)$ produces the result $c_1@\langle 2\rangle$, $t_0@\langle 2\rangle$, thereby producing the revised transformation $c_1@\langle 2\rangle$ to append to the new derivation history. We have avoided invoking transformation system control. We compute $f_1 = c_1@\langle 2\rangle(f_0)$.

Step 2. Computing *PRESERVE*(f_1, c_2, t_0@⟨2⟩) produces the result c_2, t_1@⟨2⟩ with $t_1 \equiv nil \Longrightarrow cons(s, nil)$, essentially by applying the refinement to both parts of t_0. We have again avoided use of transformation system control. Append $c2$ to the new derivation history. We compute $f_2 = c_2(f_1)$, and discard f_1.

Step 3. We attempt to compute *PRESERVE*(f_2, c_3@⟨⟩, t_1@⟨2⟩), which fails because of the interaction between t_1 and c_3 over the simultaneous removal and required presence of *nil*, respectively. There is simply no way to preserve transformation c_3@⟨⟩. We therefore banish c_3@⟨⟩, demoting c_3 below c_4. This demotion is shown in the subderivation history which branches from state f_2 and continues down the middle of the page. In practice, banishing also chops off the now-trailing transformation $c3$@⟨⟩ because it is already known to interfere with the delta; we show it in the figure to make obvious the equivalence of the derivation history tails determined by commuting the last two transformations.

Having promoted c_4, we compute *PRESERVE*(f_2, $c4$@⟨1⟩, t_1@⟨2⟩), producing the result $c4$@⟨1⟩, t_1@⟨2⟩, again without resorting to use of transformation system control. Append t_1@⟨2⟩ to the new derivation history. We compute $f''_3 = c4@\langle 1\rangle(f_2)$, and discard f_2.

Step 4. Since c_3@⟨⟩ was dropped by banishment, we find that we can make no further progress toward an implementation using the old derivation history information; we consequently throw away any remaining part of the old derivation history. We compute $f'_3 = \delta_3(f''_3)$, discard f''_3, and then give f'_3 to the transformation system to complete the implementation. The transformation system is invoked to generate the new implementation f'_4 by applying the *cons-nil* simplification at an entirely new place; the additional transformation is appended to the new derivation history to form the completed, new derivation history.

The process terminates with the new implementation f'_4, the new derivation history appropriate for f'_4, and a new starting point, the saved f'_0. We are immediately ready to apply another functional maintenance delta.

The example demonstrates successful reuse of three of the four transformations from the original derivation history. Collected measurements suggest that derivation histories are often tens of thousands of steps long. Experience in porting software [3] indirectly indicates that most such steps can be preserved.

One gains a better appreciation of the utility of this process by comparing how this same functional change would occur in a more conventional software-engineering environment. We assume the maintaining organization has the implementation, f_4, and an informal document $\hat{f}_0$ approximating f_0. The derivation history has, as usual, been lost (assuming it ever existed). The customer, who only understands the abstract program $\hat{f}_0$, appears with an informal wish to change what the abstract program does, *i.e.*, an informal approximation $\hat{\delta}_0$ of δ_0. The maintainer's job is to produce f'_4 from the source code, f_4, given just the informal $\hat{f}_0$ and informal $\hat{\delta}_0$, with no derivation history. What is he or she to do? It is difficult to see how to do anything on a problem even as simple as this, and practical maintenance often happens on specifications 10,000 times as big. It is no great surprise that maintenance in conventional software-engineering environments is a difficult task.

Rep. ISI/RR-81-99, which contains the appendices not included in the ACM version.

CR Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic Programming; D.2.2 [**Software Engineering**]: Tools and Techniques—*Computer-aided software engineering* (*CASE*); D.2.7 [**Software Engineering**]: Distribution and Maintenance; D.2.10 [**Software Engineering**]: Design; D.2.m [**Software Engineering**]: Miscellaneous—*Reusable software;* I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program modification, Program synthesis, Program transformation;* I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search—*Heuristic methods, plan execution, formation and generation;* J.6 [**Computer-Aided Engineering**]: *Computer-aided design;* K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software development, software maintenance*

General Terms: Design, Documentation, Theory

Additional Key Words and Phrases: Formal CASE, design, maintenance, reuse, transformation

About the Author:
IRA D. BAXTER is a research scientist with the Schlumberger Laboratory for Computer Science in Austin, Tex. His research interests include transformational software implementation and parallel computing. **Author's Present Address:** Schumberger Laboratory for Computer Science, P.O. Box 200015, Austin, TX 78720-0015, baxter@slcs.slb.com.

This work has been supported by the Microelectronics and Computer Corporation/Software Technology Program summer intern program, National Science Foundation CER grant CCR-8521398, and California Micro grants #85-131, #86-017, and #87-055 in conjunction with Alcoa, Inc, and the Software Productivity Consortium.