UNIVERSITY OF CALIFORNIA

IRVINE

Transformational Maintenance by Reuse of Design Histories

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Ira David Baxter

1990

Dedication

To my wife, Linda, for her incredible patience,
to Ken Simms, who always liked Automatic Programming,
and to my friends and colleagues at UCI

Contents

List of Figures

Acknowledgements

It is impossible to acknowledge appropriately all the support, encouragement, and ideas I have received over the years. I cannot thank enough those who have played major roles:

Dr. Peter Freeman, my advisor, who provided the right intellectual atmosphere in which to work, and allowed me so much freedom of direction,

Dr. David Rector, for his patience with my mathematical ignorance and his interest in my work,

Dr. Dennis Kibler, for his initial encouragement to enter the PhD program, and continued interest over the years,

My friends and colleagues of the UC Irvine Advanced Software Engineering Project, Guillermo Arango, Julio Leite, Chris Pidgeon, Ruben Prieto-Diaz, Veikko Seppanen, and Yellamraju V. Srinivas for the time we spent growing intellectually together, and the endless improvements they made to my half-baked schemes,

My friend Dr. Jim Neighbors, whose Draco system was such a fundamental inspiration, and whose insights were always very sharp,

My friend and colleague, Dr. Ted Biggerstaff, for allowing me to explore my ideas at MCC,

Rick Gros, Bill Morita, and Ken Simms, for their friendship over the years,

My parents, for suffering a perpetual student,

My wife Linda, for living in uncertainty about the future for so long.

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

Curriculum Vitae

1952	Born in Alhambra, California
1973	Bachelor Degree in Information and Computer Science, University of California at Irvine
1982	M.S. in Information and Computer Science, University of California, Irvine
1990	Ph.D. in Information and Computer Science, University of California, Irvine Dissertation: *Transformational Maintenance by Reuse of Design Histories*

Abstract of the Dissertation

Transformational Maintenance by Reuse of Design Histories

by
Ira David Baxter
Doctor of Philosophy in Information and Computer Science
University of California, Irvine, 1990
Professor Peter Freeman, Chair

Keywords: software, maintenance, evolution, reuse, replay, history, design, formal, transformation, implementation.

This thesis provides theory and procedures for modifying software artifacts implemented by a formal transformation process. Installing modifications requires knowing not only what transformations were applied (a derivation history) to construct the artifact, but also why the application sequence ensures that the artifact meets its specification. The derivation history and the justification are collectively called a design history. A Design Maintenance System (DMS), when provided with a formal change called a maintenance delta, revises a design history to guide construction of a new artifact. A DMS can be used to integrate a stream of deltas into a history, providing implementations as a side effect, leading to an incremental-evolution model for software construction.

We provide a broadly applicable formal model of transformation systems in which specifications are performance predicates, subsuming the functional specifications which are traditional for transformation systems. Such performance predicates provide vocabulary used in the design history to describe the effect of applying sets of transformations.

A nonprocedural, performance-goal-oriented Transformation Control Language (TCL) is defined to control navigation of the design space for a transformation system. Recording the execution of a TCL metaprogram directly provides a design history.

A complete classification of, and representation for, the set of possible maintenance deltas is given in terms of the inputs defined by the transformation system model. Such deltas include not only specification changes, but also changes to implementation support technologies. Delta integration procedures for revising derivation histories given functional or support technology deltas are provided, based on rearranging the order of transformations in the design space. Building on these operations, integration procedures that revise the design history for each type of delta are described. An agenda-oriented TCL execution process dovetails smoothly with the integration procedures.

Our DMS is compared to a number of other maintenance systems. By using an explicit delta and verified commutativity, our DMS often reuses transformations correctly when others fail.

Practical Tips for Maintainers

For a Friend Assigned to a Maintenance Group

by David H. H. Diamond

The fellow who designed it
Is working far away;
The spec's not been updated
For many a livelong day.
The guy who implemented it is
Promoted up the line;
And some of the enhancements
Didn't match to the design.
They haven't kept the flowcharts,
The manual's a mess,
And most of what you need to know,
You'll simply have to guess.

We do not know the reason,
Why the bugs pour in like rain,
But don't just stand here gaping!
Get out there and MAINTAIN!

Chapter 1
Transformational Maintenance

Chapter summary. Software maintenance consumes the majority of energy currently expended on software systems. This thesis describes methods for integrating changes, defined formally as *maintenance deltas*, into an implementation, given a previous implementation constructed via a formal software construction scheme, and given the design decisions that drove the previous implementation. We summarize the context and focus of this work, and outline research assumptions and questions for further reference. The chapter closes with a summary of results and an outline of the rest of the dissertation.

1.1 Introduction

Software maintenance consumes a large fraction of the lifecycle costs of software systems [Bro75, CSM+79, LS80, Boe81, Gui83, RPTU84]. Decreasing the cost of that fraction of the lifecycle is naturally the first place to look in decreasing overall lifecycle costs. It appears unlikely that the demand for change will decrease, leaving only the hope of decreasing the actual cost of performing maintenance. Remarkably, there has been less research effort in this area than one would expect [Sch87, HH88] considering the apparent payoff.

Maintenance is fundamentally concerned with change to existing artifacts[1]. Implementation is about constructing new artifacts. This thesis describes an approach to software construction that we call *Incremental Evolution*. This approach blurs the traditional software waterfall lifecycle phases of implementation and maintenance. The idea is that software construction and maintenance would be better considered as a process of integrating a stream of changes into an artifact and its supporting construction technologies to produce a stream of software versions with updated support

[1]While this thesis is mostly about software artifacts, we see little difference between the construction of software and the construction of (blueprints for) other types of artifacts. We use artifact in the general sense.

technologies, allowing the process to be cycled indefinitely (Figure 1.1[2]). Artifacts would then be built by a *continuous* design process [Mar86]; in our version, the resulting artifacts would be compared against desires to produce deltas used as feedback (Figure 1.5).

DEFINITION 1.1: *Incremental Evolution.* Any artifact construction process that generates a stream of artifacts by integrating a series of changes, to an artifact specification and/or its supporting construction technologies, into an implementation. □

Managing a software process without tools can be difficult. We describe a class of tool for managing incremental evolution: a *Design Maintenance System* (Figure 1.8). Such systems combine *maintenance deltas*, using a theory of *delta integration*, with formal design information consisting of a program *specification*, its *implementation*, and a *design* to produce a *revised implementation* and a *revised design*. The delta integration procedures determine what parts of the design can be reused directly, how to construct new parts of the design, and how to repair those parts which will be only partially retained. We call such a system a Design Maintenance System because the emphasis is on maintaining the design, with the implementation obtained as a by-product rather than being the only product. This differentiates our approach from conventional maintenance methods which usually end up maintaining just the implementation. We need the design to decide what can be reused in the face of a delta. We maintain the design because that is more efficient than regenerating it on demand.

DEFINITION 1.2: *Design Maintenance System.* A set of tools that revises an artifact design (and its corresponding implementation) according to formal maintenance deltas defined by changes to inputs of the artifact construction process. □

Automation requires formality. We build a Design Maintenance System on top of a *Transformation System* to provide the necessary base formality. Using a transformation system provides us with a formal model of a software construction process. This forces all aspects of specification, design, and lifecycle issues to appear as formal entities of some sort, thereby making it possible, at least in theory, to capture these entities. Implementing delta integration routines in the context of transformation systems produces what we call transformational maintenance.

DEFINITION 1.3: *Transformational Maintenance.* Revision of an artifact carried out in the context of transformation systems. □

Implementing altered specifications is likely to be more efficient if one reuses relevant design decisions made in previous implementations. For a transformation

[2]Throughout this thesis, diagrams composed solely of boxes and arrows are SADT diagrams [Ros77, Fai85, MM88] unless otherwise noted.

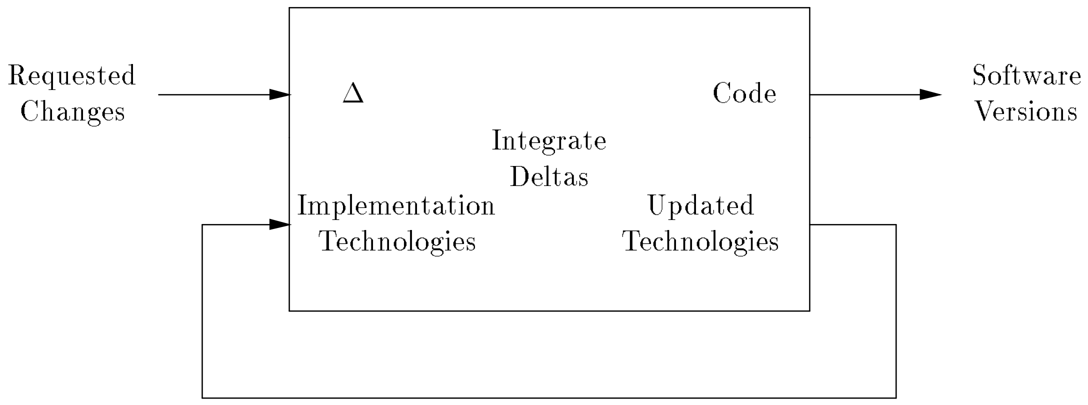


Figure 1.1: Incremental Evolution by Integrating Deltas

system, such design decisions can be captured in a structure we call a *design history*. Thus efficient transformational maintenance is made possible by the reuse of a design history.

This thesis provides definitions, mechanisms, and an architecture for a Design Maintenance System, built on top of transformation systems, to show that Incremental Evolution is possible.

Our work focuses on reducing the cost of software over its life by making maintenance more effective. We avoid the high cost of rediscovery of design information by choosing a formal software construction method based on a transformation system and capturing that design information during software construction. Captured design information is compared to formal changes, and obsolete design decisions are removed. A new implementation is then derived from the remaining design information.

The rest of this chapter provides a brief overview of the problem of design maintenance for transformation systems and our particular solution.

1.2 Problem Context

We start by considering a number of lifecycle models to establish a motivation for incremental evolution. We then move on to consider design reuse as a means for enhancing the artifact maintenance process.

1.2.1 Software Lifecycle Models

We will briefly discuss several lifecycle models to establish a motivation for the notion of Incremental Evolution:

- Conventional
- Transformational
- Incremental Specification
- Incremental Evolution

Conventional Lifecycles: A *conventional software construction* lifecycle is shown in Figure 1.2. Somehow, informal requirements are converted into some informal specification of the desired program, and the program is constructed manually. The result is compared to the desires, and informal specification of the errors are somehow converted into changes in the code. A benefit of the conventional approach is that the implementation, although expensive, only has to be constructed once; changes are added incrementally by very smart agents called programmers. The problem is that it is difficult and therefore costly for those agents to determine how to install a change.

Transformational Implementation: The *transformational implementation* approach to software construction [PS83, Agr86, Fea86] constructs an implementation in a conventional programming language by repeated application of optimizing *transforms* to a formal program specification constructed from informal requirements (Figure 1.3). Balzer [Bal85a] suggests that the transformational paradigm could significantly aid the maintenance process, by focusing maintenance on the abstract specification rather than the code. The formal specification or its implementation is validated against the requirements, and any noted inconsistencies cause the formal specification to be changed appropriately. After each change, the transformation system is presented with the entire revised specification and derives a new implementation from scratch. The benefit of this approach is that changes are made directly to

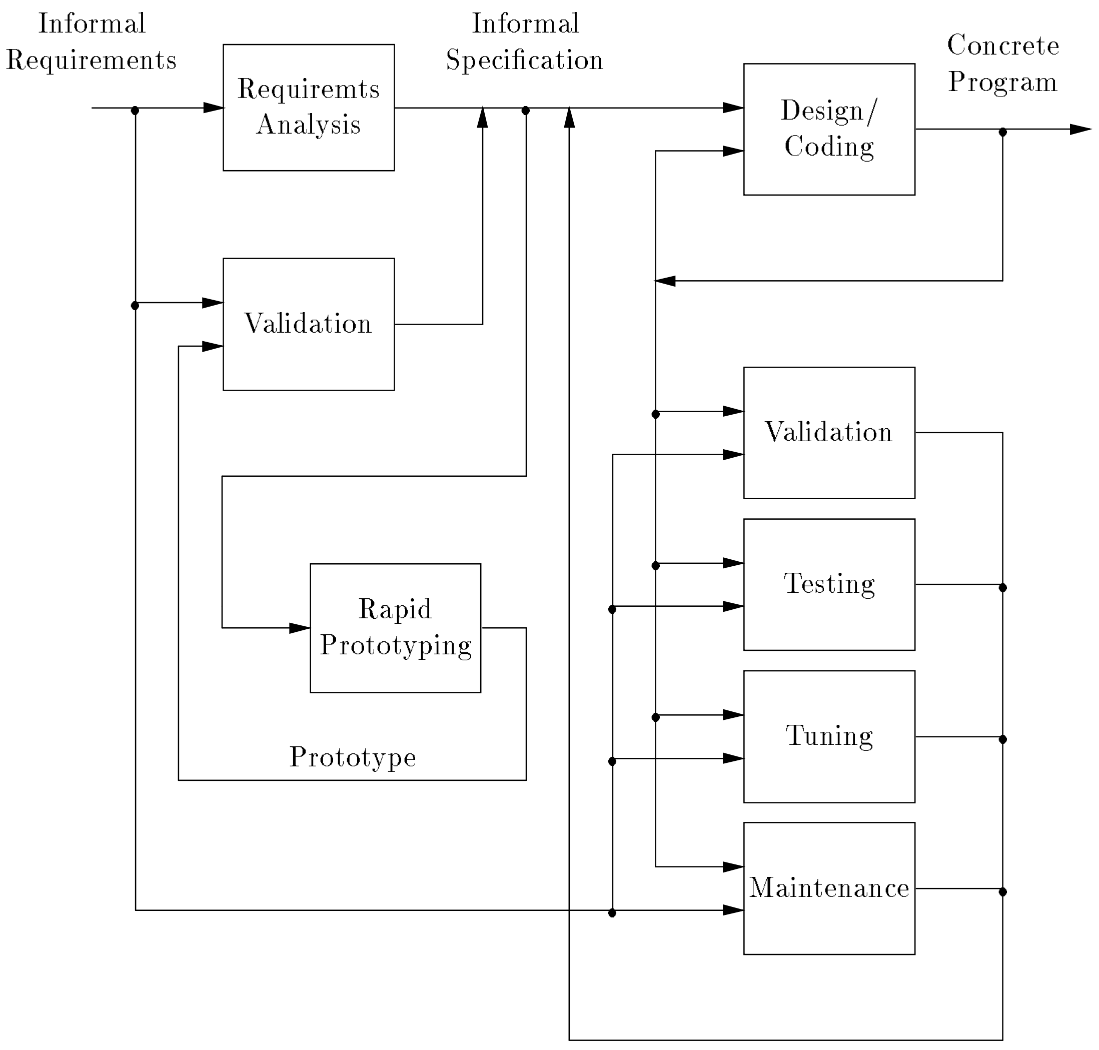


Figure 1.2: Conventional Software Construction

the specification rather than the code, and are presumably easier to make and understand because they are not buried under a mountain of programming optimizations. We call such an approach *transformational maintenance*, as maintenance takes place in the context of a transformation system.

While such an approach technically solves most of the maintenance problem, it does so by changing it into the transformational implementation problem. Initial transformational implementations are expensive to obtain and we would expect that the cost of each full re-implementation will be approximately the same as the cost of the initial implementation (if the changes are small). Such a naive approach to re-implementation would leave transformational maintenance impractical. One possible way to speed the re-implementation process to reuse design information from preceding implementations of the same artifact to avoid the cost of making all the design decisions again. To do this, one must somehow be able to decide what design information is relevant for the revised specification. With a pure transformational implementation model, the transformation system sees only a new specification, and at best possesses only an old design. How is it to decide what part of the design is relevant to the new specification? We will return to this topic shortly.

Incremental Specification: Feather [Fea84] suggests that specifications are not constructed en toto; rather, that they may evolve by elaboration of simpler specifications. The idea is that the informal requirements describe a number of differing aspects, and that one constructs a formal specification by starting with a simple formal specification that characterizes one of those aspects, and then changes that specification to accommodate the rest of the informal requirements as convenient. Using formal *evolution transforms* to modify a formal specification, Feather and Johnson [JF90, Fea89a] pursue the notion of *incremental specification*, assuming the lifecycle model in Figure 1.4. The monolithic revised specifications are fed to a transformation system for implementation. While the incremental specification approach may ease the problem of acquiring and maintaining the specification, it does not particularly help in obtaining an implementation; it has exactly the same problem acquiring revised implementations as transformational implementation. The incremental specification paradigm qualifies as a limited type of incremental evolution by our definition (it fails to address changes to the supporting construction technology), but it is not necessarily efficient.

Incremental Evolution: We envision carrying the incremental process to the extreme by integrating formal entities called *maintenance delta*s into the ultimate artifact (Figure 1.5) rather than just the specification. We call this process *incremental evolution*. The difference between incremental evolution and Balzer's scheme of reimplementing a modified specification is how far into the implementation process the

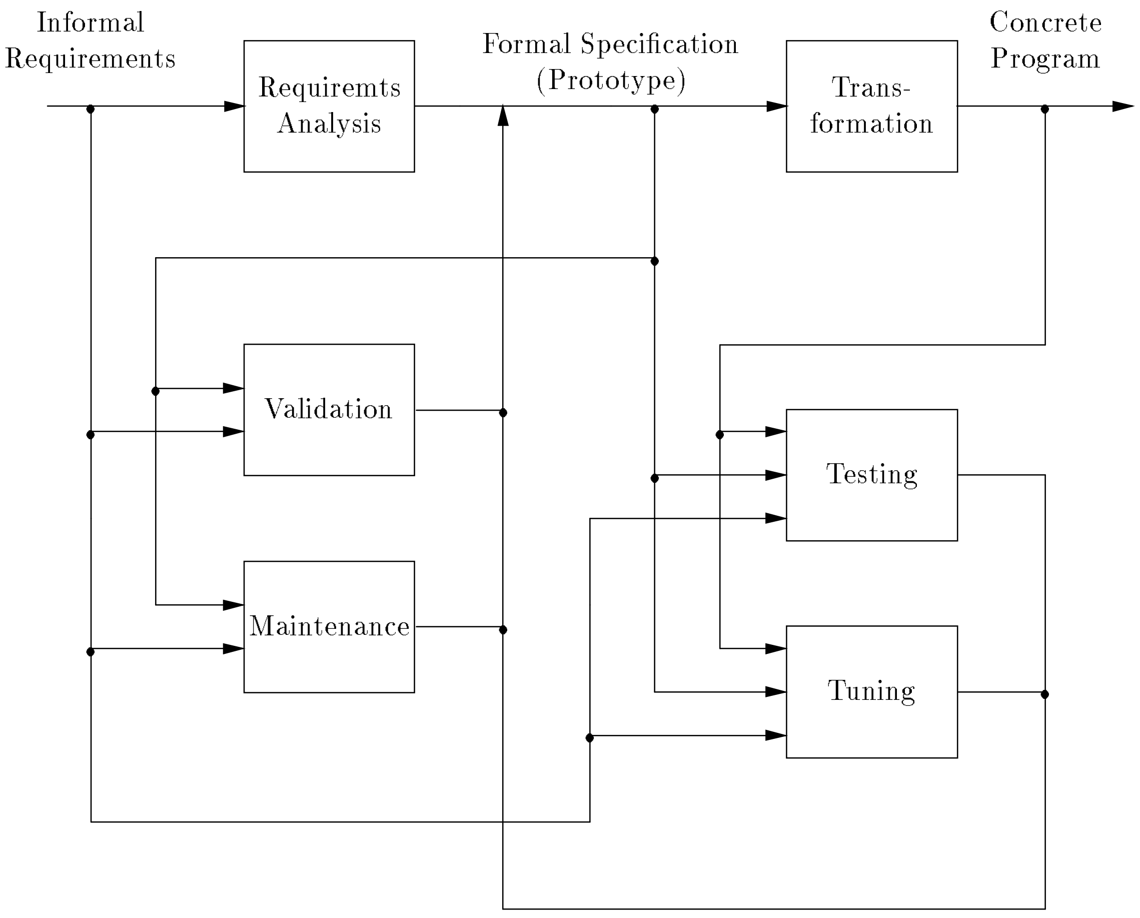


Figure 1.3: Simple Transformational Software Construction/Maintenance

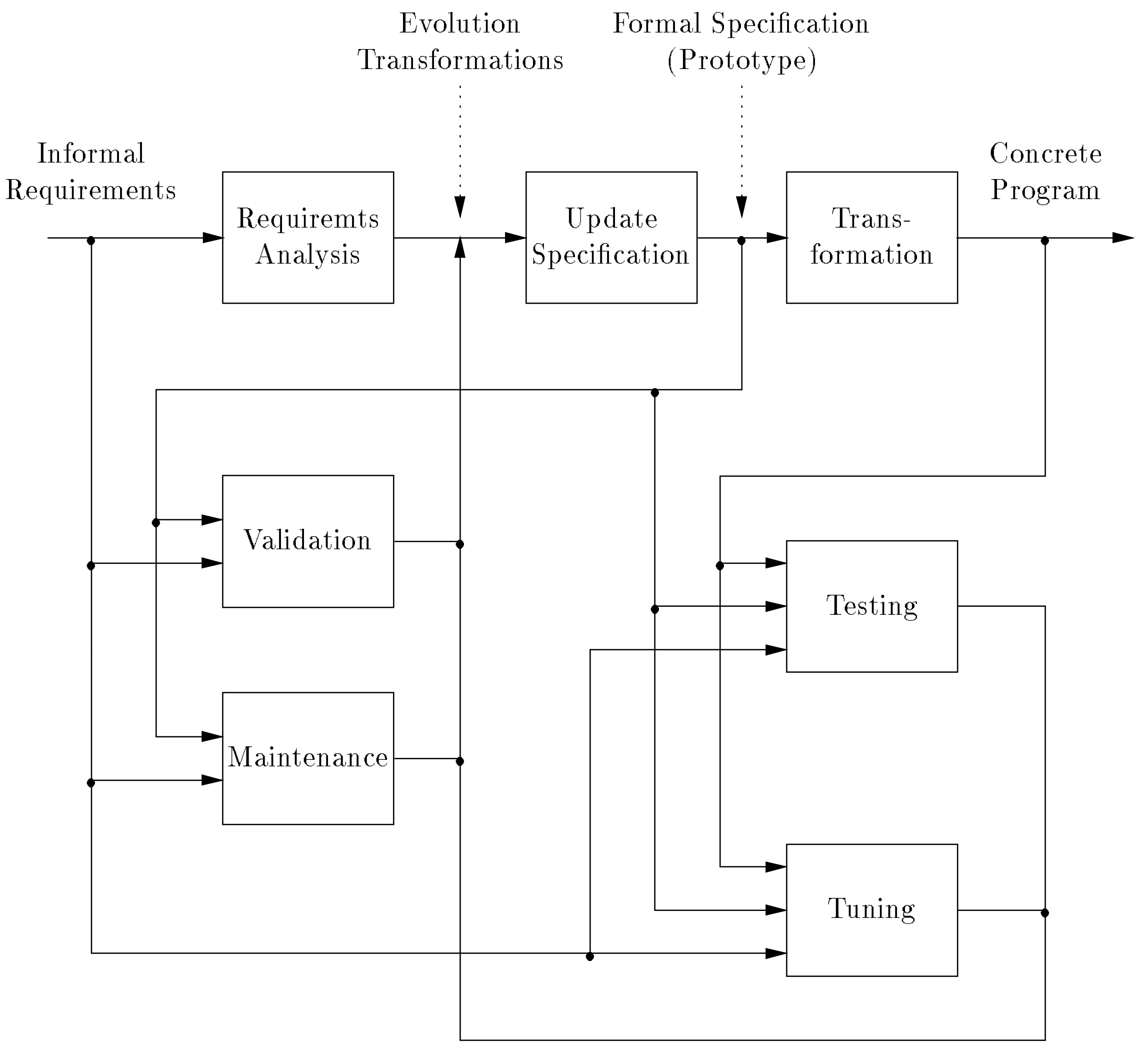


Figure 1.4: Incremental Specification followed by Transformation

deltas survive. In Balzer's scheme, the deltas disappear once the specification has been changed. For incremental evolution, we expect the deltas to directly guide changes to the implementation.

We generalize the idea of evolution transforms by allowing two sources of formal deltas:

- customer desires compared to the current version of the artifact (requirements analysis and validation)
- new or changing support technology used to specify or implement the artifact (domain engineering [Ara88])

These two sources of change remain active regardless of artificial divisions of lifecycle phases into implementation and maintenance. The deltas must be with respect to something; we assume that deltas induced by failure to meet requirements are applied to some sort of a specification, and that deltas to support technology are applied to reusable libraries of mechanisms available to the implementation process. We remark that commitment to the notion of specification does not, however, necessarily commit an incremental evolution scheme to any underlying transformation system, although we will pursue that approach in this thesis.

The requirements analysis process produces a stream of deltas as in the incremental specification approach; the initial stream provides the basis for an initial specification. The validation, testing and tuning processes are expected to compare the informal, desired effects of executing the current version of a software artifact against its actual effect, and suggest changes that need to be made to the specification; how this is accomplished is outside the scope of this thesis. The domain engineering process is expected to determine the *support technologies* (implementation mechanisms and successful implementation criteria based on understanding of the problem domain and the possible implementation technologies) and suggest corrections to the support technologies available; this activity is similarly outside the scope of this thesis. Both processes affect the design and implementation of desired programs.

The advantages of an incremental evolution paradigm are:

- The unification of implementation and maintenance
- Incremental specification/maintenance
- Support for a dynamic support technology base
- Potential for formal documentation, literally, of the changes made

Such an incremental evolution system will be practical only if we solve the problems of generating formal maintenance deltas, and integrating those deltas efficiently

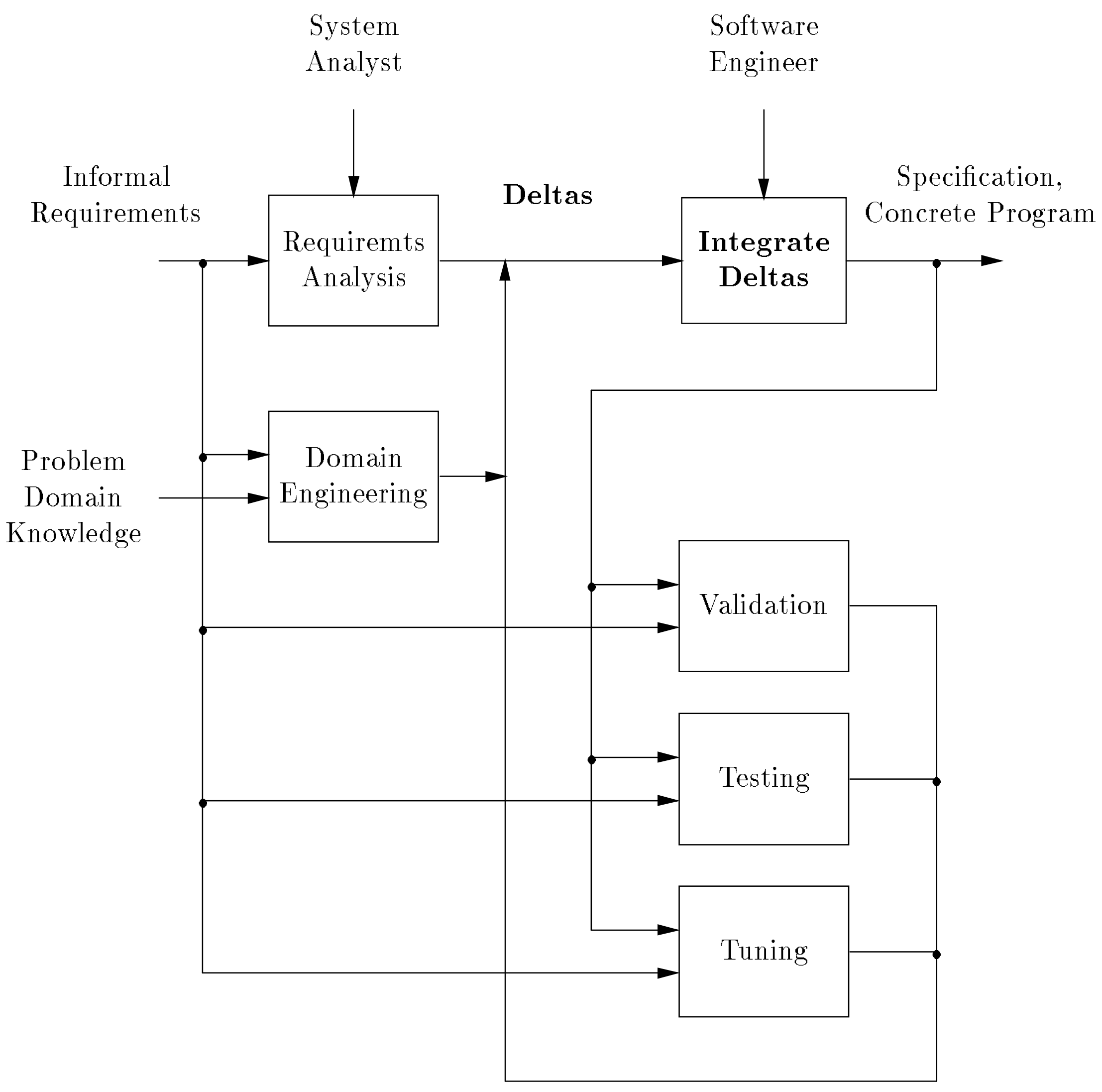


Figure 1.5: Incremental Evolution by Delta Integration

into an existing implementation. This thesis concentrates on those aspects, shown in bold in Figure 1.5, implemented in a manner to make transformational maintenance much more efficient than simple transformational re-implementation of a changed specification.

1.2.2 Effective Maintenance Needs the Design

An essential part of a program for maintenance purposes is the design, and it is usually lost, abandoned, incorrect or inaccessible. Without the design, understanding and modifying a program is nearly impossible [Sol87, Nin89]. Sneed [Sne89] describes 4 major software engineering efforts that ended in disaster precisely because trying to maintain the design was perceived as impractical; consequently the design was abandoned and the projects spiralled into chaos.

One can attempt to retrieve the lost design information by inspecting the code; Chikofsky [CC90] provides a brief overview of this area. There are number of research efforts whose purpose is to recover lost design information, [ABFP86, Wil87, Wat88, Big89b, Nin89, BCC89, SJ88]. Recently, an entire industry has appeared, called “re-engineering” [CS89, RD88] which provides tools for reorganizing existing software systems based on recovered low level design information. But we think those approaches are mis-directed: the design should never have been lost in the first place.

The reason that designs are lost is partly due to their informality in conventional design processes, which makes them difficult to record, and partly due to short-term organizational pressures to complete a product rather than document its structure. One approach to design capture and retention is to combat informality by using a formal software construction method, and adding automation to that construction method to capture the design as the implementation occurs, so that completing the product ensures the design is captured. This suggests the use of a transformation system to generate formal design information, and finding ways of storing that design information.

1.2.3 Design Information

To capture design information about an artifact, we must first know what design information is. We summarize the notion of designing with Pidgeon's remark [Pid90]

To design is to decide.

Surveys of design can be found there and in [Mos85b].

There are numerous types of design information, of which the following are critical for our purposes:

- Design knowledge: understanding of desired properties and construction techniques for classes of artifacts
- The specification of a particular artifact
- Design decisions (traditionally called *the* design) for that artifact
- Design rationale: a demonstration that the design decisions produce a satisfactory artifact

Design knowledge provides general information needed to describe and implement similar artifacts. The description knowledge provides vocabulary and metrics with which one can characterize or measure interesting properties of an artifact or its component structures. Implementation knowledge provides alternative implementation possibilities as well as information useful in determining tradeoffs between choices.

A specification gives a description of satisfactory artifacts. Without a specification, a design would have no purpose.

Design decisions describe alternatives chosen while implementing the artifact. We assume that any artifact design process involves a series of decisions about measurable properties of the final artifact, especially those which determine the component structures of that artifact. We insist that any reasonable artifact have only consistent properties. Following Pidgeon [Pid90], we define:

DEFINITION 1.4: *Design Choice.* An unresolved choice of value for an artifact property, represented by the property name of interest and a set of mutually exclusive possible alternatives. □

The alternatives may list different possible implementations of portions of the artifact. It is the purpose of the design process to make such choices in a way that the resulting artifact satisfies the specification, although a design may be inconsistent with a specification[3]. Design choices are resolved as decisions:

DEFINITION 1.5: *Design Decision.* Specification of a constraint on a design choice determining a subset of alternatives from which a selection will be made. □

[3]A specification is an artifact produced by design from customer requirements, but that is design at another level.

One or many decisions related to the same design choice may be required before a unique alternative is determined, depending on the strength of the constraints; it is even possible to overconstrain a design choice so that the set of selectable alternatives is empty. The minimal information about a design decision is a:

DEFINITION 1.6: *Design Selection.* A property name and a distinguished alternative defining its value. □

The classic blueprint is a representation of a set of design selections; it contains no alternatives. Design information should ideally include all design decisions made during the implementation process, or at least enough design selections to uniquely determine the outcome of any possible design choice. This uniquely determines the implemented artifact. Classes of satisfactory artifacts are defined by sets of design selections that uniquely determine properties of the artifact relevant to the specification.

The design rationale explains how the various design selections satisfy the specification.

DEFINITION 1.7: *Design Rationale.* An information structure that justifies how the implementation (consequences of the design selections) satisfies its specification. □

Technically, a design rationale is not required because one should be able to deduce satisfaction from the design decisions alone. In practice, however, the cost of determining just how the specification is satisfied is so complex that a design rationale provides an immense aid to those that would understand the design.

Collectively, the design rationale, the design decisions, and the specification make up design information (see Figure 1.7) specific to a particular artifact. Design knowledge is needed to generate the design rationale.

1.2.4 Design Reuse

For conventional software construction processes, considerable research has been focused on reuse of code. Reusing other software process products, such as design information, has been proposed as a method for lowering implementation costs [Fre80] of new artifacts.

Arango [Ara88] models reuse roughly as follows:

1. Identification of a class of interesting reusable information
2. Selection of a method for reusing that type of information
3. Capture of appropriate reusable information for the method
4. Discovery of a situation for which the method has promise
5. Application of the reuse method with the captured information

Each type of design information can be reused in possibly different ways. Captured design knowledge can be used to produce and answer queries about a desired artifact, or to propose new implementations. A specification can be reused as a starting point for similar artifacts. Design choices can be reused to regenerate alternatives, while design selections can be reused only to determine outcomes of such choices. Design rationales can be reused to check the validity of design choices.

Design knowledge related to descriptions is implicitly reused as vocabulary for specifications. Most transformation systems reuse design implementation knowledge in the form of a transformation library. Higher level design implementation knowledge has been reused in the form of implicit control knowledge such as the refinement phases of REFINE compiler [Gol89]. and TAMPR transformation system [BM84], as well as in the form of explicit procedural metaprograms such as PADDLE ([Wil83]) and Feather's nonprocedural notion of CONTEXT [Fea79].

1.2.5 Design Replay

A form of design reuse of great interest is that of design replay, which reuses design decisions to implement a new artifact given a slightly changed specification[4].

A *design history* is a (possibly partially ordered) sequence of actions taken by a designer during a design process. Such actions can include analysis of consequences as well as making design decisions. A design history contains information that was probably hard to obtain, and therefore has potential reuse value.

If one has a design history, then a particular reuse method is to *replay* it by applying its component actions in the specified order in the context of a changed specification. Problems occur when actions reference entities that are irrelevant to the new specification, when replayed design selections ultimately fail to satisfy the new specification, and when the existing actions fail to address new needs imposed by the new specification. What makes a replay method interesting is how it addresses these problems.

Capturing a design history requires that design actions be representable. A way of ensuring that at least some aspects of the design process are capturable is to implement that process as a transformation system. In a transformation system context, one can treat the sequence of applications of particular transformations as design selections and capture just the transformation sequence. This kind of design

[4]A situation in which one has a library of designs and an absolutely fresh specification can be converted into this case by choosing the design library element whose artifact specification most closely matches the new specification. Since we are interested in maintenance, we do not consider the matching problem further.

history is called a *derivation history*. For replay, one reapplies these transformations in their original sequence.

Mostow [Mos85c, Mos86] provides a list of reasons why design derivations are hard to replay. One of difficulties is that goal information is not present in a derivation history; there is no design rationale. A second problem is that one can only trivially replay the derivation up to the point where a change is required; replaying from this point on is *blocked* because the original conditions no longer apply.

A key insight is that the decisions, made beyond the point where a change is required in a design history, are not necessarily invalidated by the change.[5] We argue, in fact, that the very large scale of any transformational implementation ensures that many of the decisions beyond the change point are, in fact, applicable. Effective reuse of a drvh requires reuse of decisions *beyond* the blocking point. Even those decisions which no longer apply directly may contain some useful information extractable by *analogy* [Car85]. A consequence of these insights is that methods for replay hardly ever use the trivial method without some interesting variations.

A number of transformation systems exist that do some form of design history replay for software construction (PADDLE [Wil83], Φ-NIX [Bar89], finite difference synthesizer [DKMW89], REFINE [Gol89], IDEA [Lub89], PDS [CHT81]), DIOGENES [MF89a], [MF89b]). Design replay has also been used for hardware design (REDESIGN [SM84, SM85], ARGO [HA87], BOGART [MB87], although we don't think the problems are fundamentally different. We sketch a few of these systems here.

Partsch [PS83] describes replay in Cheatham's system as repeated application of previously used refinement (transform) rules until none is applicable. The implication is that the transform sequence is also lost. This is undesirable as many optimizing transformations in an implementation require other *conditioning* transformations be applied first [Fic82, Fic85], so loss of sequencing must imply some loss of optimizations. Also, different early sequencing will lead to implementations radically different than the original, and this is surely not desired. We see that order is important and must be preserved.

Goldberg's system [Gol89] records the exact sequence of the applied set of transformations. The point where each replayed transformation is applied is adjusted in an attempt to account for the change in the specification. This system appears to block when a replayed transformation fails to be applicable.

[5]Baxter [ABFP86] suggests that the actual historical order is not always relevant, and, in fact, hinders the ability to maximize the re-use of an established design.

The BOGART system [MB87] constrains a derivation history to consist of recursive decompositions of a functional specification into successively more primitive functions. The derivation history actually takes the form of a tree, with the root being the initial specification, and the set of branches emanating from a node specify the decomposition of that node into sub-nodes. The tree-structure provides a partial order on the application sequence of the transformations. Such histories can be replayed by replaying each branch until a blocking point is reached along that branch.

The PADDLE [Wil83] system, rather than capturing a derivation history, instead captures a derivation history generator, called a *program development* as a set of plans for implementation. This language shares some similarities to hierarchical planning systems [Sac77]. A plan may be achieved by any of several subplans, providing a method of handling decisions which no longer apply. Replay consists of re-executing the generator. Even so, PADDLE still blocks at the first decision it cannot handle.

1.2.6 Problem Context Summary

We are interested in implementing tools to support an Incremental Evolution lifecycle model. Doing so seems to require the reuse of design information captured for a predecessor implementation. Such design information comes in several forms, including design knowledge, the specification, a design history containing the design decisions, and a rationale for the design. Such design information can be captured in the form of a design history, and reused when the specification changes. A major obstacle to such reuse is taking into account the effect of the specification change.

1.3 Transformational Maintenance by Reuse of Design Histories

Incremental Evolution requires that changes be integrated into implementations. If we choose a transformational implementation base model, then we can formalize the changes as maintenance deltas. Application of such formal maintenance deltas to the specification and supporting infrastructure of the transformation system followed by reimplementing provides us with an inefficient form of transformational maintenance.

The formal transformation system also allows much of the design information involved to be formalized. Such formalization presents us with the possibility of providing procedures for integrating the maintenance deltas into captured design histories. We are reusing the design to produce a new design. A revised implementation can then be extracted from the updated design. Assuming relatively small changes to the design, updating and extracting should be cheaper than having to reimplement our specification from scratch.

A Design Maintenance System can be regarded as an efficiency enhancement to the batch transformational maintenance model. Incremental Evolution implemented via a Design Maintenance System avoids the repeated cost of rederivation by reusing the design information (including the previous specification) collected in the previous implementation cycle. It determines what part of the design information is still relevant by comparing it to maintenance deltas; design information that is no longer relevant is discarded, and a repair to the design that covers the delta is generated. The effect over time is that of *integrating deltas* to achieve the desired implementation.

Existing design replay systems cannot express or use performance specifications, and assume a fixed implementation technology base. We explicitly address performance specifications, both in a transformational implementation context, and in a replay context. Most replay systems capture and replay of just a derivation history; we insist on the use of the entire design history, as it provides structuring for the derivation history, thereby allowing a kind of damage control. Our design history also suggests alternatives for portions of the derivation history which become invalid because of the desired change. Our methods for delta integration preserve transformations following blocking point in a derivation history to maximize reuse of the history.

1. It is possible and practical to specify, formally, a software system.
2. Such specifications can be formally converted into implementations.
3. Specifications are more frequently modified than replaced when change is desired.
4. Revisions to specifications are not massive.
5. Statistically, small changes in specification lead to small changes in implementation.
6. There is an explicit relationship between specification and implementation called a "*design*".
7. All of the possible changes one could possibly wish to make to an artifact are expressible as formal changes.

Figure 1.6: Preconditions for Transformational Maintenance

1.4 Research Assumptions

Our research has several fundamental assumptions, listed in Figure 1.6. Reasonable justification for these assumptions can be found in everyday computing, including:

1. Many systems are specified informally, or more formally by coding them in conventional computer programming languages such as Fortran or Prolog; special languages for application generators are heavily used [Cle88].
2. Compilers [ASU86], application generators, and transformation systems [PS83, Agr86, Smi89]) convert "formal" specifications from a less executable form to a more executable form.
3. Most of the work invested in software is in maintenance, indicating that the original specification was worthwhile; rarely is a specification entirely replaced.
4. When requests for changes to software are made, the request itself is many times made in incremental terms with respect to the original specification ["add this function", "speed that up", "use this new hardware", etc.].
5. The amount of code changed in an application as a result of a change request is relatively small compared to the size of the implementation [LQ89, Sin83].
6. Designers convert specifications into programs; they rationalize each part of the program as serving some purpose with respect to the rest of the program or the specification, and thus a relationship between implementation and specification really does exist.

We think none of the above points are really controversial. Precondition (7.) does not have an everyday justification, so we will provide one in Chapter 6.

Since our emphasis is on maintenance, we assume the necessary prerequisites for a maintenance situation. In a transformational context, this consists of:

- the existence of a transformation system populated with transforms and control hueristics sufficient to transformationally implement some class of specifications.
- a specification G – the goals for a particular software artifact,
- an implementation f_G derived from the specification G
- a derivation history H consisting of the sequence of transformations applied to generate the implementation
- a design history D which justifies the individual transformations chosen.

For each desired change, the following must be true:

- The desired change can be explicitly described.
- The transformation system must be able to implement the modified specification in the absence of a design maintenance system.
- The expected cost of installing the change must be significantly less than simply re-implementing.

It is not obvious that every desired change is describable. We will show, however, that specification changes can be written down as formal expressions (*maintenance deltas*) under reasonable assumptions about formal program specifications.

1.5 Thesis Statement

Our long term objective is to develop practical tools for performing incremental evolution. Previous experience and research suggest that reuse of the current artifact's design is necessary for this to be effective. Since tools manipulate formal representations, we must use a formal software development process from which formal design information for an artifact can be extracted. We have chosen transformation systems to realize a formal software process.

It is our thesis that: **We can efficiently maintain software generated transformationally by integrating formal deltas into design histories.**

In support of this thesis, we provide definitions and mechanisms to guide installation of changes to a software artifact using design information captured during its transformational construction.

To accomplish this, we must:

- Define a representation for change requests
- Define the nature of the design information we will reuse
- Determine what part of the design information is reusable and how to reuse it based on a particular change request
- Provide a mechanism for completing the resulting partial design

The problem then becomes:

- How to capture an artifact specification, design, and implementation
- How to exhaustively define possible types of change
- How to define individual changes of each type
- How to integrate each type of change into the specification, design and implementation
- Defining an architecture combining the change integration components into a monolithic whole forming the Design Maintenance System.

The resulting solution should have the following properties:

- Always produces a revised, possibly partial, design consistent with the desired change, to allow repetitive cycling of the process
- Degrades gracefully into a purely constructive procedure when most or all of the previous design turns out to be unusable

Our intention is to lay the foundation for a Design Maintenance System.

1.6 Research Approach

To construct a Design Maintenance System, we need a basis software construction process. This process will serve to construct implementations from specifications when little or no design is present, and will help repair the reusable part of an existing design. We have chosen transformation systems to act as that basis.

Most of our solution flows indirectly from a formal model given for a transformation system. There are few such formal models, so we provide one. The notion of a *performance goal* turns out to be central to our characterization, and is virtually absent from other models. We also define a goal-oriented metaprogramming language for controlling the application of transformations in the form of a language called TCL.

Such a model determines all inputs, outputs and vocabulary used to control the operation of the transformation system. These definitions provide an exhaustive means for defining changes as formal maintenance deltas applied to these inputs in terms of the vocabulary. We define one type of maintenance delta for each possible input to the transformation system.

We capture design information in the form of a *design history* (see Figure 1.7). The actual historical sequence of transformations that convert an abstract program (satisfying the *functional* part of the specification) into a concrete implementation are recorded in a part of the design history called a *derivation history*, shown as a chain of circles. This information only shows how the abstract program was implemented, but it contains no design information in the sense of justifying *how* the implementation achieves the (rest of the performance) specification. These justifications are captured in the form of goal/plan decompositions shown as the tree in the figure. Each square box represents a performance goal, with the arcs leading downward providing the decomposition of that goal into subgoals or actions (in the form of applied transformations) whose composed effect achieves the goal; each subaction is then justifiable as achieving some higher level effect recorded in the design history. Design histories are generated effectively by tracing execution of a TCL metaprogram. The goal nodes also provide indexes back into a TCL metaprogram for use during plan repair. A partial design history is one in which some of the goal nodes do not have sons, so the entire performance specification is unsatisfied. The notion of design history is closely related to that of plan from the AI planning domain [CM85].

Given a maintenance delta, it is our desire to revise the design information, instantiated as a design history, to be consistent with that delta. We divide this problem into two parts, not necessarily sequential:

- revising the derivation history to be consistent with the delta
- revising the balance of the design history

A useful property of a derivation history is the *commutativity* of the many individual applied transformations: most can be exchanged without affecting the end result of the derivation history. We can consequently rearrange the derivation history for our convenience into two parts: a (possibly) reusable part, and a (definitely) reuseless part. Such rearrangement is controlled by the particular maintenance delta we desire to apply; transformations which interfere with the maintenance delta are *banished* into the reuseless portion. After rearrangement, simple truncation of the derivation history at the point of the first reuseless transformation retains the reusable part. We provide a number of procedures for rearranging the derivation history in the face of particular types of maintenance deltas.

The design history is revised by a *pruning* process which deletes goal nodes from the bottom towards the top when subgoals or transformations are discovered to

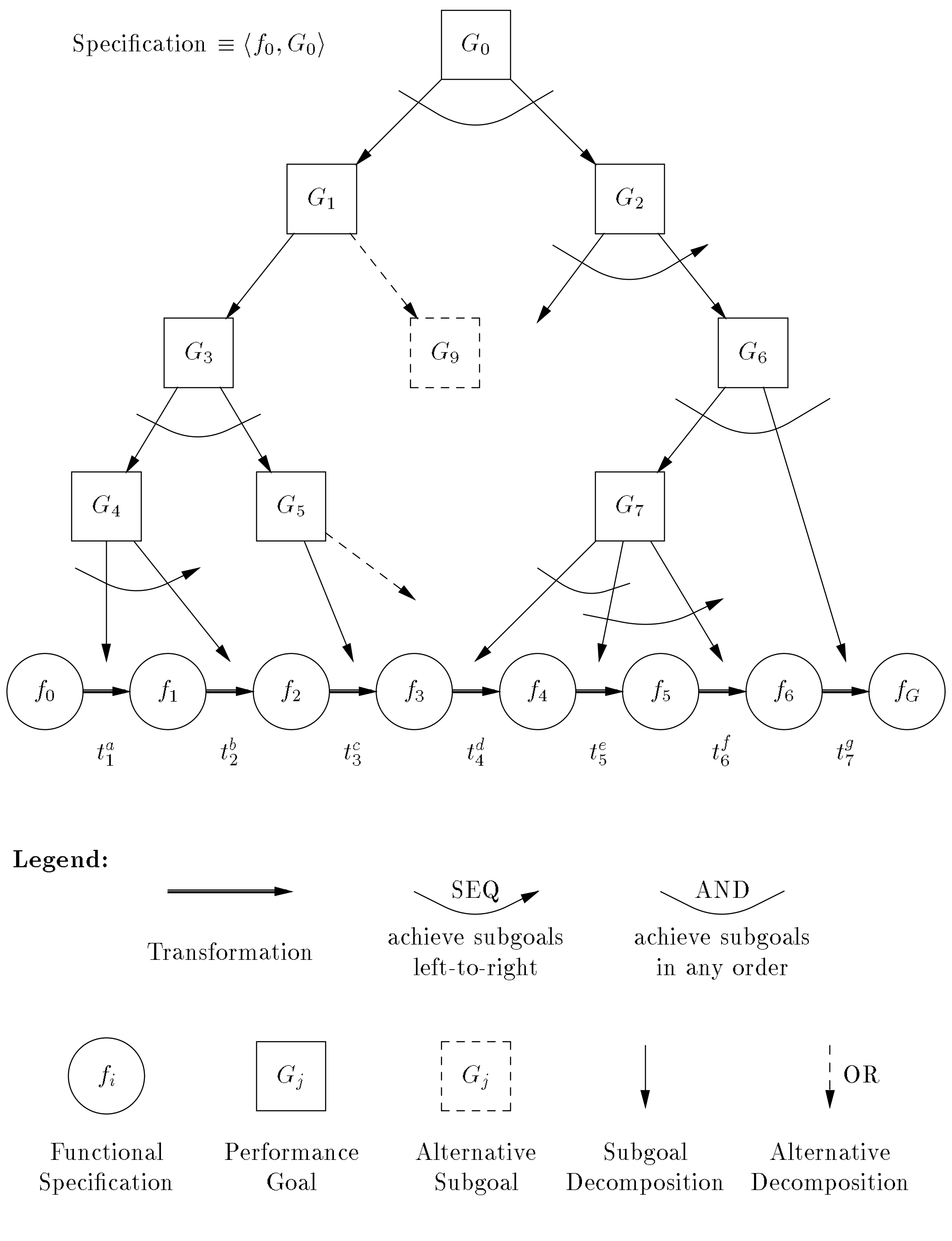


Figure 1.7: Transformational Design History including unfolded Design Plans

be no longer usable (for transformations, this occurs when the derivation history is truncated). A top-down revision occurs to handle change in performance goals. There is some interaction between derivation history and design history revision when a bad transformation is identified; all of its siblings according to the design history are also bad and must be banished.

Having obtained a partial design history, we repair it by turning it back over to the transformation system core of the Design Maintenance System. The transformation system can find ways to finish incomplete goals by using the indices stored with those goals to locate fragments of the original TCL metaprogram to re-execute. Since the transformation system can have its attention switched between portions of the original TCL metaprogram depending on the necessary repairs, execution of the TCL metaprogram is agenda-oriented rather than sequential as with most metaprogramming languages. This cleanly unifies initial design history construction with design history repair, so that only a single mechanism is necessary.

These individual components must be combined together to form a complete Design Maintenance System, which we discuss next.

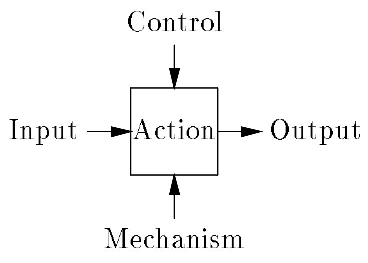
1.7 Design Maintenance System Overview

In this section, we provide a broad overview of the components of a transformational Design Maintenance System. SADT diagrams representing the major subsystems are sketched. We gloss over some of the detail because we have not yet defined our vocabulary; in following chapters, we will detail the procedures more carefully.

Delta-integration, the top level of a DMS, interfaces (Figure 1.8) to the requirements analysis, validation, and domain engineering processes (Figure 1.5). The delta-integration process must take the deltas produced by these processes, and revise the support technology, the design, and the implementation of the desired artifact. Often, a number of changes to various aspects (specifications, support technologies) come bundled as a composite delta; ideally, delta-integration would handle the composite delta in parallel rather than dealing with each aspect in a serial fashion. The feedback arcs should probably be implemented using some kind of database for long term storage; these databases are initially empty.

Delta integration requires revision of specification and support technology, as well as revising the design of the end artifact. We can see the composite delta split into its components and processed in Figure 1.9. Revising specifications and support

SADT diagram key:



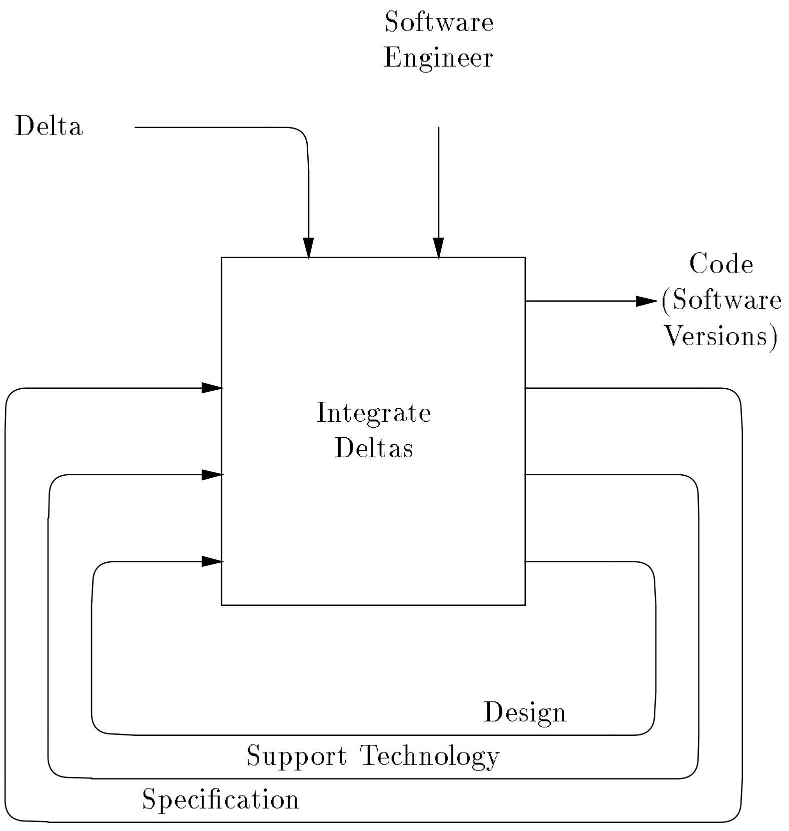


Figure 1.8: Design Maintenance System

technology is relatively straightforward, as these objects have relatively simple structure. Constructing a new implementation is accomplished by reusing parts of the design; this is accomplished by first pruning away those parts that are incompatible with the changes desired, and then repairing the pruned result. The repair mechanism is intended to complete the partial design and produce the final artifact, given the revised specification G' and some set of support technologies. It must be robust; it may be required to accept an empty design. In practice, it may produce a partial design and no artifact because the specifications are too constraining for the available support technologies. This is not serious; in fact, we expect this occur naturally during the process of implementing a large specification. This simply triggers the generation of a new delta to either shore up the support technologies, or weaken the specification. Because we do not emphasize actual production of an implementation, this is within our model.

We use a transformation system to generate and repair partial designs (Figure 1.10). The transformation system requires a specification $G' = \langle f'_0, G'_{rest} \rangle$ of an artifact to implement, design history D' describing partially how it is implemented, and support technologies (transforms (C), design methods (M), goal predicates (G), performance measuring functions (P), etc.) with which to control and carry off the implementation. In Chapter 3 we will provide definitions of these support technologies, and we will see how the specification splits into a program part to be transformed f'_0 and a termination predicate G_{rest}. The transformation system produces a resulting program $f_{G'}$ satisfying the specification G', as well as a design consistent with that implementation. The transformation system may actually change as opposed to merely augment the partial design in order to complete it; the partial design has had only the obviously incorrect parts pruned away. The control process for the transformation system must be agenda-oriented if we want repair to mesh naturally with construction. Chapter 4 describes a metaprogramming language, TCL, used to generate the design decisions, and Chapter 5 shows how to capture this design information as a design history.

Revising the support technology is conceptually straightforward, but has a number of cases. For each class of support technology used by the design repair process, that class needs to be updated according to changes defined by a composite delta $\delta_{support}$ (Figure 1.11). The composite support technology delta $\delta_{support}$ is split into component deltas, one for each support technology class. Each support technology class is revised by updating a corresponding database. The fact that some support technologies are built using others induces a consistency requirement on the components of a composite support delta. Details defining the support technology deltas and these revision procedures are provided in Chapter 6.

A program specification has a two part representation $\langle f_0, G_{rest} \rangle$, consisting of an abstract program and some additional performance constraints, determined by

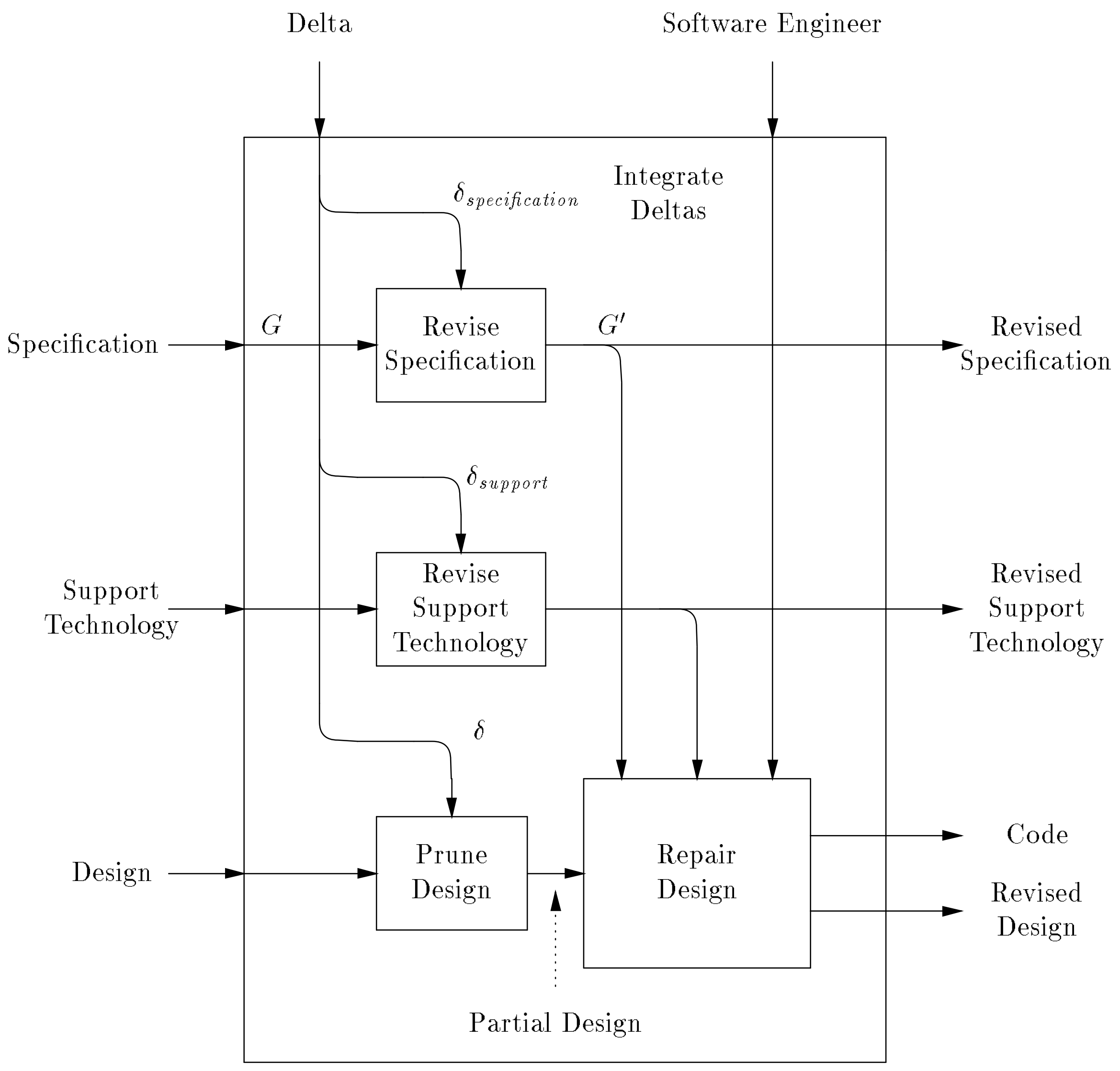


Figure 1.9: Integrating Deltas

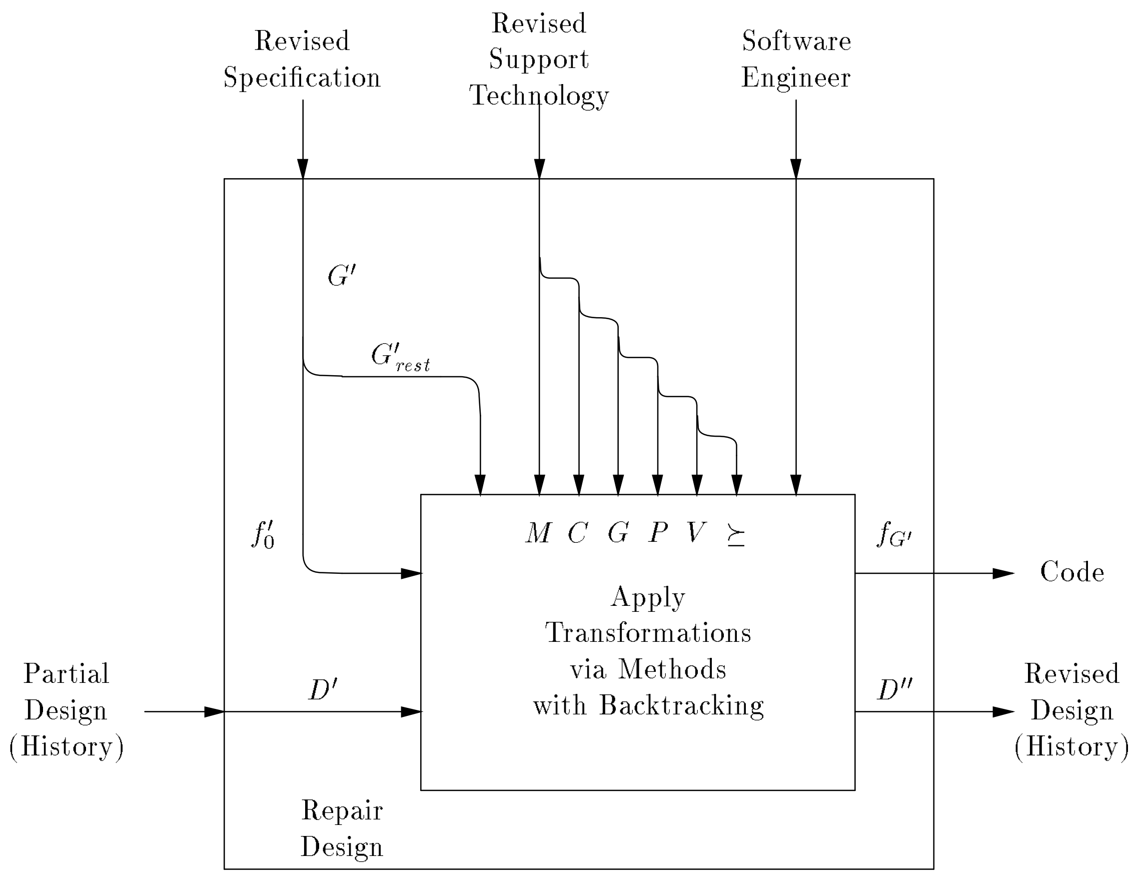


Figure 1.10: Implementing a Specification by Repairing a Partial Design

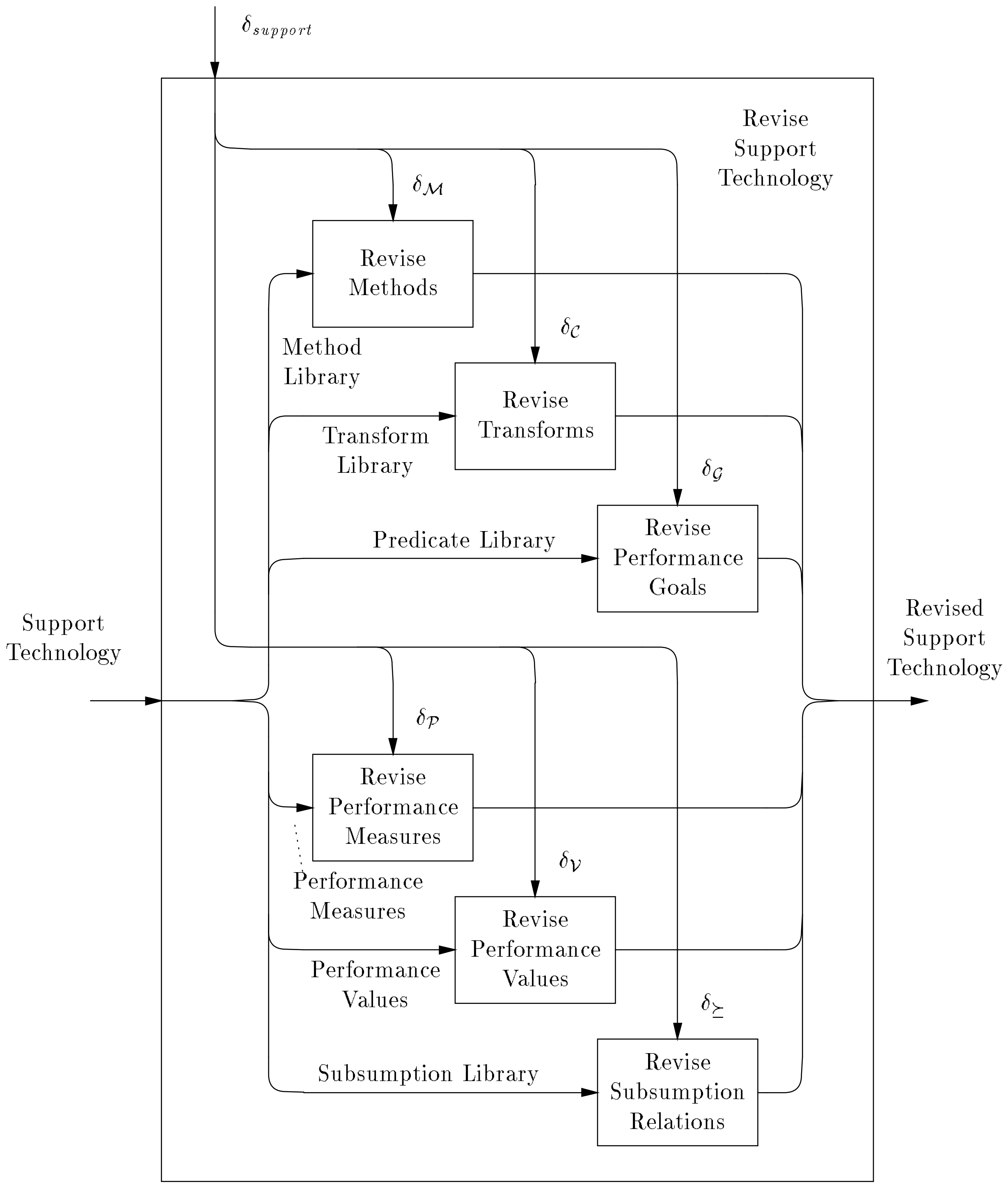


Figure 1.11: Updating Support Technologies

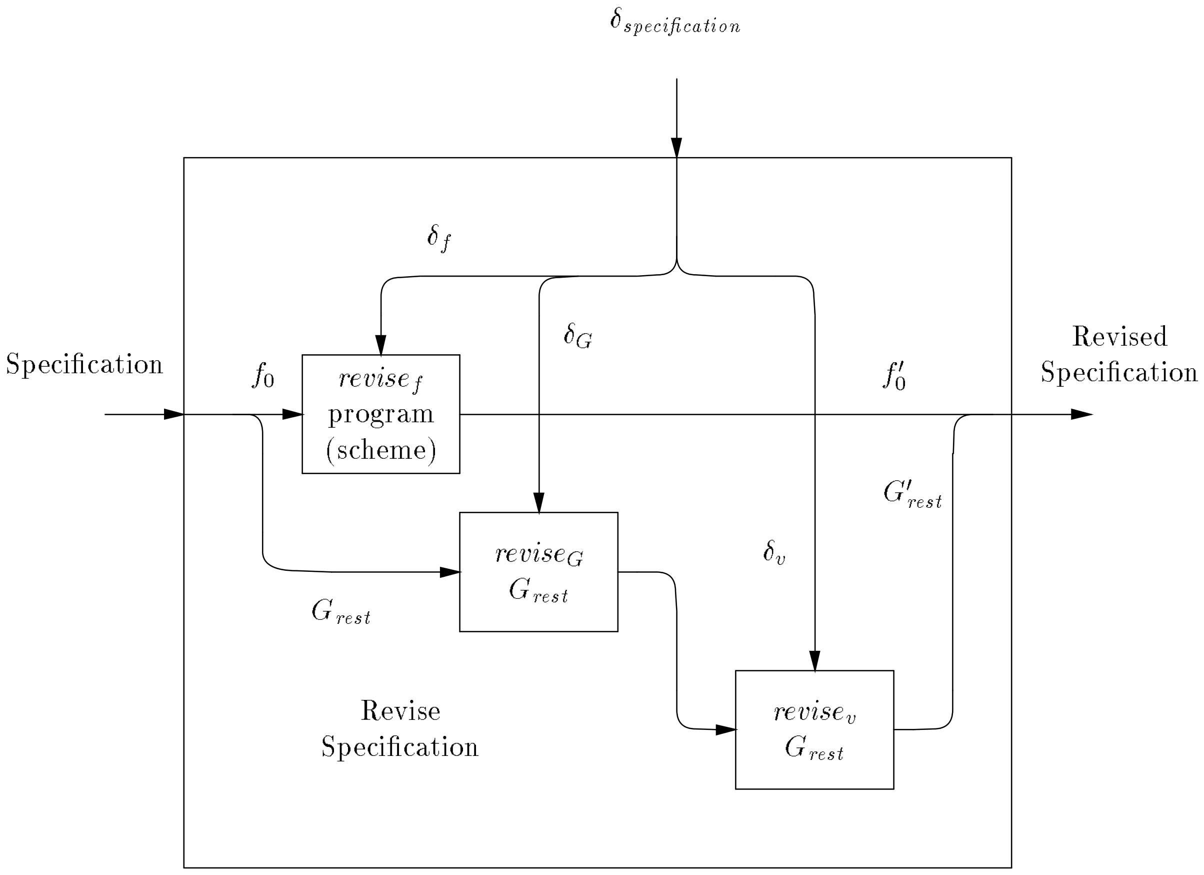


Figure 1.12: Updating Program Specification

the way the transformation system operates, as described in Chapter 3. Revising the specification consequently requires revising these parts (Figure 1.12). Deltas to performance specifications come in two flavors, with δ_v being a specialized version of the other, δ_G, accounting for the two processes that update the performance specification. These deltas and their revision procedures are also defined in Chapter 6.

The design pruning process (Figure 1.13) must consider all the aspects of the composite delta. Roughly, pruning removes those parts of the design history that are no longer valid by mark and sweep passes. Changes to the support technology make design decisions (including at least the applied transforms) that depend on those technologies illegitimate; these choices must be removed from the design history. Changes to the specification make other design decisions inappropriate; these, too, must be removed. We have separate procedures for each type of delta that simply mark the design decisions invalidated by that delta. A final pass removes all of the incorrect design choices in one sweep (*BATCHBANISH*) for efficiency reasons. Our preference is to maximize the invalid-marking at any point so that sweeps catch as much as possible on each pass, and to delay sweeps as long as possible. Details of the pruning process are spread across Chapters 7 and 8.

1.8 Contributions

The major contributions of this thesis are:

- A formulation of incremental evolution in terms of a *design maintenance system*
- An *architecture* for a design maintenance system based on a general transformation system model
- A formal classification of maintenance types
- The recognition of *explicit performance goals* as a necessary component in any design representation usable for a wide variety of maintenance tasks
- Procedures for revising design histories according to maintenance deltas based on a generalized notion of *commutativity* in the design space.

The essential value of the thesis is providing theory and building blocks needed to implement a Design Maintenance System. This is expected to lead to an incremental evolution software construction process in which formal deltas are installed via tools into implementations.

1.9 Thesis Organization

In this section, we briefly summarize the balance of the thesis. We first provide a brief overview of the organization in terms of chapters. We provide an in-depth summary of each chapter in Chapter 2.

A number of ideas and methods are required to implement the notion of Incremental Evolution. A dependency net of the major concepts is shown in Figure 1.14. We have chosen to present these concepts in the following order.

Chapter 3 provides a theoretical discussion on the nature of transformation systems, providing us with the concepts and vocabulary necessary as a prerequisite to

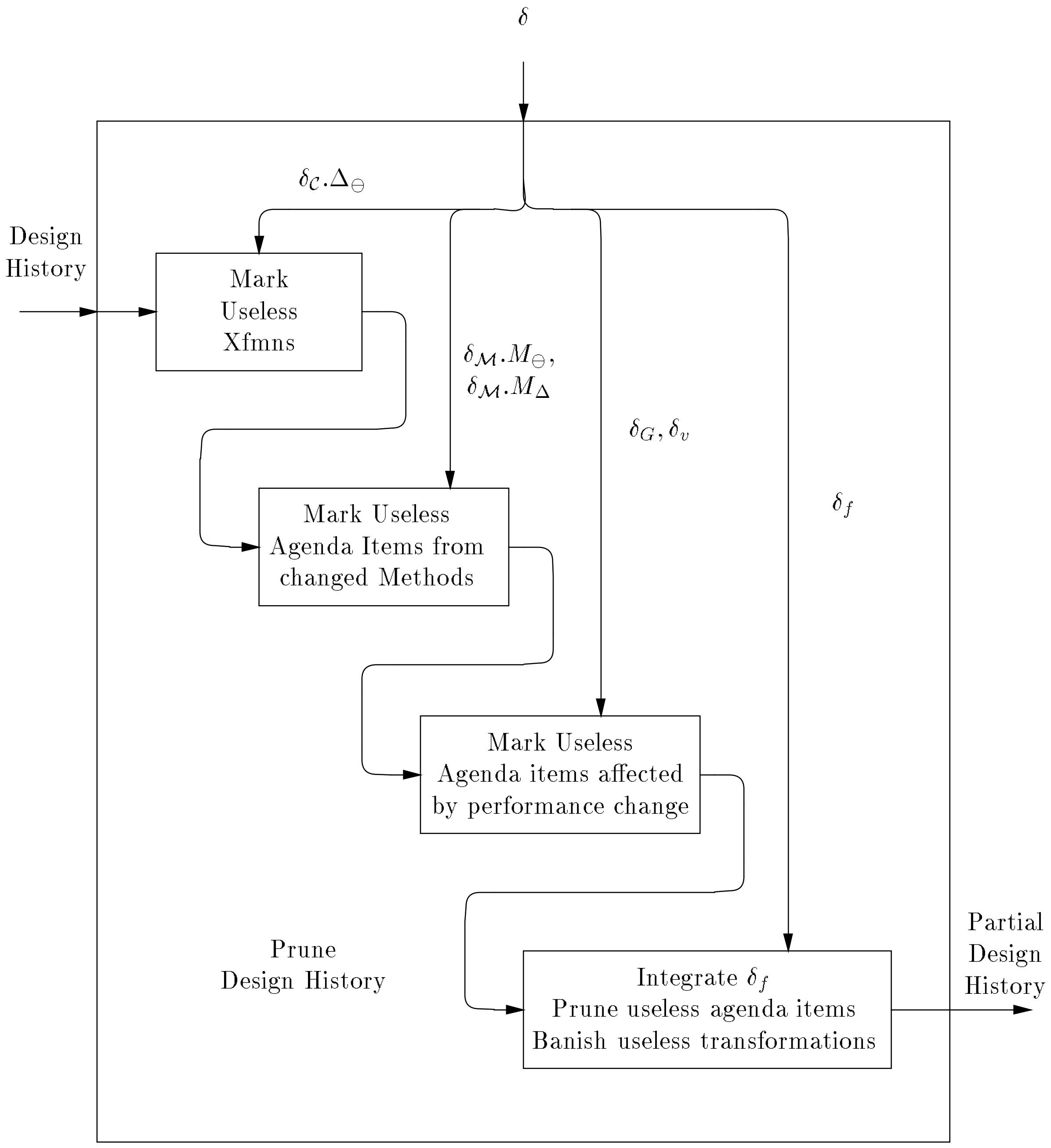

δ
$\delta_{\mathcal{C}}.\Delta_{\ominus}$
Design History
Mark Useless Xfmns
$\delta_{\mathcal{M}}.M_{\ominus}$, $\delta_{\mathcal{M}}.M_{\Delta}$
δ_G, δ_v
δ_f
Mark Useless Agenda Items from changed Methods
Mark Useless Agenda items affected by performance change
Prune Design History
Integrate δ_f Prune useless agenda items Banish useless transformations
Partial Design History

Figure 1.13: Pruning Design History according to Delta

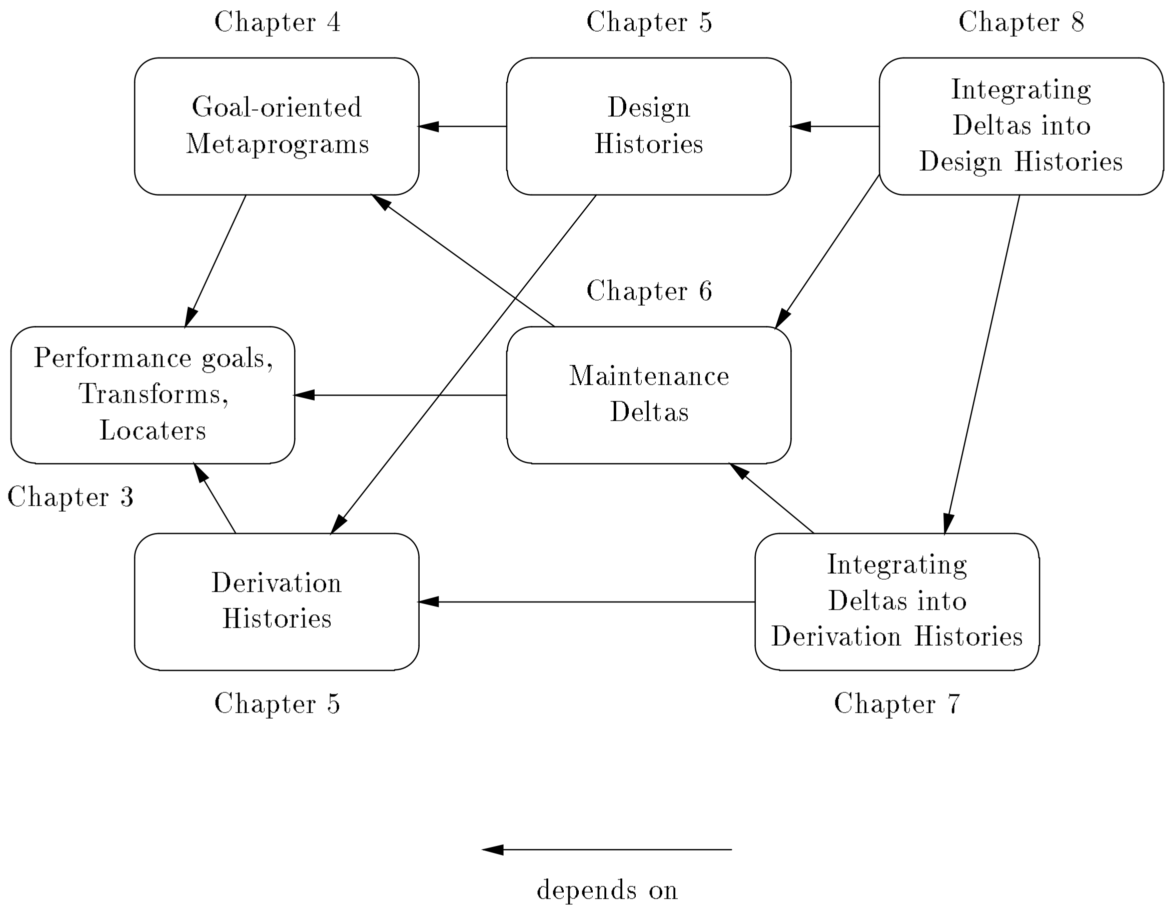

Chapter 4
Chapter 5
Chapter 8
Goal-oriented Metaprograms
Design Histories
Integrating Deltas into Design Histories
Chapter 6
Performance goals, Transforms, Locaters
Maintenance Deltas
Chapter 3
Derivation Histories
Integrating Deltas into Derivation Histories
Chapter 5
Chapter 7
depends on

Figure 1.14: Design Maintenance System Concepts

understand transformational maintenance. The theory describes *any* transformation system and so has very broad applicability.

Chapter 4 defines a *Transformation Control Language*, used to guide the application of transformations and provide the raw design information justifying the use of each transformation. A comparison to other control schemes is provided.

Chapter 5 shows how we can capture the sequence of transformations actually applied (the *derivation history* of an artifact), and the justification for applying them (a *design history*). This information is what we desire to reuse during transformational maintenance.

Chapter 6 characterizes transformational maintenance in terms of inputs to a transformation system, describes the notion of *maintenance delta*, or change to an input of a transformation system, gives an exhaustive list of such deltas for our model of transformation system, and provides representations for each. This allows us to capture a change as a formal entity.

Chapter 7 shows how we can use *commutativity* in the design space to provide basic mechanisms for rearrange a derivation history. Such rearrangements need to take into account the actual delta being processed. A practical scheme for installing commonly-occurring deltas is presented, along with a key example (Section 7.4.3).

Chapter 8 show how the design history can be used in conjunction with some of the delta types to determine useless transformations, and thereby provide direction to the derivation history rearrangement process.

Chapter 9 compares our Design Maintenance System with other (research systems) having related purposes or mechanisms, point out strengths and weaknesses of our approach.

Chapter 10 concludes by analyzing our Design Maintenance System, defines future research necessary to construct and validate a practical design maintenance system, and considers the impact of a design maintenance system on software engineering.

Appendices provide a notation index and psuedo-code for a complete system to integrating functional deltas.

Chapter 2
Thesis Overview

Chapter summary. A brief overview of the chapters in the thesis is provided, discussing each chapter topic and insights.

This is a rather large thesis. Trying to keep all the threads simultaneously may be difficult for the reader; it was for the author. We have included this chapter to provide a summary overview of the topics covered in the rest of the thesis, in the hopes that if the reader loses the thread, he can return here to pick it up again.

Each section corresponds to a chapter. We provide motivation for the chapter, a list of insights found in the chapter, and a discussion of the utility of these insights.

2.1 Transformational Implementation

Any mechanical scheme for managing change must be based on some formal construction process. We have chosen to build our DMS on one of the few truly formal construction processes known to us: Transformational Implementation. To ensure that our work is not limited to a single transformation system, or dependent on idiosyncratic properties of the same, we analyze transformation systems in Chapter 3. This provides us with vocabulary, concepts and definitions of the components of virtually any transformation system, as well as pointing out shortcomings of many existing ones. Such concepts and formal definitions are necessary to:

- define mechanisms for controlling the search through the design space
- define formal design histories
- shape the definition of types and representations of maintenance deltas
- allow us to reason about interactions between the design history and maintenance deltas.

This chapter provides the following insights:

- Specifications are *always* predicates (goals G). Most conventional transformation systems implicitly define a "specification" as a program fragment to be optimized, with no performance specifications. Our view unifies "conventional" specifications and performance specifications.
- The idea of a *correctness-preserving* transform is an instance of the more general notion of *property-preserving* transforms.
- The essence of a transformation system is its asymmetric treatment of subgoals $G_{invariant}$ and G_{rest} of the entire specification G to provide low-level control knowledge.
- Few transformation systems acknowledge the existence of performance specifications. Without them there is no formal motivation for the transformation system to apply *any* transforms!
- Absence of performance specifications limits the types of deltas expressible and therefore eventually processable by tools

The analysis of transformation systems is useful for several reasons:

- It is one of very few available. We claim a broader perspective in terms of performance predicates and the notion of property preservation for ours.
- It provides formal definitions for the concepts. These definitions can be used to classify existing systems.
- The notion of *locater* as a constraint over possible transform bindings. Locaters will be useful for reasoning "geographically" about interactions of transformations.

We expect our definitions to be helpful to those that are involved in the use, analysis or design of transformation systems simply by virtue of being formal. The concepts are defined in a general way so as to cover quite a wide variety of transformation systems, making comparisons of such systems simpler. The notion of locater we think will be an essential idea for any system that stores a derivation history.

2.2 A Transformation Control Language

A transformation system must somehow choose a sequence of transformations to apply. The control knowledge used to make those choices can come in a variety of forms. We explore one, TCL, designed to not only provide such control but also to generate the information needed to justify the final form of the program produced by the transformation system.

The ideas presented in this chapter include:

- Using AI planning ideas for metaprogramming.
- A *method*: a pair consisting of a plan and its purpose. This is used as control knowledge by using purpose to nonprocedurally locate a plan to apply. Recording method application provides design justification by connecting applied transformations back to purpose; this information is needed during design repair.
- Plan-like structure of methods
- *Locales*: computations over binding constraints, providing a mechanism for focusing the attention of the transformation system
- Clean separation of transformation actions from metaprogram. This allows reasoning about the transformations independently of the metaprogram, which is needed for managing derivation histories.

We provide

- The definition of locale, and an analysis of useful operators over locales
- A definition of a plan-like metaprogramming language, TCL
- A demonstration of its utility by modeling control mechanism of other transformation systems, showing that a number of implicit control schemes can be made explicit
- A comparison of TCL to existing control schemes

TCL is expected to be useful even if one does not want to perform transformational maintenance. It provides control in a way compatible with our general characterization of transformation systems. Its structure, based on methods, allows easy incremental addition of control knowledge. As an intermediate step to efficient transformational maintenance, it can be used for simple dynamic replay by mere re-execution.

2.3 Design Histories

While control languages such as TCL technically provide the ability to reimplement a changed specification by simply re-running the transformation process, it is our expectation that this process is expensive because navigation errors made will require considerable backtracking. Rather than rediscovering the choices made, it would be better to reuse stored choices. Design histories are the storage mechanism.

We record histories in two forms: *derivation history* and *design history*, with each design history including a derivation history. The derivation history provides a record of the actual transforms applied, where they were applied (locaters), and in what order. The design history provides justification for the applied transformations by capturing how the TCL metaprogram satisfied the original specification in terms of a goal/plan tree. The design history can consequently be used for explanation, but our interest is in using it to provide justifications for transformations proposed for reuse, and to provide indexes back into the generating TCL metaprogram for repair. During the design pruning process, we can use such indexes to locate portions of the TCL metaprogram that generated now-inappropriate transformations. The design repair process can reuse the purpose of a broken plan, and take advantage of TCL's nonprocedural nature to find a replacement method and therefore transformations. Lastly, a design history tells us which transformations work together to accomplish some purpose, and is therefore useful in locating the set of transformations made useless because a member transformation is no longer valid. Design histories are vital to reuse.

This chapter:

- formally defines a derivation history as a sequence of applied transformations
- defines operations useful on derivation history, such as indexing, splitting, concatenation and composition
- defines a design history as tree-structured plan capturing execution of a TCL metaprogram

We think that both the notion of derivation history and design history will be needed by any system which attempts to reuse the design choices. We expect the definition of a derivation history to stay the same in other systems[1]. The design history should be usable in explanations about the final program's structure to any software engineer. While the actual structure we use depends on TCL, any other transformation system using structures relating transformations to purpose will likely use something similar to our design history.

2.4 Maintenance Deltas

Given a formal development process, a formal maintenance process is possible only if we have formal descriptions of the desired changes: *maintenance deltas*.

Our view of maintenance is broad: it covers changes to every variable aspect of the underlying development process (we assume the development mechanism, i.e., the

[1]With the exception of nonlinear plans as an additional efficiency enhancement.

transformation system, is itself a constant). It handles changes to implementation technologies and definitions of performance (support technology deltas) as well as the conventional changes to a specification (specification deltas). Each change type requires procedures specific to that type to integrate a maintenance delta of that type into the design and the end artifact.

This chapter provides:

- A simple way of defining the set of maintenance deltas for a formal development process: one maintenance delta type per possible input
- Definition of an exhaustive set of delta types for our model of a transformation system
- Formal representations for each type of delta
- Definition of low-level (support technology) delta integration procedures

Our definition of delta types is more useful than that of conventional software engineering (informal) maintenance types in the following ways:

1. Formal definitions prevent confusion about what kind of change is desired
2. We have hope of providing mechanical procedures to handle each type of formal delta

Our rich model of transformation systems and supply of formal deltas make it evident that other researchers considering maintenance in a transformational context must consider more than just the so called *evolution delta*s currently popular. The support technology deltas also mesh cleanly with the pragmatic notion of *domain engineering*: the idea that one's implementation technology will evolve along with one's understanding of the problem domain.

2.5 Delta Integration into Derivation histories

We wish to install changes defined by maintenance deltas into existing artifacts. One way to do this efficiently is to reuse the design decisions from a previous transformational implementation. A derivation history contains many of those decisions, cast as applied transformations. We must consider how to reuse those transformations in the face of each type of delta. This is a necessary prerequisite to using the information from the design history.

What we learn in this chapter:

- *reuse* $\equiv$ *prune* **then** *repair*
  Surprisingly, reuse in transformational maintenance context consists mostly of identifying and removing obviously *reuseless* transformations, followed by regeneration of needed transformations. It is too hard to easily identify truly reusable transformations.
- The effect of several maintenance deltas on the shape of the design space
- That equivalent sequences of transformations (commutativity) can be used to rearrange a derivation history, for our convenience, into a likely-reusable part, and a definitely reuseless part
- maintenance deltas can guide the rearrangement process; this is why transformational maintenance with deltas is more efficient than simply reimplementing after applying a delta.
- Retaining a transformation may require changing its locater in a way dependent on the maintenance delta

These ideas are cast in the form of a number of essential procedures for rearranging a derivation history:

- *defer*: Put off application of a transformation until later
- *banish*: Move a transformation into the reuseless part of the derivation history. This can additionally serve as a kind of dependency-directed backtracking mechanism for the transformation system.
- *preserve*: Compute impact of a delta on a reusable transformation and vice-versa

These techniques are shown to preserve the legitimacy of the rearranged derivation history, so that any truncation (of the reuseless part) leaves a legal derivation history to be directly reused. The essential procedures are used to build certain delta integration procedures:

- Δ_C integration: Revise a derivation history according to a change in the available set of transforms
- Δ_f integration: Revise a derivation history according to a change in functional specification (a frequent type of deltas).

All of these procedures are illustrated with tree transformations. A key example of all these mechanisms at work is provided in Section 7.4.3. Lastly, since the techniques depend so heavily on commutativity in the design space, we present a number of empirical arguments as to why we should find that commutativity, including an experiment expressly conducted to measure it.

Any system using a derivation history should be able to take advantage of these methods; we expect AI planners in general to be able to use these techniques to handle plan repair when faced with a change in world state. We emphasize that backtracking based on *banish* is more effective than use of a dependency network because only essential interference ("different result") rather than the dependency nets' more conservative "uses result", determines what must be undone.

2.6 Delta Integration into Design Histories

Our original purpose was to maintain software efficiently by reusing design information. The preceding chapter showed how to use the design information available in a derivation history to handle certain deltas. However, just because a transformation from a derivation history initially appears to be reusable does not mean it serves a useful purpose in solving the revised problem defined by the maintenance delta. We must re-validate apparently reusable transformations to ensure that they still serve the purpose for which they were intended. Information about which transformations serve what purpose is recorded in the design history. We must also prune those parts of the design history which will be inappropriate for the revised artifact. Finally, we must repair the pruned design history by completing it, ideally using the same mechanisms that generate a fresh design history. This chapter is about integrating deltas in a design history.

The following points are made:

- All the transformations supporting the purpose defined by a method are reuseless if any one of them is (contamination)
- So called reusable transformations are only likely-reusable: they may no longer serve a useful purpose, or they may become contaminated
- Reuse of a design history consists of pruning away those parts which
  - are generated by now-invalid support technology
  - generated now-reuseless transformations

  and repairing the balance.
- Maintenance deltas identify invalid support technology and eventually reuseless transformations, providing a guide to pruning the design history
- Pruning should optimistically stop where the design history records the presence of an untried alternative

Using these ideas, we present theoretical procedures to:

- Further revise a derivation history to handle contaminated transformations
- Prune a design history for many of the identified delta types
- Execute TCL metaprograms by use of an agenda, unifying design history construction and repair

The utility of these insights and mechanisms is in providing the foundations for a practical Design Maintenance System:

- efficient transformational maintenance
- application to incremental evolution
- application to incremental domain evolution

2.7 Comparison to other Maintenance systems

Having defined the notion of a Design Maintenance System and provided mechanisms for supporting it, this chapter compares our methods to those of other existing production and research systems. Such a comparison is useful in determining how Design Maintenance System integrates existing ideas or advances new ones.

We primarily find that:

- Few maintenance systems use performance goals or record design histories containing performance goals. *without such information, there is no way to validate the utility of a re-used transformation.*
- Many derivation history replay systems block when encountering a problem
- No other derivation history replay systems reorder the sequence of transformations
- Explicit deltas are rarely used to guide installation of change.
- Nonlinear planners offer a notion of partial state which would be useful in further research.

To perform our comparison, we must often cast the concepts and methods of other systems in terms related to those of our broad transformational model. Such a recasting makes it easier to understand the relations between the systems, and the state of the field as a whole.

2.8 Conclusion

We conclude by analyzing the Design Maintenance System.

Interesting ideas that come from the analysis include:

- Essential versus artificial modularity: the true relation of design entities versus the firewalls installed in conventional software systems
- A perspective on architecture: those design aspects which are expensive to change (as opposed to those which are coincidentally present, such as friezes on Greek temples)

Last but not least, this chapter provides:

- An analysis of the problems with our system
- A list of topics for further research

The ultimate point of this work is to provide a solid foundation for the construction of a software process that supports a continuous model of design, *Incremental Evolution*.

2.9 Summary

A summary of the chapters of the thesis, in terms of content, lessons, and contributions has been provided.

With the overview completed, we turn our attention to the technical details.

Chapter 3
Transformational Implementation

Chapter summary. We provide definitions of basic concepts on which transformation systems are built, emphasizing performance specifications, left implicit in most transformation systems. The transformational implementation process is defined and analyzed. We discuss properties of the transformational design space. Both the definitions and the properties are needed to characterize and implement transformational maintenance.

Any approach to (semi-)automated software construction requires a formalization of the notions of specification, implementation, and some software construction process. Such a formalization is also a prerequisite to formalizing the notion of maintenance.

This chapter provides a formalization of the basic concepts involved in the software transformational implementation process, at the level of the artifacts manipulated directly, the mechanisms for manipulating the artifacts, and means for determining completion of the transformation process.

A number of papers about the theory and practice of particular transformation systems include [BD77, GB78, Kib78, Fea82, BM84, Nei84a, SKW85, BU86]. We describe a general model of transformation systems that emphasizes explicit performance specifications, which are implicit in most extant systems[1], and compare our model to some systems in detail.

We also consider aspects of the design space through which the transformation system must navigate to find a solution. The scale aspect provides us with the motivation to perform incremental maintenance rather than simply reimplement from scratch. Structural aspects of the design space will provide us with critical insights

[1]Mostow notes, [Mos85b]:

> Somewhat surprisingly, most of the systems ... leave the goal structure of the design implicit.

- Program(Scheme)s: Objects manipulated by a transformation system
- Performance Measures: functions that determine qualities of programs
- Performance Predicates: tests that programs have certain properties
- Specifications: Descriptions of desired properties of final programs
- States: Program plus cached inferences about program
- Transforms: Program modifiers
- Bindings and Locaters: Places on a state and place specifiers
- Transformations: Bound Transforms
- Property-preserving vs. Non-property-preserving transforms

Figure 3.1: Basic Concepts for Transformational Implementation

about how to accomplish such maintenance. In Chapter 4, we will discuss the higher-level issue of *control* of navigation through the design space.

3.1 Basic Concepts

Transformation systems are used to "transform" specifications into desired programs. A very simple model is that an abstract-but-inefficient program, taken as the specification of a desired computation, is incrementally changed into a concrete and efficient implementation by repeated application of "correctness-preserving" transformations [Fea79, pp. 2-10]. The individual transformations replace program fragments containing inefficient constructs with efficient program fragments that have identical properties concerning the computed results (thus the term correctness-preserving).

Our model is a bit more detailed. Before we can discuss the transformation process, we need to consider the fundamental concepts (Figure 3.1).

We first discuss the artifacts manipulated by transformation systems, called *programs*. Arbitrary qualities of programs can be determined by applying appropriate *performance measures. Performance predicates* over programs determine if a program has some desirable property, and usually are defined in terms of a relation between a performance measure and some fixed performance value. We then consider *specifications*, which ultimately determine which program a transformation system is supposed to produce. Specifications used in practical systems can be divided into a number of *performance goals*, which determine if a program achieves a desired aspect; this information is eventually used to control the search for a solution. Performance goals are defined in terms of performance predicates.

We then move on to define the notions of *transform*, which cover the notion of rewrite rules, and *transformation*, which are applications of the rules. We discuss the idea of *binding*, which defines a place in a program, and *locater*s, which are a specification of a place. These ideas are need to define the mechanics of the transformation process. The characterization of transforms as *property-preserving* or not turns out to be a key aid to guiding the transformation system.

A very interesting alternative characterization is provided by [BEH$^+$87, Part II]; our approach shares the notion of program schemes, and a version of performance predicates. Another good general survey of transformation systems can be found in [Fea86].

Practical transformation systems must have the the basic concepts instantiated before they can be used. This process has been called *domain engineering* [Ara88] and is a difficult problem in its own right. We will touch on the role this plays occasionally in this section.

3.1.1 Program Schemes

The main purpose of a software development process is to produce an executable computer program. A transformation system must consequently manipulate representations[2] of computer programs as data objects. It is convenient to process generalizations of computer programs which are identical in all but a few places; we represent the differing places by parameters[3] standing for program fragments, and call a program with zero or more scheme parameters a program scheme:

DEFINITION 3.1: *Program Scheme.* A syntactic construct representing a class of programs, allowing parameters where one would expect complete syntactic constructs. Scheme parameters can be instantiated by substituting other program schemes. □

We use "?*name*" to denote program scheme variables.

[2]The conventional wisdom is that of Agresti [Agr86]:

> Transformational implementation is an approach ... to apply a series of transformations that change a specification into a concrete software system.

This implies that a transformation system "transforms a specification" into a program. They transform programs, not specifications.

[3]Not the typical variables of procedural languages.

An example program scheme is the PASCAL program fragment containing both a PASCAL program variable x and a scheme parameter $?m$ for the body of the loop:

$$\textbf{while } x \textbf{ do } ?m$$

A program scheme without parameters is just a particular program. This definition follows that of the CIP project [BEH$^+$87, BMPP89], which suggests that one should not only use transformation systems to develop programs, but also to develop program schemes. This potentially allows a transformation system to participate in the construction of its own transforms, as many transforms have representations which include a pair of program schemes with shared scheme parameters [EM85, p. 124], [PS83].

Since our intention is to characterize the transformation process without committing to representational details, we avoid (as far as possible) defining any particular structure for program schemes (or any other objects involved in the transformation process), although we will use some in examples. Instead we depend on the interactions of the objects to define their essential properties, in the style of category theory [AM75, Gol84].

To prevent cluttering the text, we shall use the term "*program*" to mean "program scheme". We use $\mathcal{F}$ to denote the set of possible programs, and $f_i \in \mathcal{F}$ to denote particular instances. The symbol f was chosen because historically each program implicitly represented some desired *functionality* in transformation systems. The notion of performance measures in the next section makes it clear that functionality is merely a derived property of each program. As a mnemonic aid, we suggest you think of fs as program *forms*.

Programs are represented in a variety of ways. We provide some sample program representations:

- Strings representing a sentential form (string derivable from the goal symbol) of a chosen grammar [ASU86, p. 168], with named nonterminal instances. For a simple PASCAL grammar, the following is a program with scheme parameters $?x$ and $?y$ for nonterminals *TARGET* and *EXP* respectively:

$$\langle ?y : \mathit{TARGET}\rangle[i] := \mathit{alpha} + \langle ?x : \mathit{EXP}\rangle;$$

- Trees representing terms $t \in T_{OP}(X)$ determined by a signature $\langle S, OP\rangle$ with S being a set of sorts, X being a set of parameter names, and inductively defined by recursive composition of OP, a set of constant and operation symbols, over terms [EM85, p. 17]. Tree nodes represent operators from OP or scheme parameters.
- Jungles: forests of acyclic hypergraphs, with nodes and edges labeled with sorts, operation symbols, and parameter names taken from a signature [HKP87] used to represent terms with identical substructures.
- Graphs representing algorithmic programs, with nodes representing operators, program variables, or scheme parameters, and arcs representing connections between them [Ehr78, vdB81, Sow84]. Edges can reflect "consumes" for value-producing operators, "transfers-control-to" for control-operations, "defines" for variable declarations.
- Semantic networks with virtual links (see Section 9.4.7)

3.1.2 Performance Measures

A transformation system must determine if the program it is currently manipulating has some desired property. To do this usually requires two steps: computing some quality called a *performance measure*, and then comparing that measure to some reference value. There are typically many possible aspects of an artifact of interest; most will require a performance measure.

DEFINITION 3.2: *Performance Measure i.* A function $p_i : \mathcal{F} \to \mathcal{V}_i$ from programs to a set of *performance values* $\mathcal{V}_i$. □

We denote individual values in $\mathcal{V}_i$ as v_k, or $v_{i,k}$ when the performance value type would be otherwise unclear. We use $\mathcal{P}$ to denote the set of possible performance measures, and P to denote particular sets of performance measures.

This definition covers such diverse measures as:

- implementation technology measures such as programming language:

  $p_{language} : \mathcal{F} \rightarrow \mathcal{V}_{language} \equiv \{FORTRAN, C, C{+}{+}, SNOBOL, LISP, PROLOG, \cdots\}$

- source line count: $p_{sloc} : \mathcal{F} \rightarrow \mathcal{V}_{sloc} \equiv \mathbf{Nat}$ (Natural numbers)
- GOTO count: $p_{gotos} : \mathcal{F} \rightarrow \mathcal{V}_{gotos} \equiv \mathbf{Nat}$
- McCabe's cyclomatic complexity numbers:

  $$p_{McCabe} : \mathcal{F} \rightarrow \mathcal{V}_{McCabe} \equiv \mathbf{Nat}$$

  and Halsted's volume/level measures [Fai85, p. 324]:

  $$p_{Halsted} : \mathcal{F} \rightarrow \mathcal{V}_{Halstead} \equiv \mathbf{Real}$$

- Module coupling and cohesion [Fai85, pp. 148-149]:

  $$p_{coupling} : \mathcal{F} \rightarrow \{content, common, control, stamp, data\}$$

  $$p_{cohesion} : \mathcal{F} \rightarrow \{ \begin{array}{c} coincidental, logical, temporal, communication \\ sequential, functional, informational \end{array} \}$$

- $\mathcal{O}$ complexity cost computations [AHU74, p. 2]:

  $$p_{complexity} : \mathcal{F} \rightarrow \mathcal{V}_{complexity} \equiv \mathbf{Polynomials}$$

- Denotational program semantics [Sto77, Pag81, All86]:

  $$p_{meaning} : \mathcal{F} \rightarrow \mathcal{V}_{meaning} \equiv \mathbf{Functions}$$

  Another possible form for $\mathcal{V}_{meaning}$ are input/output predicates and extensional relations [MDG86].

- Theories (complete set of facts known about a program [TM87]):

  $$p_{theory} : \mathcal{F} \rightarrow \mathcal{V}_{theory} \equiv \{theories\}$$

- Models of programs as algebraic specifications [ST88]:

  $$p_{models} : \mathcal{F} \rightarrow \mathcal{V}_{models} \equiv powerset(\mathbf{Algebras})$$

- Termination [BPW80]:

  $$p_{terminates} : \mathcal{F} \rightarrow \mathcal{V}_{terminates} \equiv \{true, false, unknown\}$$

- Vague "-ilities" such as

  $$p_{readability} : \mathcal{F} \rightarrow \mathcal{V}_{readability} \equiv \{low, medium, high\}$$

  assuming they can be formalized.

In particular, performance measures are intended to cover those properties which are a consequence of the particular form of the program rather than its derivation. Pidgeon [Pid90] characterizes similar measures as "observation channels", but also allows observations on resources consumed during development.

Functionality ($p_{meaning}$) is commonly assumed to be *the* aspect intended by a program. However, our perspective is that a program has multiple measurable aspects, of which functionality is merely an arbitrary choice.[4]

Performance measures are not always easy to compute. In some cases, approximations will do. For computational complexity, determining the actual cost can be very hard to do in general, but one can build conservative estimators as is done by MEDUSA [McC88]. Sometimes, however, we can actually finesse computing a performance value entirely (we will see later that the non-symmetric nature of transformation systems allows us to get away with this). When treating algebraic specifications as programs, one would not ever want to actually compute the set of algebras which are models for a particular algebraic specification [ST88], but one does want to reason about that set, and so the notion p_{models} is still useful.

Reasoning about performance values is aided by an abstract notion of performance subsumption: the intuitive idea is that some performance values are at least as good as others. For *every* set of performance values $\mathcal{V}_i$, we assume the existence of a (possibly trivial) binary *subsumption* relation, in which every value subsumes itself and possibly some other values. We will use this later to define a class of property-preserving transforms.

DEFINITION 3.3: *Subsumption.* A preordered relation $\succeq_i \subseteq \mathcal{V}_i \times \mathcal{V}_i$. If $x \succeq_i y$, we say that x subsumes y. □

Many subsumption relations are actually partial[5] orders, although we have little need of that fact.

[4]This perspective is given strength by an unsolved problem in secure operating systems: preventing covert signaling channels (example: a supposedly trustworthy Trojan program leaking confidential information as a bit stream by changing the page fault rate to signal ones versus zeros to a detection device outside the system). From the point of view of the spy, the program's functionality is to leak bits, not to perform the service asked by the application. This is simply a instance of choice of an unusual performance measure as "functionality".

[5]Preordered: $\forall x, y, z : x \succeq x$, and $(x \succeq y) \wedge (y \succeq z) \supset (x \succeq z)$. A partial order also requires that $(x \succeq y) \wedge (y \succeq x) \supset x = y$.

Likely examples of subsumption relations are:

- V_{sloc}: the complete standard ordering $\leq$ over **NAT** defining $v_1 \succeq_{sloc} v_2 \equiv v_1 \leq v_2$.
- $V_{cohesion}$: degree of cohesion [Fai85, p. 149]:

$$\begin{gathered} \textit{informational} \succeq \textit{functional} \succeq \textit{sequential} \succeq \\ \textit{communication} \succeq \textit{temporal} \succeq \textit{logical} \succeq \textit{coincidental} \end{gathered}$$

- $V_{complexity}$: asymptotic polynomial domination

$$O(1) \leq O(n) \leq O(n \log n) \leq O(n^2) \leq O(2^n)$$

  defines

$$O(1) \succeq_{complexity} O(n) \succeq_{complexity} O(n \log n) \succeq_{complexity} O(n^2) \succeq_{complexity} O(2^n)$$

- $V_{language}$: language subsetting:

$$C \succeq_{language} C{+}{+}$$

  A program is surely in *C++* if it is in *C*; there is no relation between *C* and *PROLOG*.
- $V_{language}$: use of language in production software development environments:

$$\textit{FORTRAN} \succeq_{production-oriented} C \succeq_{production-oriented} \textit{PROLOG}$$

  and

$$C \succeq_{production-oriented} \textit{FORTRAN}$$

  We interpret this as "FORTRAN is just as production-oriented as C" and vice-versa, with both being more production-oriented than *PROLOG*. Since it is obvious that $\textit{FORTRAN} \neq C$, $\succeq_{production-oriented}$ is only a preorder.
- $V_{meaning}$: function generalization:

$$f \succeq_{meaning} g \iff \forall x : \textit{defined}(g(x)) \supset \textit{defined}(f(x)) \wedge g(x) = f(x)$$

  i.e., f computes everything that g computes, and perhaps something else besides.
- V_{theory}: theory inclusion:

$$p_{theory}(f_1) \succeq_{theory} p_{theory}(f_2) \iff p_{theory}(f_1) \supseteq p_{theory}(f_2)$$

  This is the notion of implementation defined by [TM87].

- V_{models}: model subsets:
$$p_{models}(f_1) \succeq_{models} p_{models}(f_2) \iff p_{models}(f_1) \subseteq p_{models}(f_2)$$
This is the notion of implementation defined by [ST88].
- $V_{readability}$: "more readable":
$$high \succeq_{readability} medium \succeq_{readability} low$$

Broy [BPW80] argues for the utility of program relations constrained to be preorders, and provides quite an interesting list of possible relations, including some for nondeterministic programs. We emphasize preorders on the more primitive notion of performance measures because of their value as generators of such program relations.

3.1.3 Performance Predicates

Performance predicates verify that a program has a desired property (say, computes the desired result by virtue of denoting a particular function), rather than determining some performance measure.

DEFINITION 3.4: *Performance Predicate.* A predicate $G_i \subseteq \mathcal{F}$ over programs. □

The symbol G was chosen in anticipation of using performance predicates as *goals* in the transformation process. We use $\mathcal{G}$ to represent the set of possible performance predicates, and g_i to represent particular primitive predicates. We use G to represent predicates when we know little about any structure they might have, or when they are explicitly composed from primitive predicates, such as a conjunction. We will sometimes treat conjunctive predicates as a sets and use set notation to manipulate such predicates; e.g., we will write $g \in G$ if $G = \ldots \wedge g \wedge \ldots$.

Performance predicates are not given extensionally, but can be supplied as characteristic functions $G_i : \mathcal{F} \rightarrow$ **Boolean** or specializations. A rather trivial example is $G_{FORTRAN}(f)$, which is true if f is written in *FORTRAN*; another example, for structured programmers, is $G_{no-GOTOs}$.

Performance predicates G_i are often definable terms of performance measures:
$$G_i : (\mathcal{F} \rightarrow \mathcal{V}_j) \rightarrow \textbf{Boolean} \equiv G_i(f) \equiv g_i(p_j(f))$$
We might define $G_{no-GOTOs}(f) \equiv g_{is-zero}(p_{GOTO-count}(f))$. In fact, we can often encode a performance predicate as a relation between a performance value computed by some p_i and an explicit desired performance value constant, $v_c \in \mathcal{V}_i$:
$$G_i : (\mathcal{F} \rightarrow \mathcal{V}_j) \times \mathcal{V}_j \rightarrow \textbf{Boolean} \equiv G_i(f) \equiv g_i(p_j(f), v_c)$$
An example is $G_{fits-in-one-page}(f) \equiv p_{sloc}(f) \leq 66$; our *FORTRAN* tester becomes $G_{FORTRAN}(f) \equiv p_{language}(f) = FORTRAN$.

For a performance predicate G defined in terms of a relation between a performance measure p_i and a desired constant $v_c \in \mathcal{V}_i$, the relation g is often the subsumption relation $\succeq_i$, as with $G_{fits-in-one-page}$. A more interesting case is $G_{FFT}(f) \equiv p_{meaning}(f) \succeq v_{FFT}$, which says a program f is an FFT program if it computes FFTs, and perhaps something else besides. We shall have use for performance predicates based on subsumption relations.

In analyzing a number of algorithm syntheses, Steier [SA89, 104] found that most 'nonfunctional' (i.e., performance) goals were not explicitly represented, although they invariably drove the synthesis process. He claims that we do not know how to express useful performance goals yet, and that further research is required to determine this. We make no claim that our characterization completely solves the problem; we suggest that one must start somewhere with an explicit representation, and our characterization seems like an obvious first choice. We obviously have not determined *which* performance measures or goals are useful.

3.1.4 Specifications

A software development process must convert a vague notion of a customer ideal into a running computer program (system). There are, conceptually, two major steps to this process:

- Conversion of "vague notions" into concrete goals.
- Construction of a program that achieves those goals.

The process of converting such "vague notions" into concrete goals is generally called *requirements analysis* [Pre87, Lei87, Lei88] and is an extremely difficult problem in its own right. Part of the difficulty is in acquiring the proper vocabulary in which to state the goals, and has been pursued to some extent by others [Nei80, Ara88] under the name *domain analysis*. Another problem is the conversion of a customer's desires into a description using a predefined vocabulary and validating that conversion [Fic87, RW88, Lei88]. Yet another difficulty is encoding a goal achievable with the implementation technologies at hand; Arango [Ara88] outlines a *domain engineering* methodology that defines vocabularies for implementable solutions using a set of reusable components.

As the point of automatic programming is to convert desires into programs, a necessary step is to acquire a fully *formal* statement of the requirements, on the assumption that automatic programming cannot occur with informal descriptions.

For the purpose of transformational implementation, (and to clarify our view of what the terminology of traditional SE should be), we define:

DEFINITION 3.5: *Requirements.* An *informal* statement of the goals to be achieved by an artifact. □

and we define:

DEFINITION 3.6: *Specification.* A *formal* statement of the goals to be achieved: a predicate $\subseteq \mathcal{F}$. □

Specifications are usually defined intensionally with performance predicates over programs.

DEFINITION 3.7: *Performance goal.* Any performance predicate used in a specification. □

We note that the goals for an artifact may cover not only its functionality, but also its form and properties derivable from the form. Thus our notion of specification covers not only functionality of programs in terms of input and outputs, but also what is conventionally termed performance, such as space, and time, as well as less conventional properties such as target language, degree of module cohesion, or models.

The problem of acquiring the requirements, and maintaining traceability from requirements to specifications is important, but beyond the scope of this thesis.

A traditional SE definition of specification emphasizes that the specification describe *what* is desired, rather than *how* the final artifact should work [Fai85, p. 88]. This is valuable in the sense that it decouples possible implementations from characterizations of what are valid solutions, leaving the implementors as much freedom as possible. In this view, "*how*" is essentially an executable program.

We do not see specifications as necessarily *what*. A formal specification may, in fact, insist on the use of particular programs for accomplishing certain aspects of a desired computation (probably requiring a $G_{contains} : \mathcal{F} \rightarrow \mathbf{boolean}$ in order to state it), without making it any less of a specification[6]. This view of specification is consistent with very high level specifications such as predicate calculus with sets, and very low level specifications such as state machines and procedures, depending on the particular specification formalism used.

We explicitly avoid the notion of *process specifications*: constraints over resources *consumed during the construction (or modification)* of an artifact, such as CPU-time (especially that expended to compute performance or other process measures/predicates), man-hours, dollars, LISP-machines, number of transformations applied, etc. We do this to restrict the scope of the research to manageable size.

[6]Further discussion of specifications as programs and programs as specifications can be found in [TM87].

3.1.5 Specifications defined by multiple Performance Goals

The extensional characterization of a specification (as a subset $G \subseteq \mathcal{F}$) is not appropriate for practical use, because it is impractical to construct. Given various domains of discourse i for describing relevant aspects of a desired artifact, a single monolithic predicate '*specification*' is likely to be expressed as the conjunction of a number of sub-predicates G_i each expressing conditions for a particular performance aspect i:

$$\textit{specification} \equiv \bigwedge_{i=1}^{n} G_i$$

Typically, one performance goal will constrain what is traditionally termed the *functionality* ($p_{meaning}$); this term specifies what the ultimate program f is to supposed to compute (as opposed to how "well" it does it), and is the goal traditionally given primary importance. The remaining goals specify "lesser" performance properties of the implemented program. Functionality is emphasized over other performance goals simply because, in practice, most customers prefer non-functionality-performance degraded programs over functionality degraded programs. We observe that tradeoffs do exist, and functionality is sometimes traded away to achieve better performance on a lesser functionality; typical is the implicit acceptance of bounded-size integers in C programs due to their efficiency in widely available machines with fixed word sizes. Note that the specification is stated in terms of the *syntactic structure* of the ultimate program f; this allows us to extract function and other performance properties by inspecting what f actually does.

Specification Styles

In practice, some representation of the specification must be provided to a transformation system. Assuming a conjunctive specification, what is really required are representations for the individual performance goals. We see several practical styles of representing such individual goals, possibly mixed in any one conjunctive specification:

- *direct specifications*, providing G_i directly
- *performance bound specifications*, defining $G_i(f) \equiv p_i(f) \succeq_i v_{i,j}$ in terms of a "specification" $\langle p_i, v_{i,j}\rangle$. The value $v_{i,j}$ is called a *performance bound*.
- *indirect specifications*, defining $G_i(f) \equiv p_i(f) \succeq_i p_i(f_0)$ in terms of a "specification" $\langle p_i, f_0\rangle$
- *base specifications*, defining $G_{base}(f) \equiv p_{base}(f) \succeq_{base} p_{base}(f_0)$ in terms of a "base specification" program f_0. Which performance measure p_{base} is used is a constant for each transformation system; obviously a transformation system can have at most a single base specification in a conjunctive specification. Such specifications are conventionally known as *functional specifications* because of the frequent additional assumption that $p_{base} \equiv p_{meaning}$. This is, unfortunately, the conventional meaning of specification accepted in the transformational community.

We call a specification containing mixed styles a *mixed specification* and write it as a tuple containing style instances as elements, so $\langle f_0, v_i, g_j\rangle$ is a specification containing a base specification f_0, a performance bound v_i, and a direct specification g_j.

Direct specification is just that; the system analyst must encode his desired performance goals directly in some predicate-constructing language understood by the transformation system, or perhaps select a performance goal from a list built into the transformation system (such as the frequently built-in, very complicated predicate $G_{optimizedsomewhat}$). We are not very interested in the structure of such a predicate-constructing language for this thesis, as the special cases that are interesting to us are covered by the other specification styles. A simple example of such a language would allow the vocabulary we have defined so far, i.e., allow references to performance measuring functions, performance values, and various relations between them. An example direct specification might then be represented as:

$$\begin{array}{l} p_{meaning}(f) \succeq_{meaning} v_{meaning,FFT} \\ \wedge p_{complexity}(f) \succeq_{complexity} \mathcal{O}(inputsize(f)\ \log inputsize(f)) \\ \wedge p_{sloc}(f) \succeq_{sloc} 1000 \\ \wedge p_{language}(f) \succeq_{language} LISP \\ \wedge p_{readability} \succeq_{readability} medium \end{array}$$

This would describe a program that computed a particular function $v_{meaning,FFT}$ (say, a Fast Fourier Transform), had $\mathcal{O}(n \log n)$ running time or better, was at most 1000 source lines in size, was coded in LISP or some subset of LISP, and was anywhere from moderately to highly readable.

Performance bound specifications take advantage of the fact that many performance goals are simply upper bounds on acceptability of some performance measure; it is sufficient to simply supply the upper bound, as the rest of the predicate is stylized, and can be generated automatically. A common shorthand for specifications of this form is simply a list of performance value constants $v_{i,j} \in \mathcal{V}_i$ implicitly defining the specification:

$$\bigwedge_i p_i(f) \succeq_i v_{i,j}$$

The preceding specification is then written simply as:

$$\langle v_{meaning,FFT}, n \log n, 1000, LISP, medium \rangle$$

We will see variations of this type of specification when we attempt to change the (non-"functional") performance of an existing artifact.

Indirect specifications derive a performance bound specification from a supplied pair $\langle p_i, f_0 \rangle$. This can occur in practice when an existing implementation f_{93} is satisfactory for some performance aspect p_i, but not another, and the engineering organization wishes a new artifact at least as good as the old; thus $\langle p_i, f_{93} \rangle$ as an indirect specification. Another good reason for using indirect specifications is that $v = p_i(f_0)$ may be difficult to represent, especially if $\mathcal{V}_i$ is constructed in a very general way, whereas f_0 may be encoded in a specialized problem domain language suitable for the job at hand[7]. This is one of the essential idea behind the Draco paradigm [Nei84a], and Draco specifications are in fact precisely such pairs $\langle p_{domainmeaning}, f_{domaininstance}\rangle$. The complications are then effectively hidden in p_i and fall on the domain engineer rather than the specifier, at the price of having the specifier learn a specialized language, and mentioning p_i along with his program.

Base specifications occur not only for the same reasons as indirect specifications, but also for a practical reason that we outline in more detail in Section 3.2.2: the need for a transformation system to start with a program. Most often $p_{base} \equiv p_{meaning}$ is assumed because of its complexity, and the specification f_0, often an instance of a wide spectrum language, is called a *functional specification.* More specialized transformation systems such as TAMPR [BM84], accepting functional LISP, and Belkhouche's abstract-data-type implementor [BU86] also allow functional specifications with assumed $p_{meaning}$.

Wide spectrum languages with predicates and set notations tend to make it easy to confuse a direct specification with a program. Confusing the problem description with the actual specification is consequently common. Example: any procedural program denotes some function. The function is what the transformation system is to implement; it does so by manipulating the program.

We see that specifications for practical transformation systems are *always* performance goals, albeit in various disguises.

[7]Traditional transformation systems with wide spectrum languages [SKW85, Bal85a, BMPP89], fit this characterization if we simply treat the wide-spectrum language as a domain language. It is interesting to note that Smith [Smi89] describes the specification acquisition process for a wide-spectrum-language-based transformation system as first requiring definitions of appropriate formal terminology for the problem, before specifying the problem itself; the difference is thus one of make-domain-now versus use-pre-existing domain.

3.1.6 Design States

In practical situations, the transformation system will have a program f_i, and a set of specific consequences $Q_i = \{q_{i,j}\}$ inferred and cached about that particular program scheme. Caching is needed to avoid repeated re-computation of the same results. Typical consequences may be:

- data flow analyses
- value range restrictions induced by context
- estimated execution frequencies

Such consequences can be defined as performance measures over programs, but are usually not used in goal predicates.

DEFINITION 3.8: *Design State.* A pair $s_i = \langle f_i, Q_i \rangle$ consisting of a program f_i and a set $Q_i \subseteq \{\, \langle p, p(f_i) \rangle \mid p \in \mathcal{P} \,\} \cup \{\, \langle g, g(f_i) \rangle \mid g \in \mathcal{G} \,\}$ of cached conclusions drawn about f_i. □

We use $\mathcal{S}$ to represent the set of possible design states, with s representing individual states. We extend the definition of performance measures p and performance predicates g to allow application to states, by applying them to the program component of a state, as follows:

$$\forall p \in \mathcal{P}, s = \langle f, Q \rangle \supset p(s) \equiv p(f)$$

$$\forall g \in \mathcal{G}, s = \langle f, Q \rangle \supset g(s) \equiv g(f)$$

Practical versions of performance measures and predicates may take advantage of the cached facts Q to speed up the computations.

An instance of what we call design states is the representation scheme dubbed "webs" by [MSNT88]. This scheme captures both an abstract syntax tree for a program, and also captures *used* and *ref* data flow analysis results as labeled links between nodes in the abstract syntax tree.

Pidgeon [Pid90] proposes that design states include not only a program, but also the desired specification G, measures of consumed resources, as well as the definitions of the performance predicates themselves. All this information was necessary to model rationality of the transforming agent, even in the face of learning. Our states are much simpler because we do not deal with resource management, and our specifications are static during the course of implementation.

3.1.7 Transforms, Bindings, Locaters and Transformations

Intuitively, a *transform* is simply some formal modification to be applied to a program. We extend the notion to transforms applied to states composed of programs and consequences; this allows many consequence-generation rules to be defined as transforms (as is done for webs). Transforms often can be applied to several places in a state, and so we have a need of a mechanism, traditionally called a *binding*, to specify precisely where and how a transform "matches" to the state. Bindings are always dependent on the transform and the state in which it is applied.

DEFINITION 3.9: *Transform.* A two argument partial function:

$$t : \mathcal{S} \times \mathit{bindings}(t, s) \rightarrow \mathcal{S}$$

which maps state (programs) via bindings to new states (programs). $\square$

The set of possible transforms is denoted $\mathcal{T}$. Individual members $t_i \in \mathcal{T}$ are distinguished by subscripts. Sets of transforms are denoted T. We use $\mathcal{B}$ to denote the set of all possible bindings of all possible transforms to all possible states, and understand that application of a particular transform requires a binding appropriate to that transform and the state to which it is being applied.

Since a binding must specify some sort of connection to the state, and we wish eventually to record transitions between states $\langle s_1, s_2 \rangle$ caused by applications of transforms without reference to the actual state, we introduce the notion of a *locater* as a kind of state-relative pointing device. Unlike bindings, a locater value is independent of any transform or any state, but acts as a constraint on bindings when used for any particular transform on a state.

DEFINITION 3.10: *Locater.* A constraint on bindings:

$$\ell : \mathcal{S} \times \mathcal{T} \rightarrow \mathit{powerset}(\mathcal{B})$$

$\square$

We denote the set of possible locater values by $\mathcal{L}$, with individual members $\ell \in \mathcal{L}$. For the purposes of this thesis, one can think of locaters as specifying a place in a state according to some "geometry" dependent on the state representation, but in general they are just simply constraints[8]. Finding a suitable set of locaters for various state representations may be a difficult problem which we ignore for this thesis (but see the definition of *path*, below, for a practical locater for states represented by trees).

[8]If one considers that transforms might be parameterized, the locater also includes constraints on the parameters.

Most transformation systems contain a pattern matcher to decide precisely how a transform matches a state; since a locater may allow a transform to match a state in more than one way, this is the set-valued total function:

$$match : \mathcal{T} \times \mathcal{S} \times \mathcal{L} \rightarrow powerset(\mathcal{B})$$

The pattern matcher is used by a transform applier (partial function) to apply transforms to states given locaters:

$$apply : \mathcal{T} \times \mathcal{S} \times \mathcal{L} \rightarrow \mathcal{S}$$

The value of this function is well-defined only when a unique binding is chosen by the matcher:

$$defined(apply(t, s, \ell)) \equiv match(t, s, \ell) = \{b\}, b \in \mathcal{B}$$

We will find it useful, for transformational maintenance, to capture the potential application of a transform at a specified location. We call this potential application a *transformation*. The intuitive distinction between transform and transformation is the same as the distinction between the AI notion of *operator* and *operation*.

DEFINITION 3.11: *Transformation.* A pair $\langle t, \ell \rangle$ with $t \in \mathcal{T}$ and $\ell \in \mathcal{L}$, denoted t^ℓ.
□

We define the notation $t^\ell(s) \equiv apply(t, s, \ell)$. We use the notation $\mathcal{X}$ to mean the set of possible transformations $\mathcal{T} \times \mathcal{L}$, with x representing individual transformations.

Often a transform is represented by some concrete object r from an arbitrary representation set $\mathcal{R}$, and a general transform-constructing mechanism $\Theta : \mathcal{R} \rightarrow \mathcal{T}$ embedded in the transformation system is used to construct the actual transform from its representation when it is applied. We abuse the notation and write t_{name} or t_r to stand for some $\Theta(r_{name})$, or even r_{name} itself if we have a specific transform in mind. We will also write $t : \mathcal{S} \times \mathcal{L} \rightarrow \mathcal{S}$ when defining a particular transform, the states it manipulates, and the form of the locaters.

Examples of transforms, transformations, and locaters

To show the generality of the definitions, we present some example transforms and locaters, followed by discussion of a variety of representations used in systems that we would call transformation systems.

The string production with nonterminals:

$$t_{AbcA \Longrightarrow cAb} : \mathbf{strings} \times (\mathbf{Nat} \rightarrow \mathbf{Nat}) \rightarrow \mathbf{strings}$$

The subscript on t is a representation in this example. The symbol $\Longrightarrow$ is read as "transforms to". A state in this case is a particular string. A locater is a map from origin-one indices of symbols in a string to be rewritten to the indices of the symbol in the lefthand pattern side of the production, indicating which string symbols match which pattern symbol; string symbols not matching the production are mapped to a fixed value, say, zero. As an example, the locater

$$\ell_{61} = \{1 \rightarrow 0, 2 \rightarrow 1, 3 \rightarrow 1, 4 \rightarrow 2, 5 \rightarrow 3, 6 \rightarrow 1, 7 \rightarrow 1, 8 \rightarrow 0\}$$

ensures

$$t_{AbcA \Longrightarrow cAb}(\text{“}bzzbczzq\text{”}, \ell_{61}) = \text{“}bczzbq\text{”}$$

The tree transform:

$$t_{distribute-multiply} : \mathbf{Tree} \times \mathbf{Path} \rightarrow \mathbf{Tree}$$

with its representation being the tree rewrite:

$$r_{distribute-multiply} \equiv ?a * (?b + ?c) \Longrightarrow ?a * ?b + ?a * ?c$$

where states consist of expression trees.

Locaters for tree transforms are *paths*, a sequence of integers i selecting successive one-origin subtrees [BEH+87] to select the point of proposed application of the transform, expressed as a sequence $\langle i_1, i_2, \cdots \rangle$.

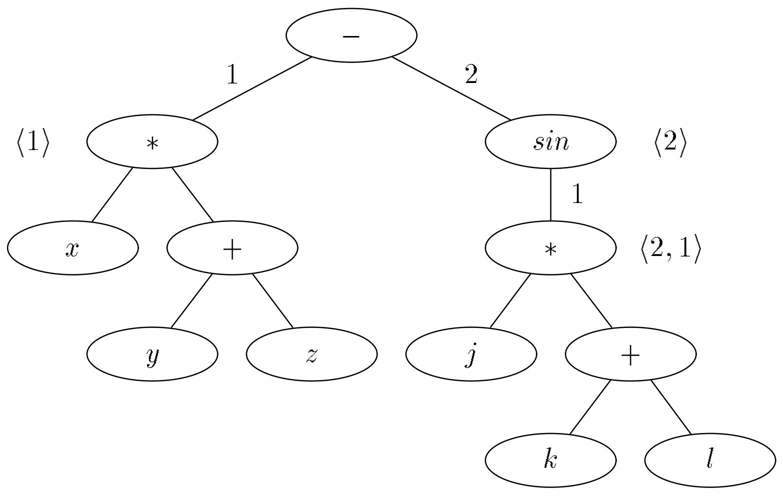


Figure 3.2: A tree program and some locaters

As an example,

$$t_{distribute-multiply}(x * (y + z) - sin(j * (k + l), \langle 2,1\rangle) = x * (y + z) - sin(j * k + j * l)$$

Figure 3.2 shows the program before transformation and the locater used.

We shall use tree transforms in other examples later. When we define a tree transform and specify a locater simultaneously, we write the transform representation followed by a locater value:

$$(treepattern \Longrightarrow treereplacement)\ @\langle path\rangle$$

We typically drop the outer parentheses:

$$?a * (?b + ?c) \Longrightarrow ?a * ?b + ?a * ?c@\langle 2,1\rangle$$

or, if we use a known tree transform,

$$t_{distribute-multiply}@\langle 2,1\rangle$$

A conditional tree rewrite:

$$t_{useless-WHILE} : \langle \mathbf{Eqn}, \mathbf{Facts} \rangle \times \langle \mathbf{Path}, \mathbf{Nat} \rangle \rightarrow \langle \mathbf{Eqn}, \mathbf{Facts} \rangle$$

with its representation being:

$$\mathbf{while}\ ?x\ \mathbf{do}\ ?y \Longrightarrow \mathbf{skip} \quad \mathit{if}\ ?x = \mathit{false}$$

States consist of a directed acyclic graph representation of a program coupled with a fact database resulting from symbolic execution. The locater consists of a pair, consisting of a **Path** as defined in the last example, and an index into a fact database specifying which fact justifies the conditional in the rewrite.

The theory morphism:

$$t_{stack-to-LISP} : \mathbf{Term} \times \mathbf{Any} \rightarrow \mathbf{Term}$$

represented by

$$\{top \Longrightarrow car, pop \Longrightarrow cdr, push \Longrightarrow cons, empty \Longrightarrow nil\}$$

which maps states consisting of a term from a simple stack algebra to a term in an algebra defining a version of LISP, by mapping the individual operations in the stack algebra to operations in the LISP algebra. In this case, a locater is unnecessary, as the transform is applied to the entire state:

$$t_{stack-to-LISP}(top(push(x, empty))) = (cdr(cons(x, nil)))$$

LALR parser generators:

$$t_{LALR-parser-generator} : \langle \mathbf{Domain}, \mathbf{Term} \rangle \times \mathbf{Any} \rightarrow \langle \mathbf{Domain}, \mathbf{Term} \rangle$$

which maps BNF-style syntax equations into parser tables. The **Domain** is a tag indicating to a potential $p_{meaning}$ how to interpret the particular term. The locaters in this case are also ignored.

Discussion: Having seen a few examples, we now describe a range of schemes used for transforms.

A typical representation for a transform is a tree-to-tree rewrite [SHFN76, Kib78, Nei84a], with pattern variables used as scheme parameters. This choice is common apparently because many programming languages are easily parsed into abstract syntax trees, encouraging use of such abstract syntax trees as specific representations for programs. Often such rewrites are augmented with an applicability condition to form *conditional rewrites* [CHT81, Rea86, Hec88, BMPP89]. The combination of a tree-rewriting mechanism with a particular tree-to-tree rewrite and locater instance form the actual map from programs to programs. The representation for transforms is generally compatible with the representation method used to encode the programs. Graph-to-graph transforms have been used [Nag78, Ehr78, vdB81, Sow84, YNT86, Wil87, YNTL88, PP89] because of the ability of graphs to syntactically express semantic relations poorly expressed by trees, such as symbol definitions, control flow graphs and shared entities. More specialized directed acyclic graph-rewriting systems [HKP87] have been proposed to gain the advantage of shared entities without the pattern-matching costs necessary for full graph matching. String-to-string transforms are theoretically useful [Pos43] but don't seem to be used much for transformation systems, probably because tree rewriting is almost as easy to implement and is more natural for expressions.

In practice, the representations of values for locaters are, like the transforms, usually dependent on the representation structure of the programs. For tree-to-tree rewrites, locaters can be paths, as defined earlier in the examples. Such locaters can be used to generate bindings of values to pattern variables; the relation between nodes of the pattern tree to nodes in the subtree being revised is determined uniquely by a path. For graph transforms, bindings can be specified by an injective map from nodes of the pattern graph to the program graph [Ehr78]. Locaters for graph transforms are problematical at this time, but a sequence of graph patterns to select successively smaller subgraphs may work.

There are classes of transforms that affect the entire state. While not usually found in conventional transformation systems, parser generators (such as LALR parser generators like YACC [Joh80]) constitute transforms by our definition, which map BNF-like programs into parsing tables, as well as data-flow analyses [ASU86] which simply augment the set of cached consequences. So, too, do *theory morphisms*, useful for mapping terms (a particular representation for programs) defined in one algebra into terms defined in another algebra. For such transforms, the corresponding transformations have effectively trivial locaters.

Although our simple characterizations of transform, binding, locater and transformation are satisfactory for this thesis, these notions are considerably more complex, especially if one is interested in the actual mechanics of rewriting and how those mechanics are related to the representation of the programs being transformed. Some initial investigation of the foundations of rewriting is being pursued by [Sri91].

3.1.8 Property-preserving versus Non-property-preserving transforms

Transformation systems usually operate by applying so-called "correctness-preserving transforms"[9]. Using this terminology, transformational maintenance is often initiated by applying non-correctness-preserving transforms; more on this in Chapter 6. Consequently we shall have need for definitions for both.

Partsch [PS83] generalizes the notion of "correctness-preserving" by defining a transform to be correct if a certain, arbitrary [BPPW80], transitive semantic relation[10] holds between its input and output, i.e., there is a fixed, pre-determined relation $\rho(f, t^{\ell}(f))$. Bauer defines a (correct) transform to be an inference rule [BEH+87, pp. 30-31] concluding such a semantic relation. As these definitions of correctness-preserving are imprecise with respect to what a correctness-preserving transform is correct, we prefer instead to define *property-preserving transforms* in terms of preserved properties as follows:

Definition 3.12: *G_i-preserving transform c.* Relative to a performance predicate G_i, any transform $c \in \mathcal{T}$ with the property:

$$\forall s, \ell : \mathit{defined}(c^{\ell}(s)) \supset (G_i(s) \supset G_i(c^{\ell}(s)))$$

The set of property-preserving transforms with respect to G_i is denoted[11] $\mathcal{C}_i$. □

[9]An interesting difficulty with this formulation is the notion of *approximate* transforms. Useful transformation systems have been built with transforms that are not provably correct, but are intuitively close to correct; this is analogous to floating point arithmetic versus the mathematical notion of real arithmetic. Kant [DKMW89] describes a system that implements finite differencing programs for solving partial differential equations by replacing continuous functions with approximations. Perhaps such transforms can be treated as merely missing the additional conditions which make them correct, although such conditions may be difficult to define in practice.

[10]Our performance predicates g are only defined on a single *program*, not a pair of *programs*.

[11]We chose the letter c to stand for property-preserving transforms because of their historical association with the term *correctness*-preserving transforms.

If $G_{i,j} \equiv p_i(s) \succeq v_{i,j}$, then a transform c is $G_{i,j}$-preserving if:

$$\forall s, \ell : \mathit{defined}(c^\ell(s)) \supset ((p_i(s) \succeq_i v_j) \supset (p_i(c^\ell(s)) \succeq_i v_j))$$

One way to ensure this is to require the transform c to be monotonic in p_i:

Definition 3.13: *p_i-monotonic transform t.* Relative to a performance measure p_i, any transform $t \in \mathcal{T}$ with the property:

$$\forall s, \ell : \mathit{defined}(t^\ell(s)) \supset p_i(t^\ell(s)) \succeq_i p_i(s)$$

□

Each p_i-monotonic transform c is $G_{i,j}$-preserving for all j.

Definition 3.14: *p_i-preserving transform c.* Any p_i-monotonic transform c. □

Using this terminology, the conventional definition of correctness-preserving transform is simply $p_{meaning}$-preserving.

A significant advantage obtained from the notion of p_i-preserving is the identification of the set of $\mathcal{C}_i$. If p_i is a frequently used performance measure, such as $i = \mathit{meaning}$ or $i = \mathit{complexity}$, sets of transforms that preserve those properties can be identified in advance of transformational implementation.

The definition of monotonicity is motivated by the definition of *refinement* provided by Sanella [ST88], in which an algebra specification f_a is defined to be an implementation of algebra specification f_b if $\mathit{models}(f_a) \subseteq \mathit{models}(f_b)$. If one is implementing an algebra-specification transformationally, then an algebra-specification is a program, and p_{models} is a performance measure by our definition. The "$\subseteq$" idea provided the inspiration for subsumption[12]. This formulation covers neatly the notion of implementation of [TM87], in which implementations are super-theories.

Defining property-preserving transforms automatically determines a complementary set of non-property-preserving transforms:

Definition 3.15: *Non-property-preserving transform n.* Any $n \in (\mathcal{T} - \mathcal{C}_i)$. □

Figure 3.3 provides an overview of the relation between various basic aspects of a transformation system.

With the basic concepts defined, we are now ready to consider transformation systems as a whole.

[12]Thanks to Y. V. Srinivas for a hint along these lines.

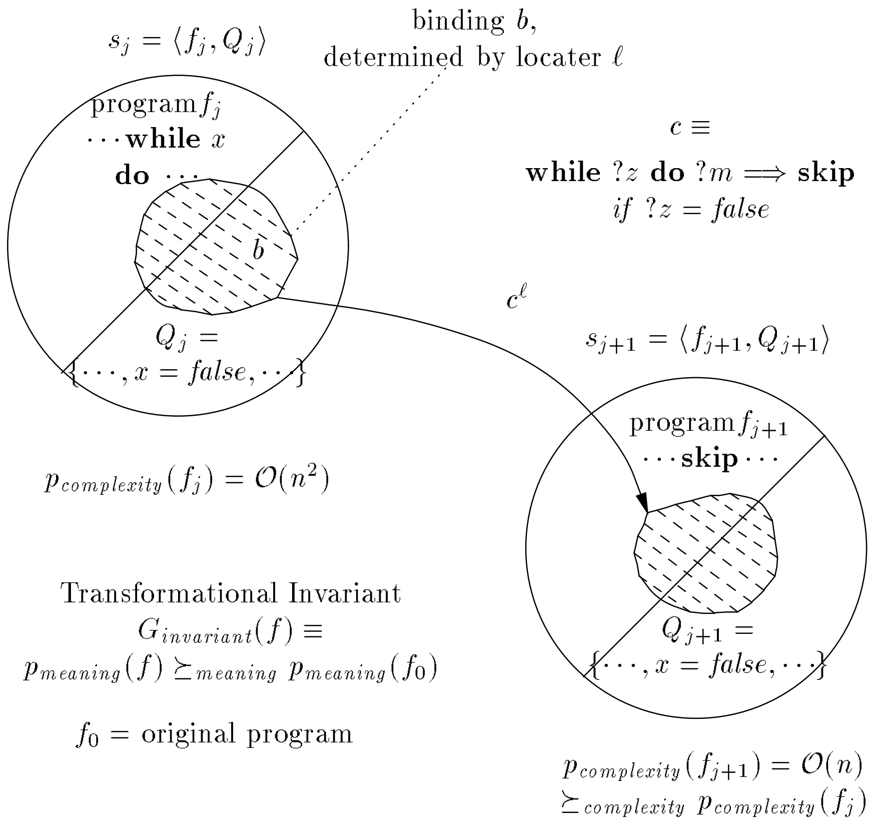

$s_j = \langle f_j, Q_j \rangle$
binding b,
determined by locater ℓ
programf_j
$\cdots$ **while** x
do $\cdots$
b
$Q_j =$
$\{\cdots, x = \mathit{false}, \cdots\}$
$c \equiv$
while $?z$ **do** $?m \Longrightarrow$ **skip**
if $?z = \mathit{false}$
c^ℓ
$s_{j+1} = \langle f_{j+1}, Q_{j+1} \rangle$
programf_{j+1}
$\cdots$**skip**$\cdots$
$Q_{j+1} =$
$\{\cdots, x = \mathit{false}, \cdots\}$
$p_{complexity}(f_j) = \mathcal{O}(n^2)$
Transformational Invariant
$G_{invariant}(f) \equiv$
$p_{meaning}(f) \succeq_{meaning} p_{meaning}(f_0)$
$f_0 =$ original program
$p_{complexity}(f_{j+1}) = \mathcal{O}(n)$
$\succeq_{complexity}\ p_{complexity}(f_j)$

Figure 3.3: Notational overview for transformations mapping states

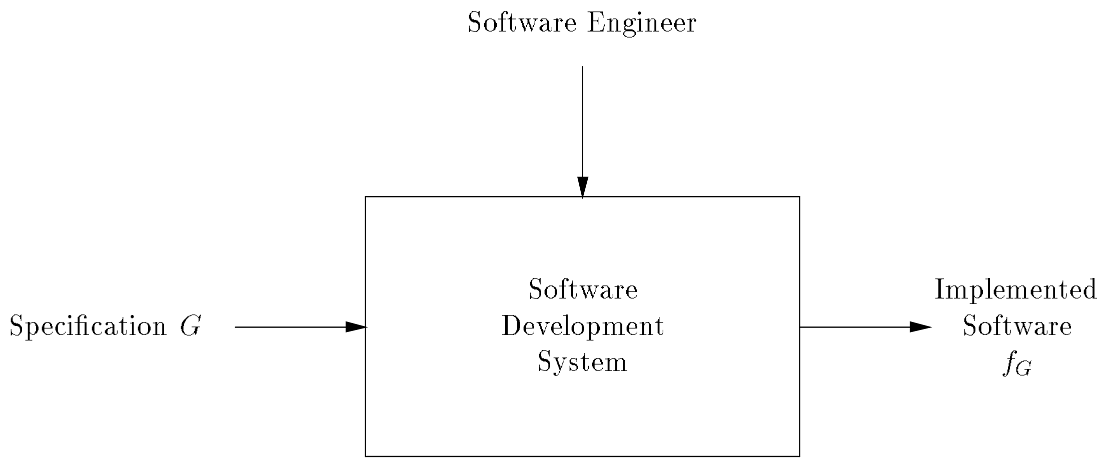


Figure 3.4: Model of development symmetric with respect to specification G

3.2 Model of a Transformation system

In this section we provide a model of transformation systems using the basic concepts developed. We first characterize general software development systems to provide a reference used to highlight a characteristic feature of transformation systems. Our model of a transformation system is then described and analyzed. We discuss how a practical asymmetry in its operation leads to use of particular specification styles. We close with a comparison of the model to existing transformation systems.

3.2.1 Purpose of a Software Development System

A software development system (SDS) (Figure 3.4) is intended to somehow produce a program meeting its specification, that is, is to find a program f that satisfies the specification G:

$$\exists f : G(f)$$

So we define:

DEFINITION 3.16: *Implementation.* A program f_G that satisfies a specification G. □

This characterization does not give any leverage to the software development process. About all one can do to construct an implementation (find an f_G) with this level of detail is to naively enumerate programs for f_G and apply the predicate G until an one is found that passes. Such a naive SDS treats the usual internal structure of the specification,

$$G = G_1 \wedge G_2 \wedge G_3 \cdots$$

in a completely symmetric fashion: none of the individual performance goals G_i has any special status over any other performance goal. Every proposed implementation f_G must be tested against all of the performance goals to ensure that all are satisfied.

The problem of determining how to obtain an implementation efficiently is called the *control* problem. We shall discuss this in more detail in Chapter 4, but transformation systems have a basic, low-level control mechanism built in from the start, which we discuss in the next section.

3.2.2 Actual Transformation system Model

In this section, we describe the purpose and mechanism of a transformation system.

Transformation systems are software development systems with a sophisticated approach to generating the implementation. They have a peculiar, but practical, asymmetry about how they handle the individual performance goals. Such asymmetry can lead to faster operation of the the software development system; this asymmetry is precisely what distinguishes a transformation system from a blind enumeration.

A transformation system (Figure 3.5) accepts a specification (consisting of a program and some performance specifications), and attempts to find a sequence of transformations to apply to a program representing one aspect of the specification, such that the transformed program satisfies the balance of the specification. The transformed program is output as the implemented software.

DEFINITION 3.17: *Transformation system.* Any mechanism which applies a sequence of property-preserving transforms to a given program scheme to find an implementation[13]. □

[13]We distinguish *Synthesis Systems* as those which accept *only* a specification and invent a program to meet that specification; thus synthesis systems are more ambitious than transformation systems. It is our expectation that they are also much harder to implement, hence our concentration on transformation systems. Practical tools along these lines may actually end up being hybrids, similar to REFINE [SKW85] as augmented by KIDS [Smi89].

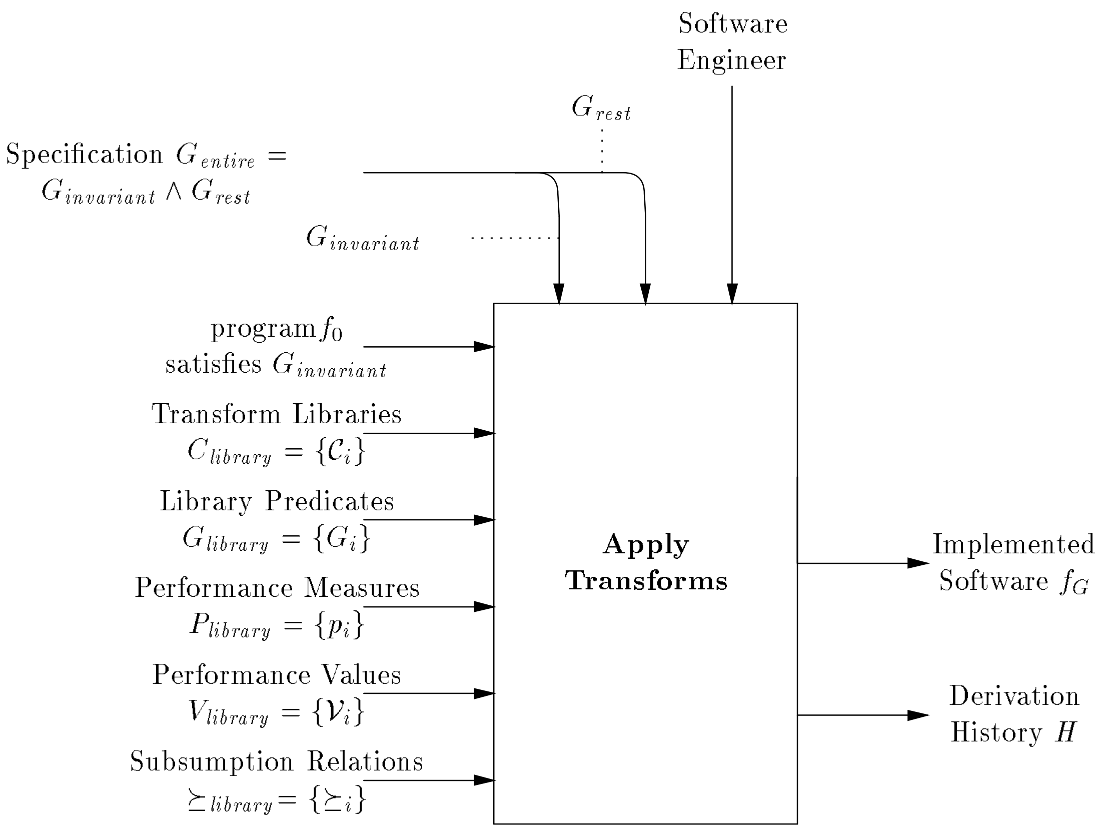

Software Engineer
G_{rest}
Specification $G_{entire} = G_{invariant} \wedge G_{rest}$
$G_{invariant}$
program f_0 satisfies $G_{invariant}$
Transform Libraries $C_{library} = \{\mathcal{C}_i\}$
Library Predicates $G_{library} = \{G_i\}$
Performance Measures $P_{library} = \{p_i\}$
Performance Values $V_{library} = \{\mathcal{V}_i\}$
Subsumption Relations $\succeq_{library} = \{\succeq_i\}$
Apply Transforms
Implemented Software f_G
Derivation History H

Figure 3.5: Model of transformation system

A transformation system may optionally produce the sequence of transformations used, called a *derivation history*. This sequence will be of value when attempting to construct a modified implementation. We defer further discussion of this output until Chapter 5.

Other inputs used by the transformation system are a set of performance predicates and performance measures, which provide a vocabulary for stating the program specification and actual functions used to extract qualities of states, and a *transformation library* of property-preserving transforms C, indexed by the goals they preserve. Control knowledge is also needed, but we will discuss that in Chapter 4.

The generation of the transformation sequence is difficult enough so that we expect a human designer (called the Software Engineer) to provide assistance to the transformation system essentially in the form of breaking ties between possible choices the transformation system might face. If the grand dream of transformation systems is achieved and the process is nearly automatic, then human assistance will be small but nontrivial [Bal85a]; in the meantime, human assistance may provide the bulk of the direction with the transformation system merely handling clerical detail [BEH$^+$87, BMPP89] such as listing choices and applying the transformations mechanically.

Like any SDS, the problem for a transformation system is to find a program f such that $G_{entire}(f)$. The specification G_{entire} is assumed to be decomposable into smaller specifications of which we have some knowledge. Frequently G_{entire} takes the form[14]:

$$G_1(f) \wedge G_2(f) \wedge G_3(f) \wedge \cdots$$

Each transformation system defines a decomposition of G_{entire}:

$$G_{invariant}(f) \wedge G_{rest}(f) \supset G_{entire}(f)$$

The particular performance goal $G_{invariant}$ chosen[15] is such that an initial approximation f_0 of the desired program $f_{G_{entire}}$ can be divined from the structure of $G_{invariant}$, by a mechanism outside the transformation system. This has the nice effect that now one of the terms, $G_{invariant}(f_0)$, of the implicant is satisfied; the transformation system need only find out how to satisfy the rest (obviously, if we have only a single predicate, then the implementation process is complete). The state of the transformation system is initialized to $\langle s_0, G_{invariant}, G_{rest}\rangle$, where $s_0 = \langle f_0, \emptyset\rangle$ is the initial design state consisting of the initial program scheme and the empty set of consequences. It remains for the transformation system to find a way to satisfy G_{rest} by transforming s_0. $G_{invariant}$ is called the *transformation invariant.*

For each performance goal G_i, there is a set of property-preserving transforms $\mathcal{C}_i$. The task of the transformation system when in state $\langle s_0, G_{invariant}, G_{rest}\rangle$ is to choose sequences of members of $\mathcal{C}_{invariant}$ with appropriate locaters, $c_1^{\ell_1}, c_2^{\ell_2}, \cdots, c_k^{\ell_k}$, such that $G_{rest}(c_k^{\ell_k}(c_{k-1}^{\ell_{k-1}}(\cdots(c_1^{\ell_1})(s_0))))$ is true. By applying only members of $\mathcal{C}_{invariant}$, the property $G_{invariant}$ cannot be lost[16] and therefore need not be continually tested.

[14]Should a specification be disjunctive, the transformation system can simply treat it as multiple conjunctive specifications, and an implementation need only be found for one.

[15]Current transformation systems each have a fixed decomposition method. In principle, there is no reason why this decomposition cannot be dynamic, but there is little evidence that such an approach has been tried. Such a dynamic decomposition would provide an additional backtrack point in case the transformation process failed on the initial decomposition, a luxury present transformation systems do not have.

[16]One could apply some chain of non-property-preserving transforms $n_1, n_2, \cdots$ to s_j at any point, provided that the composition Πn_i is property-preserving. In practice this is rarely done, as few such sequences appear to be interesting; further, if the composition is a property-preserving transform and interesting, one can expect it to be present in $\mathcal{C}_i$. One can expect groups of non-property-preserving transforms to be applied in transformation systems in which the representational constraints on

```
function $TransformationSystem(program{:}s,\ Predicate{:}G_{invariant},$
                                    $Predicate{:}G_{rest},\ Transforms{:}\mathcal{C}_{invariant})$
    returns $program$
  while $G_{rest}(s)=false$ do
    choose $c_i \in \mathcal{C}_{invariant}$
    choose $\ell \in \mathcal{L}$ suchthat $defined(c_i^{\ell}(s))$
    $s \leftarrow c_i^{\ell}(s)$
  end while
  return $s$
end
```

Figure 3.6: Simple model of transformation system

For a complicated $G_{invariant}$, or for large-scale states s, this can be a big computational savings. This is a key architectural feature of transformation systems: staying on a plateau of achieved property by only applying property-preserving transforms[17].

Typically, a transformation system will be manipulating a state which implicitly defines a computation ($G_{implicit} \equiv G_{meaning,j}$), and applying computation-preserving transforms $c \in \mathcal{C}_{meaning}$ in an attempt to find a state in which the other desired performance goals, such as the amount of code $G_{sloc,1000}$, are achieved. However, it is perfectly reasonable for the state to initially describe a program which consumes a desired amount of code space (satisfies $G_{sloc,1000}$), and apply code-space-preserving transforms $c \in \mathcal{C}_{sloc}$ to see if the desired computation is achieved.

transforms prevent one from directly stating a desired property-preserving transform. An example of such a system is TI [Bal85a], whose transforms are coded in a language called PADDLE [Wil83], in which a property-preserving transform is achieved by a series of structural "edits" applied to the syntax tree representing a program. Such systems should require explicit notions of sequencing and *atomicity of transactions* to ensure that all the non-property-preserving transforms are either applied in the correct order, or not applied, as a group. In PADDLE's case, the need for atomicity is not explicitly acknowledged, but is implicitly present due to the execution rules for PADDLE, and usually indicated by the transform designer by grouping the non-property-preserving transforms into a single syntactic PADDLE entity like a procedure. In any case, our model does not cover grouped non-property-preserving transforms; one must assume a corresponding property-preserving transform.

[17]If one could constrain specifications to a purely conjunctive form, and could easily compute intersections of sets of property-preserving transforms for different performance measures used by the performance goals, then one might be able to do true hill climbing in the design space. This would be accomplished by moving up to each plateau as successive goals were achieved, and constraining the applicable property-preserving transform set to be the intersection of all property-preserving transform sets for the remaining goals.

A transformation system can theoretically switch from using one type of property-preserving transforms to another, by changing which performance goal is preserved. Assuming that at step m, for some G_j and G'_{rest}:

- the transformation system state is $\langle s_m, G_{invariant}, G_{rest}\rangle$,
- $G_j \wedge G'_{rest} \supset G_{invariant} \wedge G_{rest}$,
- $G_{invariant}(s_m) = true$ and $G_j(s_m) = true$

then the transformation system can change its focus of attention from $G_{invariant}$ to G_j by changing the state to $\langle s_m, G_j, G'_{rest}\rangle$. When the form of G_{rest} and G'_{rest} are conjunctions of individual performance predicates, this amounts to swapping a single predicate from the invariant slot with a predicate in the to-be-achieved slot. Further transforms must then be chosen from $\mathcal{C}_j$ rather than $\mathcal{C}_{invariant}$. The utility of this can be seen when designing time-critical routines for operating systems; it can be more efficient to enumerate routines having extremely tight constraints if there are only a few such routines. No systems familiar to this author change the focus in this manner; this is not surprising since doing so would require $G_{invariant}$ to be explicit, whereas it almost always implicit.

A complication occurs in practice. The sets $\mathcal{C}_i$ are not actually available to the transformation system. Instead, for each set $\mathcal{C}_i$, there is an *approximating* set of transformations C_i. The transformation system uses C_i whenever $\mathcal{C}_i$ is desired. This approximation occurs as a consequence of human fallibility in the construction of the transform set; the supply of transforms is nearly the desired set, but may include some faulty transforms or be missing some needed transforms. This approximation requires occasional action be taken to bring the sets C_i more in line with their ideal $\mathcal{C}_i$. Similarly, we expect all the reference inputs used by the transformation system (G_i, p_i, $\mathcal{V}_i$, $\succeq_i$) to only be near approximations of the truth. Correction of these approximations is one source of evolutionary pressure.

3.2.3 Partitioning the specification

Now let us consider how the initial partitioning of the specification G occurs. Transformation systems must operate on a program, but specifications technically only come as a set of goal predicates. Somehow an initial program f_0 must be manufactured.

Such an f_0 can be obtained by use of a *synthesis system*. Given a specification of the form $G_{meaning} \wedge G_{rest}$, transformation systems containing synthesis subsystems (such as KIDS [Smi89]) use the synthesis subsystem to generate f_0 from the supplied $G_{meaning}$; $G_{invariant}$ then becomes the specification $G_{meaning}$, and the transformation system is started in state $\langle\langle f_i, \emptyset\rangle, G_{meaning}, G_{rest}\rangle$, effectively partitioning the specification.

However, most existing transformation systems finesse the need to decompose the initial specification at all, by assuming a particular $G_{invariant}$ and requiring the specifier to supply the mixed specification $\langle f_0, G_{rest}\rangle$. Consistent with the definition of base specifications, the invariant is defined as:

$$G_{invariant}(f) \equiv p_{base}(f) \succeq p_{base}(f_0)$$

in which p_{base} need not be made explicit, as $G_{invariant}$ never needs to be evaluated by the transformation system. This indirectly defines the set of usable transforms to be the p_{base}-preserving transforms; this set can be constructed in advance of seeing the specification since p_{base} is a constant for the transformation system. Usually, p_{base} is defined to be a semantic function like $p_{meaning}$. This is characteristic of systems with both wide-spectrum language approaches [BBG+78] such as REFINE [Rea86, SKW85], CIP-L [BMPP89], or GIST [Bal85a, Sca86], and domain-specific approaches like Draco [Nei80, Nei84a]. An apparent advantage for the implementors of the transformation system is that $p_{meaning}$ need not be explicitly defined. In practice, leaving $p_{meaning}$ implicit/informal can make determining property-preserving transforms $\mathcal{C}_{meaning}$ difficult, and for domain-oriented transformation systems, can lead to *semantic earthquakes* [Bax86].

3.2.4 Comparison of transformation model to real systems

This model of transformation systems is a synthesis of the best features of a number of existing systems. CIP [BMPP89] has programs with scheme variables. LIBRA [Kan81] and MEDUSA [McC88] have explicit performance measures; some work has been done on an efficiency analyzer for KIDS [Smi89]; MEDUSA allows only a single performance bound, computation complexity, to be explicitly specified. All other transformation systems mentioned in this section have implicit performance goals. A base specification f_0 is supplied to the TAMPR [BM84], Draco [Nei84a], LIBRA, TI [Bal85a], CIP, MEDUSA, and REFINE [Rea86, SKW85], CIP-L systems; the CYPRESS synthesis subsystem [Smi85] of KIDS being the exception, accepting $G_{invariant}$, and manufacturing f_0 for processing by REFINE. Every transformation system considered had sets of property-preserving transforms defined only for the $p_{base} \equiv p_{meaning}$ implicit in the programs manipulated.

The practice of supplying *only* a program as a specification has the dubious advantage of not requiring any performance properties or performance predicates to be defined to that transformation system before it is used to apply transformations. We believe that this absence of performance predicates is a key stumbling block when it comes to providing transformation systems with control mechanisms. In fact, since extant systems are almost never provided a G_{rest}, it is hard to understand why any transformations are applied at all (after all, $G_{invariant}$ is satisfied by definition) until we realize that these systems must have an implicit $G_{rest} \equiv G_{implicit}$, such as the REFINE transformational compiler[18]. A minor variation is the TI system [Bal85a], in which G_{rest} is not implicit in the transformation system itself, but is implicit in the control supplied to the transformation system in the form of a set of PADDLE [Wil83] procedures. A third variation is a purely interactive transformation environment such as CIP [BMPP89], in which $G_{rest} \equiv G_{designer}$, as the designer looks at intermediate states, and if he is not satisfied with their performance, directs applications of further transformations. Lastly, there are hybrid semi-interactive systems such as KIDS [Smi89], which have $G_{rest} \equiv G_{implicit} \wedge G_{designer}$ with the implicit performance goals being inherited from the underlying transformation system (in this case, the REFINE compiler). The only advantage to implicit performance specifications is that they need not be formalized; in the long run, we think this benefit is not as great as the need to provide control for navigation.

A model of MEDUSA

Our model is probably best exemplified by the MEDUSA system [McC87], which "synthesizes" routines to solve planar intersection problems from computational geometry, given complexity constraints. A mixed specification $\langle f_0, v_{complexity}\rangle$ is provided to MEDUSA. $G_{invariant}(f) \equiv p_{meaning}(f) \succeq p_{meaning}(f_0)$ is assumed; $p_{meaning}$ is never instantiated or computed, and G_{rest} is defined as $G_{complexity}(f) \equiv p_{complexity}(f) \succeq_{complexity} v_{complexity}$. f_0 itself is defined by a single typed scheme variable representing the desired function by being typed (named) for that function; we will call this a *placeholder*. Transforms are replacements of placeholders representing functions by code skeletons implementing those functions; the code skeletons may in turn contain placeholders. The set of placeholder-to-skeleton maps defines the set of property-preserving transforms $\mathcal{C}_{meaning}$ which are $G_{invariant}$-preserving.

Each MEDUSA transform c_i is associated with a cost formula for computing $p_{complexity}(c_i)$ from $p_{complexity}$ values for the instantiations of placeholders in c_i. Starting with a symbolic complexity $v_{complexity,0}$ for f_0, a lower bound estimate on $p_{complexity}$ for any f_i can be incrementally maintained by substitution of the cost formula of c_{i-1}. Monotonicity of growth of the lower bound on $p_{complexity}$ as components are refined occurs as a consequence of the purely functional nature of all of MEDUSA's programs,

[18]Unsurprisingly, [SA89, 104] found that most design decisions involved in a number of documented algorithm syntheses were based on non-functionality properties, what we call performance properties, and that such performance goals were not explicitly represented

the absence of any transforms that can simplify away code, and the compositional nature its transforms.

This monotonic growth allows MEDUSA to cut off an overly expensive implementation early by using $\succeq_{complexity}$. Whenever a proposed c_i^ℓ has the property that $p_{complexity}(c_i^\ell(f_i)) \not\preceq v_{complexity}$, no further property-preserving transforms can lower the cost, and thus application of c_i^ℓ is useless and can be ignored. This is MEDUSA's main hueristic, and is quite close to our earlier characterization of stepping up plateaus of increasing partial specification satisfaction by restricting sets of allowable property-preserving transforms.

Having defined a model of transformational implementation, we now turn our attention to properties of the transformational implementation space.

3.3 Properties of the design space

The *design space* (Figure 3.7) is the set of possible implementations determined by application of property-preserving transforms to the initial program, assuming the transformation invariant is held constant. Design states are shown as circles. The initial state $s_0 = \langle f_0, \emptyset \rangle$ forms the root of the space. Each arrow represents a particular transformation from one design state to another; the number of transformations leaving any particular state is widely variable, depending on the state and on the set of available property-preserving transforms. We call the average number of transformations leaving a design state the *branching factor*. The nodes along the bottom represent possible implementations (although implementations may be "interior" to the space), each satisfying a different set of performance goals. The bold arrows represent a particular *derivation history*, the string of transformations leading to a particular implementation satisfying some set of performance goals.

There are several interesting observations about the nature of typical design spaces:

- The average derivation history is quite long: 10^4 transformations.
- The design space is (consequently) enormous: 10^{43} states.
- There tend to be cross-links, or multiple paths, to many of the states in the space; for very small spaces, the number of paths [19] is at least 10.

[19]See the discussion in Chapter 7.

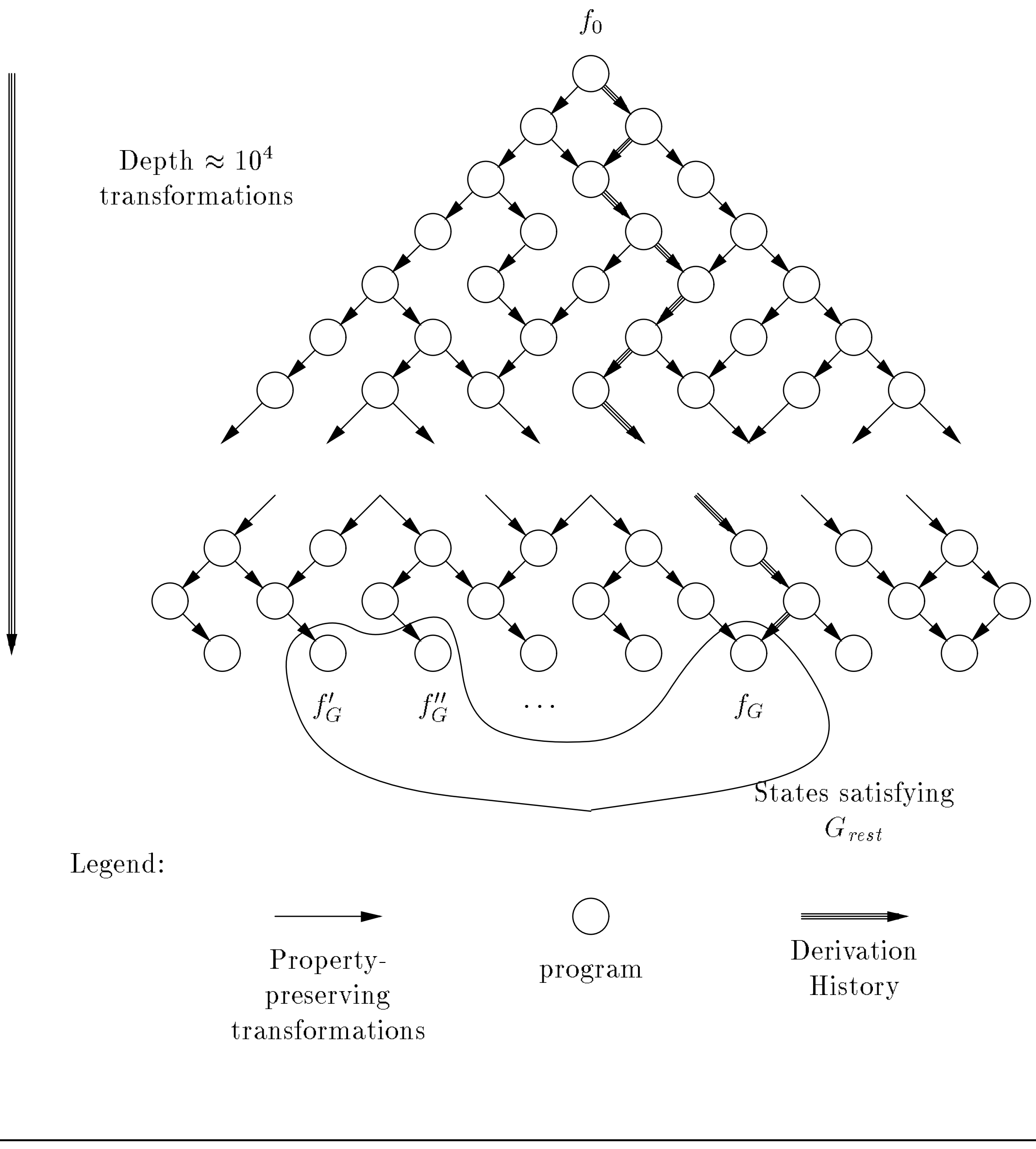

f_0
Depth $\approx 10^4$ transformations
f'_G
f''_G
$\cdots$
f_G
States satisfying G_{rest}
Legend:
Property-preserving transformations
program
Derivation History

Figure 3.7: Huge design space with multiple solution paths

3.3.1 Long chains of transformations are required

Experience suggests that even for moderate size specifications (represented by moderate size initial programs f_0) a large number of transformations are required to locate an implementation, Figure 3.8:

- Goldberg [Gol89] reports that a very small functional specification for a topological sort requires roughly 40 transformation "steps" to get an efficient implementation. The steps are what Lowry [LD89, p. 285] calls "large grained rules", which are really metaprograms (packages) of what we define as transforms, so we think this value is low by an order of magnitude.
- The TAMPR system reportedly used about 10,000 steps to transform a 1300 line functional LISP program to FORTRAN [Boy84].
- Porting the core of the Draco system (roughly 2400 lines of LISP source code) transformationally from a DEC20 to a VAX [ABFP86] requires roughly 40,000 transformations [Bax87b]. Since both Boyle's conversion and our porting reimplement an already mostly-implemented (procedural specification) system (as opposed to implementing a really abstract specification) it should be clear that this number of transformation steps is conservative.
- Barstow [Bar88, Bar89] estimates that he will need to apply 10,000 transformation steps[20] to implement a 500 line specification. The transform application sequence is specified by a manually-generated script.
- Paulson [Pau87, p. 10] reports on two verification projects: verifying an associative memory unit (AMU) and the Viper microprocessor[21]. The AMU proof (control) took 30 part-time man months to develop, occupied 4800 lines of ML, and required 10.5 hours of computer time to validate. Verification of the Viper microprocessor required "a million primitive inferences".

These results are consistent with Wile's [Wil83] comment that the problem of *scale* is one of the most critical to handle.

[20]It is surprising to observe the consistency (same order of magnitude) in the number of transformation steps required for specs in the multiple-thousand line case. Perhaps there is some property of the level of specification language, or its distance from an implementation, that leads statistically to such a result?

[21]While verification proofs and transformational implementation do not produce the same results, both types of tools do considerable numbers of rewrites, so we feel justified to use the statistics here. Both technologies require reuse of the derivation history to make them practical.

Who	What	Specification size	Transformations
[Gol89]	topological sort	$<$ 10 lines REFINE	40 large-grain
[Boy84]	TAMPR system	1300 lines functional LISP	$\approx$ 10000
[Bax87b]	Draco system port	2400 lines abstract LISP	$\approx$ 40000
[Bar89]	Device control	500 lines Φ-lang	$\approx$ 10000
[Pau87]	assoc mem unit	4800 lines ML	10.5 cpu hours
[Pau87]	Viper verification	unstated	10^6 inferences

Figure 3.8: Reported costs on transformational implementation

3.3.2 The design space is enormous

Let us consider Barstow's Φ-NIX system [Bar88, Bar89] as an example. Assume, with extreme optimism, that an average branching factor of 1.01 was achievable[22]. Given that $10,000$ transformations are expected to produce an implementation from a specification [Bar88], we compute that roughly $1.01^{10000} = 1.6 \times 10^{43}$ possible sequences of applications of transformations to the original programs leading to possible implementations, of which some are acceptable. Clearly, a complete exploration of this space is impossible. Some means for navigating the space is a necessity.

We discuss navigation and control in Chapter 4.

3.3.3 Multiple paths between connected states

A specification determines a family of possible implementations depending on individual design decisions (each transformation representing a decision to apply it); an overly simplistic view is that there is a single path from the initial state to a particular implementation leading to a tree-like view of the implementation space [Par76, GB78, KB81]. In practice, however, *for each path between a pair of states, it is likely that there are alternative paths between the same states* [SA89] (Figure 3.7).

This multiple-path-to-a-state property is a consequence partly of the algebraic properties of the transforms, but it is mostly due to the sheer size of the state. The typically large "diameter" of a state allows pairs of transformations over parts of the state separated by some distance to trivially commute [Bax88]. As a consequence,

[22]If we use trees to encode states, a better estimate is one transformation per tree node. The Draco port [Bax87b] had trees with easily $10,000$ nodes, so a branching factor of $10,000$ would be far more realistic.

many times two sequential transformations will "commute"; *this tells us that often the end result does not depend on the exact decision order.* An important consequence is that *two accidentally adjacent transformations can almost always be exchanged without affecting the result.* This is an essential property of many search spaces which we feel has been greatly underrated[23]. Note that we have no guarantee of a multiplicity of paths; just a reasonably high likelihood. Chapter 7 provides a number of different arguments as to why this should be true for the search spaces of transformational implementation. We will also see how to use this property to help revise a derivation history in Chapter 7.

Knowledge of the multiplicity of paths can also help us reduce the apparent size of the search space by eliminating some obviously redundant paths. An arbitrary labeling of the transformations can provide a basis for a *lexical ordering* of paths that are equivalent under commutation of transformations; only the path with the least ordering need actually be considered. A search method based on this insight was implemented and is described in [Bax88].

3.3.4 On the high cost of transformational implementation

If one expects that there are potentially several transformations, representing different implementation technologies and possible locaters, that can be applied to a particular state, then the problem of choosing among the transformations arises. With a fully automated specification-to-code system, the implementation of a specification is very likely to require a considerable amount of search in order to choose a good path through the design space from program to the final code (this occurs in the **LIBRA** system [Kan79, KB81]). Because of the size of the design space, it is probable that any such system will visit nodes only within a fixed distance d of the path defined by the final derivation history followed.[24] The area covered by this band is the length of the band (the actual number of design choices k) times the width of the band (b^d for branching factor b), or kb^d. For $k = 10,000$ (a moderately complex program), $b = 2$ and $d = 5$, with 250 ms. per transformation[25], it would take about 25 hours of CPU to obtain an implementation this way. Even an order-of-magnitude speedup makes the machine cost per implementation fairly large. It is perhaps an unavoidable cost to

[23]Parnas [Par76] noted the commutativity of some decisions, but did not make any observations as to frequency or how that might help one during development or maintenance.

[24]The Draco port was done in such a way that only one transformation was applicable at any point ($d = 0$) in the program (i.e., there were no alternative transformations); no human intervention was required. The consequence was that while we succeeded in porting the system, the resulting code is quite "vanilla" (and consequently inefficient) in its flavor.

[25]The rate of transformation application determined by the Draco porting project, 40000 transformations in about 3 hours on a Sun 3, holding $d = 0$ to avoid the control problem.

explore this region for a first implementation, but if it (mostly) replaced a dedicated designer, we would likely accept it.

Until such choices can be made entirely automatically, we must expect that human designers will be called upon to make at least some of those choices; Balzer [Bal85a] makes a convincing argument that we will never be able to entirely automate the process. If even a small percentage of the transformations require designer intervention, a valuable resource (the designer's time) will be consumed during the implementation process. Each such intervention requires that a designer be shown the problem, determine an explanation of its cause, possibly by examination of its context, determine a resolution, and enter the solution (as the application of a property-preserving transform) into the transformation system, at the cost of some fraction of an hour for each. The **GLITTER** system [Fic82, Fic85] was able to automatically produce 45 out of 50 required transformations; if this is average and scales linearly, we could expect a designer to be involved in about 10% of the transformations. Assuming, with good tool support, 10 minutes of designer time per necessary interaction, a typical string of $10,000$ transformations will require 1000 designer interactions, or about 166 hours of concentrated designer time for *each* fresh reimplementation. Of course, this time is not all likely to come in a block, so the designer may have to be idle waiting for the next interaction to occur.

We see that considerable energy is required to acquire an implementation, both on the part of the machine, and on the part of a participating designer. If we could build an artifact just once, such costs might in fact be considered extremely reasonable. The pervasive need for maintenance suggests that we will actually need to repeatedly modify the artifact. The "modify specification and reimplement transformationally" model of Balzer [Bal85a] would make us pay this price for *each* implementation. Even if the machine time were acceptable, the designer would soon grow tired of the interactions needed for each implementation. If we believe that much of the implementation does not change, the designer will especially object to handling the *same* problems each time.

A solution is to reuse the derivation history (including the transformations chosen by the designer) from a previous implementation to guide the next implementation. Reusing the derivation history requires we be able to validate its individual steps, for which we need a design history, or justification, which we shall address in Chapter 5.

3.4 Summary

This chapter has provided a formalization of the transformational construction process. It has defined terminology necessary for the rest of the thesis, and provided some details about on the nature, size, and costs of the search space, which will be useful in engineering future transformation systems.

It contributions are:

- A general model of the transformation process, emphasizing the role of performance predicates, and explicitly defining the transformation system inputs and outputs
- a classification of specifications in terms of how they interpreted to produce goal predicates
- The notion of an explicit *locater* as a pointing device, or more generally, as a constraint on transform application.
- The careful distinction of *transform* and *transformation*, based on the idea of locaters.
- A survey of existing transformational implementation costs, showing the current high costs, and an extrapolation of those costs
- Recognition of the existence of multiple paths in the design space between two identical states, which will later be necessary for revising an implementation

It has also identified some general weaknesses on the part of extant transformation systems:

- The general absence of explicit performance specifications G_{rest}
- The consequential non-obviousness of motivation for the transformation system to apply any transforms whatsoever
- The absence of explicit specification of $G_{invariant}$ limits choice of which transforms are legitimate to apply

We do not pursue these further in this thesis. We remark that these weaknesses also appear to be present in conventional software engineering methodologies, and solutions might consequently improve conventional practice.

Chapter 4
A Transformation Control Language

Chapter summary. A performance-goal oriented, semi-procedural language for controlling the transformational implementation process is described. Satisfaction of intermediate goals provides purpose, and therefore justification, for the transformations generated, necessary during maintenance.

Determining a path through the design space to an implementation is a difficult problem. Considering the number of steps involved in a typical transformational implementation, it is clear that a manual scheme for choosing and applying transformations will be ineffective in all the but the simplest of cases. Consequently, (semi-) automated means for navigating the design space are needed.

Hardwired control regimes are one possible answer, but suffer from being inflexible, and requiring that the control knowledge be acquired before the transformation system is ever used.

Explicitly specified control knowledge alleviates the problems of hardwired regimes, but to date have been purely procedural. The control knowledge coded procedurally has been termed a *metaprogram* [Fea79, Fea86]. Procedural languages have long suffered the problem of requiring that all the interactions of knowledge used in an application be thought out in advance of use; non-procedural languages offer the possibility of simple encoding of knowledge and use of that knowledge in problem-specific fashions without explicit planning for that case.

We additionally desire a control regime which can help us understand how the applied transformations effect the final implementation, and which is non-procedural to simplify coding. We think such a combination has considerable synergism. The understanding of the relation of goals to applied transformations is crucial to effective determination of transformations reusable during maintenance.

In this chapter, we describe the essence of a semi-procedural metaprogramming language, TCL, for controlling the transformational implementation process.

4.1 Requirements for control knowledge

We believe that navigation techniques should:

- provide focus-of-attention similar to rule and data filtering
- be explicit as opposed to implicit
- be decoupled from the transformations they use
- be somewhat non-procedural
- be easily added to existing libraries of techniques
- be easily referenced or indexed for use in explanations

The key problem in controlling the transformation process is deciding what transformation to apply next, and where to apply it. Similar problems occur with pure production systems, in which two major themes for controlling production application are [BFKM85]:

- rule filtering: choosing a subset of rules to consider for each firing cycle
- data filtering: limiting rule firing to a subset of database facts

Without such filtering, the branching factor at each point of the design space is very large, and the transformation system will be bogged down simply trying to enumerate the choices.

Most early transformation systems have either very little control knowledge, or have this knowledge "wired-in". The CIP system [BBG+78, BEH+87, BMPP89], has literally no control knowledge; interactive application of individual transformations was expected. Transformation systems such as [BM84, BU86, SKW85, McC88] have hardwired navigation in the sense that the techniques are entirely procedural, and are not open to revision once the transformation system has been constructed. For example the TAMPR system [BM84], knows procedurally to apply transformations to push the program through seven stages to achieve an implementation; the REFINE transformational compiler is similar, using 5 major phases [Gol89]. Furthermore, the control regimes are opaque in the sense that information about *why* the various applied transformations were chosen is unavailable. Nonexistent or implicit control knowledge cannot be used to justify why transformations were applied once implementation is complete. We desire the control knowledge to be explicit so it can at least be referenced for explanatory purposes.

A *single* property-preserving transform may have several different effects as seen by various performance measures, each effect being useful in one or more plans. Consider the distributive law:

$$?x * (?y + ?z) \Rightarrow ?x * ?y + ?x * ?z$$

Its application has negative effects on $p_{Halstead}$ as the program volume grows without any increase in function. Yet embedded in a simplification routine that is trying to eliminate multiplications by possible application of multiplicative inverses, the law is quite useful. Consequently a particular transform may play different roles in different navigation methods. We should keep the transforms separate from the plans in which they are used: 1) to economize on reasoning about the possible useful effects, 2) to allow us to reason about effects of interactions of multiple transforms independent of the the plans in which they participate (this will be necessary for transformational maintenance), and 3) to allow us to recognize effects common to several transforms. Recognition of shared effect is key to the generalization leading to code optimization by finite differencing as characterized by [Smi89], in which distributive laws (a generalization of transformations with a certain kind of structure) play a key role.

Non-procedural navigation knowledge is desired in the sense that we want the transformation system to decide for itself how to the apply the knowledge. Yet we cannot go too far in the direction of purely declarative knowledge, for really efficient application of this knowledge is only possible when knowledge has been compiled in the form of procedures. Since general compilation of control knowledge from desired effects is unlikely to be efficient in the near future, useful navigation knowledge must have an element of procedurality to it. Our hope is to minimize the amount of reasoning required by the transformation system during the implementation process. We consequently expect that control knowledge will always be a mixture of declarative and procedural knowledge.

Addition of control knowledge to existing libraries is necessary for the long-term growth of utility of a transformation system, and necessary for short-term application when needed control knowledge is absent.

Capture of application of control knowledge must be relatively easy, so that we are encouraged to do so for the purpose of explaining *why* each transformation was applied. "Why" ultimately boils down to the synergism of the indirect effects on performance predicates of all the transforms triggered by the containing method. We will need this information to allow a design maintenance system to reason about which transformations in a derivation history are still useful.

4.2 TCL: A Transformation Control Language

We designed a metaprogramming language, TCL [Bax87a], to meet these requirements. The most important insight behind TCL is how much of the planning literature seems to have gone unnoticed (or at least, unmentioned) in the transformation system literature; in particular, the notion of explicit hierarchical plans (procedure plus goals). Explicit control mechanisms with hierarchical procedures have appeared for several transformation systems [Wil83, GMM+78, GMW79, Pau87, Gol89, KB88, KB89b]. However, these all miss a major point:

A procedure needs to be associated explicitly with its intended purpose.

Without such information, a procedure cannot be looked up by its purpose, cannot be validated after application in either its original context or a new context, and cannot be used in any serious explanation. Considering that performance predicates are needed to state such goals, and that few transformation systems make *any* performance predicates explicit, the absence of plans with described purpose is hardly a surprise.

In this section, we discuss an abstract characterization of a navigation process based on the primitive mechanisms we identified for TCL.

The main ideas in TCL are:

- Methods– heuristic procedures for controlling the transformation process
- Mixture of procedural and non-procedural invocation of methods
- Implicit and explicit alternatives
- Indexing of methods by performance predicates in postconditions
- Binding locales (this and locaters are new additions to [Bax87a])

We begin with discussion of locales as a necessary prerequisite to understanding actions involving locales, and thus methods.

4.2.1 Locales and locale expressions

A locale is a region of a program constraining the bindings of a potentially-applicable transform . Henceforth we will limit the use of the term locator for locales that are expected to generate just a single binding. Locales are consequently a form of data filtering.

DEFINITION 4.1: *Locale.* A (loose) constraint on bindings:

$$\ell : \mathcal{S} \times \mathcal{T} \rightarrow powerset(\mathcal{B})$$

□

Given a state and a transform, a locale can be used to determine if a locater value is legitimate. It is convenient to name locale values with *locale variables*, designated lv, to allow a locale to be multiply referenced.

Locale operations compute new locales. Such operations can acquire the value of a locale variable, limit a locale value to a narrower region, or expand the scope of a region somewhat. A locale expression e is simply one or more composed locale operations computing a single locale. In a sense, a locale expression is a procedural construct, in that it states *how* to limit the area in which a transform should be applied. A nonprocedural characterization of locales might delay the definition of a region until more information had been collected or it was actually needed.

The set of appropriate locale operations is currently a little unclear. One needs ways of delimiting the scope of application of possible transforms to particular regions of programs. Obvious candidates for locale operations are:

- *locale generators*, such as "*entirestate*", locater constants (such as tree paths $\langle 2, 3, 1 \rangle$) and actual locater values of applied transforms. PADDLE [Wil83] provided implicit locale generation via pattern matching, used to define where later tree edits would apply; we insist on locales as first class objects, so that they may be manipulated.
- *locale references*: a locale variable name stands for the locale value it names.
- those provided by the structure of the underlying representation for the programs, such as:
  - *locale-narrowing*, similar to the subtree selection provided by the LOCALE commands of the Draco system [Nei84b]
  - *locale-moving*, such as tree-navigation techniques (strings of operations "*up*" and "*kthson*" operations) used by SPECIALIST [Kib78]
  - *locale-expanding* operations, such as "anywhere in the subtree" for a locale selecting a node in a tree-based representation scheme, "expand to include neighbors of current locale" in a graph-based representation scheme, or "locaters of any property-preserving transforms which intersect the current locale." The ARIES specification assistant [JF90] allows walking a parse tree via semantic links.
- *locale performance partitions* of the programs implicitly generated by scope-of-effect on performance value functions, such as basic blocks (for p_{sloc}) or name scope (for $p_{meaning}$). A very interesting partitioning called *program slices* [HPR87, HR88] is used to divide a program into semantically well-defined but independent pieces (one should actually be able to optimize separate slices independently, although no work of this nature has been tried.)
- *locale combining operations*, such as unions, intersections, and complements; these operations are well defined because locales are (binding) set functions. Intuitively, these combine regions of the state.

The utility of these constraints will be determined by their effectiveness in encoding useful methods. We simply assume that some locale operations are available.

4.2.2 Methods and Actions

Individual property-preserving transforms applied by a transformation system by definition preserve $G_{invariant}$. However, the transformation system has to somehow achieve G_{rest} by applying strings of such transformations. How can it determine such a string?

Following the lead of conjunctive planners whose goals are sets of propositions [Sac77, CM85, Wil88, Kam89] we split the target specification G_{rest} into smaller parts, each of which is achievable by (logically) separate (but possibly overlapping) plans.

In a transformational context, we use the term "*method*" to refer to plans coupled with the descriptive information describing their effect. The name "method" reminds us of their heuristic nature; we do not expect to always be able to code control *algorithms*, either because our techniques only work on special cases, or because it is sometimes cheaper to simply try the method than it would be to evaluate an accurate test defining when the method worked. The term "method" and its definition are inspired by work on *meta-level inference* [Sil86], which uses heuristic methods to solve college-entrance-examination algebra equations. Each TCL method consists of a postcondition indicating what effects it is expected to achieve, and a body that gives a procedure for how to accomplish the result. TCL methods can have preconditions; but as they are not important to this thesis, we don't make any effort to give them special status.

DEFINITION 4.2: *Method.* A triple $m = \langle i, a, G \rangle$ consisting of an identifier i, a procedure a and a performance predicate G, called the method postcondition. The intent is that if it is legitimate to apply procedure a to a design state s_j, then $G(a(s_j))$ is true with reasonable probability[1]. Thus, a method m is a heuristic procedure a for achieving the effect G. □

We extend performance predicates in method postconditions to allow variables standing for arbitrary predicates or values; this is used match method postconditions with arbitrary predicates, in a fashion similar to Prolog. The identifier i of a method is used to distinguish methods collected in a set of methods M.

[1]A useful piece of information that might be associated with each method is an empirically estimated probability of success; then one could rank methods, providing a natural trial order.

Each procedure describes a collection of steps, as well as sequencing of those steps. Steps may be transform applications, direct calls to sub-methods, or non-procedural invocations of methods, as well as ordering of steps[2].

DEFINITION 4.3: *Procedure.* A collection of actions with sequencing constraints. The set of procedures is denoted $\mathcal{A}$ (think of these as "actions"), with individual procedures $a_i \in \mathcal{A}$. Procedures are recursively defined from the following actions:

- $APPLY(c_i, e)$, applies transformation c_i^ℓ somewhere in locale selected by expression e.
- $SEQ(a_1, a_2)$, defining sequential composition of actions.
- $AND(a_1, a_2)$, defining parallel composition of actions.
- $OR(a_1, a_2)$, defining a choice between actions.
- $ELSE(a_1, a_2)$, defining a secondary choice between actions.
- $CALL(name, \sigma)$ where $name$ is the name of a method, and σ is an argument list consisting of performance predicates or locale expressions. We use the notation $name\sigma$ to mean the action $name$ with σ substituted appropriately.
- $REQUIRE(G, e)$ where G is a performance predicate, and e is a locale expression.
- $ACHIEVE(G, e)$ where G is a performance predicate, and e is a locale expression.
- $ACHIEVEBY(G, e, a)$ where G is a performance predicate, e is a locale expression, and a is an action.
- $PLAN(A, >_{plan})$ where A is a set of actions, and $>_{plan} \subseteq A \times A$ is a partial ordering (specific to the plan) over those actions, constraining order of execution. Sometimes an empty partial order (no constraints) is useful; applying a bag of transforms in non-overlapping locales is such a circumstance. An empty partial order is represented by the symbol $\emptyset$. A $PLAN$ captures the notion of a *nonlinear plan* [Cha87, CM85, Kam89].
- $RETURN(e_1, e_2, e_3, \ldots)$, computing multiple locale values to be passed to a parent action. This is useful only as the last action of a plan.
- $LET(lv_1, lv_2, \ldots, a)$, capturing multiple locale values returned from action a. The scope of the lv_is are plan elements necessarily following the LET.
- $LOCALE(lv, e, a)$ where lv is a variable name whose scope is the action a, and e is a locale expression, possibly containing references to locale variables. This construct allows locales to be computed and passed by name to sub-actions.

□

[2]TCL also has commands for displaying state information and querying the software engineer about choices. These have little actual effect on the nature of control for TCL, and we consequently leave them out for brevity.

More complex actions could be defined but they would mostly be compositions of these primitive actions, so we do not discuss them further here.

Note that *LOCALE*s and *LET*s form a single-assignment language with static scoping similar to the notion of *let* in functional languages.

We remark that methods only attempt to apply a limited subset of the transforms available to the transformation system: those directly invoked via *APPLY*, and those indirectly invoked via *ACHIEVE*. Thus methods are a form of *rule filtering.*

Since methods are a kind of program, it could be useful develop them transformationally, as proposed by [KB89a].

4.2.3 A goal directed transformation process

A transformation system is supplied with a set of methods $M_{library}$ as well as a (partial) specification when it starts (Figure 4.1). Initially, the system constructs the procedure $ACHIEVE(G_{rest}, entirestate)$ and "executes" that procedure.

Procedure execution consists of performing actions as specified by the procedure steps, starting with the outermost. Flow of control is explicitly given by the various actions. An action may fail; if so, backtracking occurs to a choice point and an alternative action is tried. Given that the current state is s_j, each of the consequences of possible actions are given in the following paragraphs:

$APPLY(c_i, e)$ changes the design state to $s_{j+1} = c_i^\ell(s_j)$ for some ℓ such that $e(s_j, \ell) = true$. The value ℓ is logically *RETURN*ed to a parent action; see discussion of *RETURN* below.

$SEQ(a_1, a_2)$ executes $a_2(a_1(s_j))$. If either action fails, then *SEQ* fails.

$AND(a_1, a_2)$ executes any interleaving of a_1 or a_2 under the assumption that the serialization is equivalent to both $a_2(a_1(s_j))$ and $a_1(a_2(s_j))$. If either action fails, then *AND* fails.

$OR(a_1, a_2)$ establishes a choice point, and then executes, nondeterministically, either a_1 or a_2. If the chosen action succeeds, then *OR* succeeds. If the chosen action fails, then the other action is tried. If both a_1 and a_2 fail, then *OR* fails. A parent action of *OR* does not hear about *OR* failing unless both alternatives have been tried.

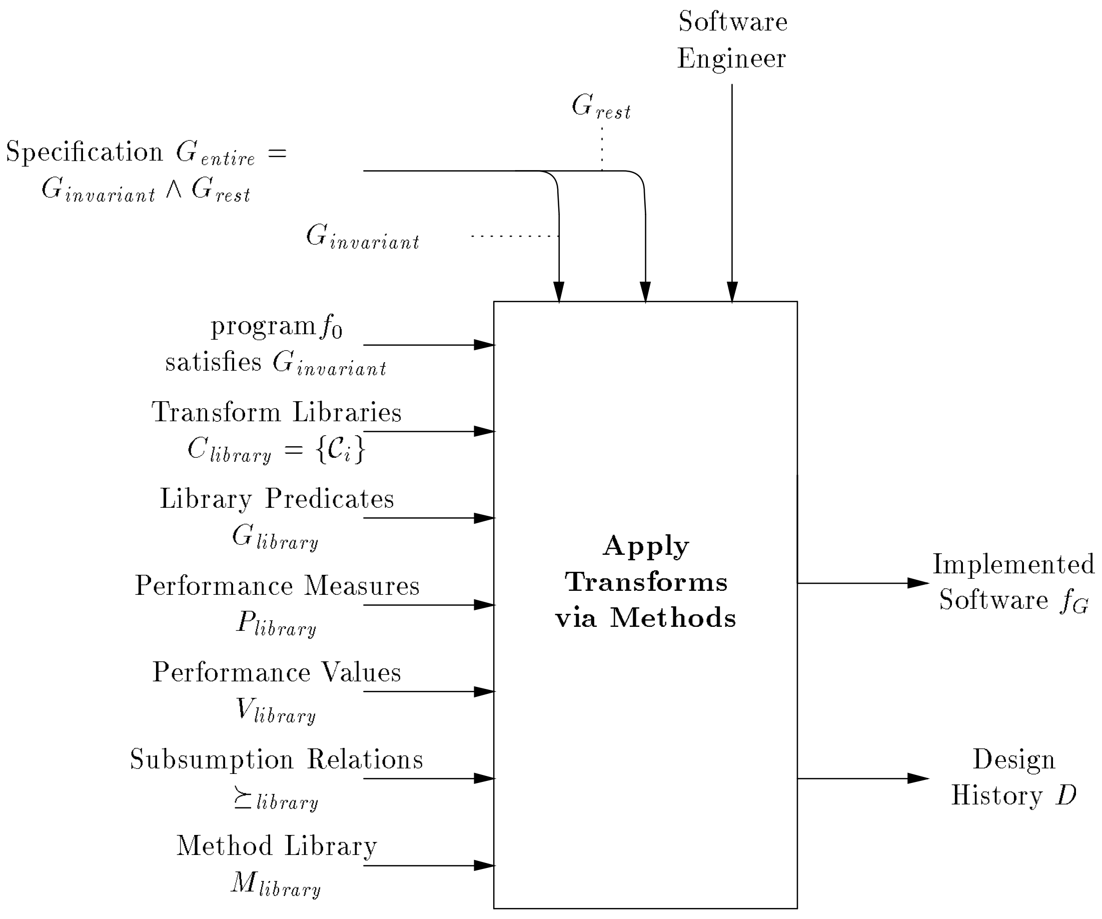

Software Engineer
G_{rest}
Specification $G_{entire} = G_{invariant} \wedge G_{rest}$
$G_{invariant}$
program f_0 satisfies $G_{invariant}$
Transform Libraries $C_{library} = \{\mathcal{C}_i\}$
Library Predicates $G_{library}$
Performance Measures $P_{library}$
Performance Values $V_{library}$
Subsumption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

Figure 4.1: Transformation system controlled by Methods

$ELSE(a_1, a_2)$ establishes a choice point for a_2, and executes a_1. If a_1 succeeds, then $ELSE$ succeeds. If a_1 fails, then a_2 is tried. If both a_1 and a_2 fail, then $ELSE$ fails. The difference between OR and $ELSE$ is that for $ELSE$, a_2 can be designed with the additional knowledge that a_1 failed.

$REQUIRE(G, e)$ succeeds if $G(F(e, s_j))$ is true, otherwise fails; this basically just a test. $F(e, s_j)$ refers to a region of the program f_j defined by the possible locaters determined by e; this region must be a well-formed program sub-scheme. $REQUIRE$ can be used to establish a *filter* [Kam89] precondition for a method by structuring the method as $SEQ(REQUIRE(G_{pre}), a_{body})$. A filter precondition is one that is necessary for success of the method, but is inappropriate to try to $ACHIEVE$; usually filter conditions test unchangeable attributes of entities, e.g., the type of a variable. $REQUIRE$ is used to check that part of a method postcondition which is not provably true after method body execution by structuring the method as $SEQ(a_{body}, REQUIRE(G_{probablytrue}))$.

$CALL(name, \sigma)$ logically executes the procedure $SEQ(a_{name}\sigma, REQUIRE(G_{name}\sigma))$ defined by the components of the named method $\langle name, a_{name}, G_{name}\rangle \in M$; in practice, it simply executes $a_{name}\sigma$. The substitution σ provides values for variables defined in the named method. $CALL$ fails if the body of the method fails, or the postcondition G is false on completion of method execution. Postcondition checking is optimized by structuring methods as outlined under the description of $REQUIRE$. This avoids the need to actually evaluate expensive performance predicates (and their underlying performance measures) which are provably true by means outside the transformation system.

$ACHIEVE(G, e)$ attempts to solve the performance goal $G(F(e, s_j))$ by partitioning it into subgoals of which at least one appears readily solvable by an existing method, and then attempts to solve the rest of the subgoals. In detail, $ACHIEVE$ succeeds immediately if $G(F(e, s_j)) = true$. Otherwise it performs a version of means-ends problem solving by establishing nondeterministic choices for each method $m_k = \langle i, a_k, G_k \rangle$ in the method library such that there is a substitution $\sigma_{k,x}$, and performance predicate G_x with[3] $G_k\sigma_{k,x}, G_x \vdash G$. Each choice is tried in turn by executing the procedure $AND(CALL(m_k, \langle e, \sigma_{k,x} \rangle), ACHIEVE(G_x, e))$. Success of any choice causes success of $ACHIEVE$. Failure of all choices causes failure of $ACHIEVE$. The identifier i in retrieved methods is ignored. Note that $ACHIEVE$ can accomplish, non-procedurally, parallel actions that end in a desired result because it breaks a desired goal into multiple subgoals. It can also accomplish non-procedural sequential composition of actions to effect a goal if methods have a structure $SEQ(ACHIEVE(G_{pre}, e), a_{body})$; then backwards chaining for goal decomposition takes place when the method is invoked.

$ACHIEVEBY(G, e, a)$ is identical to $ACHIEVE(G, e)$ with the proviso that action a is attempted first. This provides a way of designating a favored action.

$RETURN(e_1, e_2, e_3, \ldots)$ computes multiple locale values to be passed to some ancestral LET. This can be used to pass locations of interest from a plan or method back to a higher-level plan.

$LET(lv_1, lv_2, \ldots, a)$ captures multiple locale values $RETURN$ed by action a and assigns to the variables lv_1, lv_2, in the order $RETURN$ed.

$LOCALE(lv, e, a)$ assigns the value of locale expression e to the new variable lv, and executes action a with variable lv visible for use in locale expressions in a. Any previously visible lv may be used in e, but is not visible in a.

$PLAN(A, >_{plan})$ executes the actions $a_i \in A$ in any order consistent with the partial order $>_{plan}$. Failure of any action a_i causes failure of $PLAN$. Note that each a_i can be an OR or $ACHIEVE$ action, so there may be multiple ways for a $PLAN$ to succeed.

[3]This requires second order matching facilities [HL78] and a theorem prover. We don't dictate how strong; a very simple one might simply compare subterms of conjunctive normal forms for unifiability. The theorem prover may even reside externally in the form of a software engineer who supplies the right values. We do expect that implications based on dominated performance bounds (using various $\succeq$ relations) would be included in such a theorem prover.

Nonlinear plans are generalizations of both

$$SEQ(a_1, a_2) \equiv PLAN(\{a_1, a_2\}, a_1 > a_2)$$

and

$$AND(a_1, a_2) \equiv PLAN(\{a_1, a_2)\}, \emptyset)$$

Consequently, we shall allow use of *SEQ* or *AND* in examples, but shall deal only with *PLAN* in theoretical developments.

4.2.4 The role of the Software Engineer

The software engineer, during operation of the transformation system, is limited to at most selecting from possible choices available to the transformation system when executing the following actions:

- *APPLY*: the engineer chooses the exact locater
- *ACHIEVE*: the engineer chooses which method to use, the partition goal G_x and/or the substitution σ
- *OR*: the engineer chooses which alternative

Whether or not, and how this interaction takes place is beyond the scope of this document.

Both Balzer and Cheatham suggest the need for an interactive transformational implementation process because of the virtual certainty that the transformation system is incapable of performing all possible optimizations [Bal85a, CHT81]; some optimizations will invariably come from an outside agent. The software engineer (wearing his domain engineering hat) may have insights for useful new transforms and/or methods while guiding an implementation , but is not allowed to introduce them directly during the implementation process according to our model. Such insights must be introduced instead as *support technology deltas* (Chapter 6). This requirement does not prevent an interactive process of switching between implementation and applying deltas to revise information known to the transformation system.

4.2.5 TCL Examples

In this section, we provide several examples of TCL to give the reader an idea of its scope.

Control via Script

TCL can express *any* specific transformation sequence, or script, of transformations $c_1^{\ell_1}, c_2^{\ell_2}, \ldots, c_k^{\ell_k}$ as follows:

$$m_{specific} = \langle specific, a_{specific}, ?g \rangle$$

with action

$$\begin{aligned} a_{specific}(lv) = & SEQ(APPLY(c_1, \ell_1), \\ & \quad SEQ(APPLY(c_2, \ell_2), \\ & \qquad SEQ(APPLY(c_3, \ell_3), \ldots \\ & \qquad\quad \ldots APPLY(c_k, \ell_k)))) \end{aligned}$$

The "$?g$" postcondition (meaning 'unknown') of the method will match a top level $ACHIEVE(true, entirestate)$, and $a_{specific}$ will then run.

The constructibility of any transformation sequence allows TCL to be used to build a method for constructing any specific implementable artifact, although that method may be useful only for that single artifact.

Barstow [Bar88, Bar89] apparently expects to use scripts of some type to apply transformations in Φ-NIX. He does not give enough detail to determine what locaters, if he has them, look like.

Blind Search Control

This example uses TCL to describe a transformation system with completely blind search, i.e., is willing to try any property-preserving transform anywhere, anytime. All one need to do is to package all the property-preserving transforms c_i in a single method

$$m_{blind} = \langle blind, a_{blind}, ?g \rangle$$

with action

$$\begin{aligned} a_{blind}(lv) = & OR(REQUIRE(?g, lv), \\ & \quad SEQ(OR(APPLY(c_1, lv), \\ & \qquad OR(APPLY(c_2, lv), \\ & \qquad\quad OR(APPLY(c_3, lv), \ldots \\ & \qquad\qquad \ldots APPLY(c_{|C|}, lv)))) \\ & \qquad ACHIEVE(?g, lv))) \end{aligned}$$

The *REQUIRE* action in this example does not act as a precondition of m_{blind} because it is not part of a $SEQ(REQUIRE(\ldots),\ldots)$ idiom. Instead, the *REQUIRE* acts as a loop termination test of the tail-recursion caused by the trailing *ACHIEVE*.

Starting TCL with $ACHIEVE(G_{rest}, entirestate)$ will run m_{blind} because its post-condition, $?g$, has the property that $?g, true \vdash G_{rest}$ with $\sigma \equiv\ ?g \rightarrow G_{rest}$, causing TCL to propose the procedure

$$AND(CALL(blind, entirestate), ACHIEVE(true)))$$

M_{blind} will then nondeterministically try every possible sequence of transforms, and test the resulting string for the desired G_{rest} via the substitution of σ on $?g$. A failed test will cause a backtrack, and an alternative will be generated.

Using the same form as m_{blind}, and using just transforms which monotonically decrease some performance value (e.g., transforms which decrease execution time such as $?x + 0 \Longrightarrow ?x$), one can form "simplification" methods similar to those of PADDLE [Wil83], but constrained to operate over a specified locale, which PADDLE cannot express.

Control for MEDUSA

In Section 3.2.4 we briefly described the MEDUSA system. We emulate some[4] of the implicit control of MEDUSA which achieves complexity limits, using explicit TCL methods. Rather than using an example from MEDUSA's domain of computational geometry, we use a more familiar example of sorting.

[4]MEDUSA also has built-in constraint propagation to further minimize poor choices. TCL does not provide constraint propagation, although it would appear to be a promising addition.

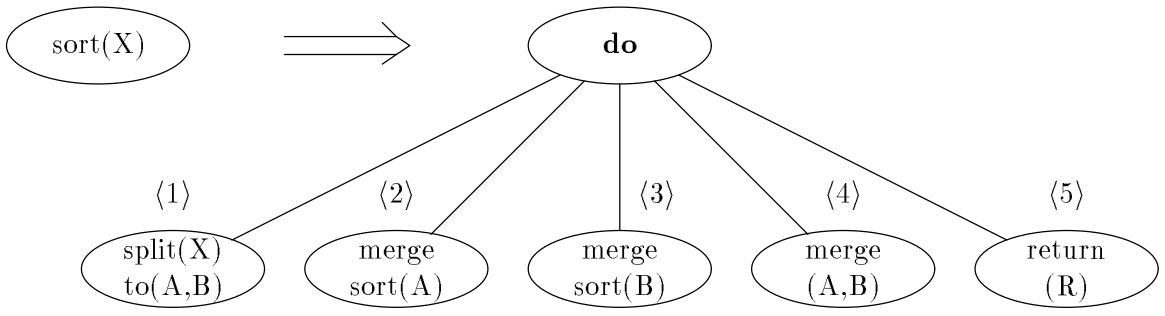


Figure 4.2: $c_{meaning,sort,mergesort}$ transform and "*kthson*"s

Remembering that MEDUSA transforms substitute code-skeletons for placeholders, we assume the existence of a set of $c_{meaning,j,k} : j \rightarrow codeskeleton_k$ for every placeholder j. For each such transform, we define a method

$$m_{j,k}(lv) : \left\langle \begin{array}{l} j{-}to{-}k, \\ APPLY(c_{meaning,j,k}, lv), \\ p_{complexity}(codeskeleton_k) \succeq_{complexity} p_{complexity}(j) \end{array} \right\rangle$$

For instance, assume the placeholder "*sort*", with two transformations:

$$c_{meaning,sort,bubblesort} : sort(X) \Longrightarrow \ldots codeforbubblesort \ldots$$

and

$$\begin{array}{ll} c_{meaning,sort,mergesort}{:} & sort(X) \Longrightarrow \\ & split(X)to(A,B); \\ & mergesort(A); \\ & mergesort(B); \\ & merge(A,B)to(R); \\ & return(R) \end{array}$$

A possible tree transformation to mergesort is shown in Figure 4.2 where each $\langle k \rangle$ indicates a *kthson* of the right hand side, used in $a_{mergesort}$ below.

These two transforms induce the following methods, decorated with complexity costs:

$$m_{sort,bubblesort}(lv) = \left\langle \begin{array}{l} sort - to - bubblesort, \\ \ldots APPLY(c_{meaning,sort,bubblesort}, lv), \ldots \\ p_{complexity}(e) = \mathcal{O}(n^2) \end{array} \right\rangle$$

and

$$m_{sort,mergesort}(lv) = \left\langle \begin{array}{l} sort - to - mergesort, \\ a_{mergesort}, \\ p_{complexity}(e) = \mathcal{O}(n \log n) \end{array} \right\rangle$$

with

```
$a_{mergesort}(lv) =$
  $SEQ(LET(lv_0, APPLY(c_{meaning,sort,mergesort}, lv)),$
       $PLAN(LOCALE(lv_1, 1thson(lv_0), ACHIEVE(p_{complexity} \succeq \mathcal{O}(n), lv_1))$
              $LOCALE(lv_2, 2thson(lv_0), ACHIEVE(p_{complexity} \succeq \mathcal{O}(n \log n), lv_2))$
              $LOCALE(lv_3, 3thson(lv_0), ACHIEVE(p_{complexity} \succeq \mathcal{O}(n \log n), lv_3))$
              $LOCALE(lv_4, 4thson(lv_0), ACHIEVE(p_{complexity} \succeq \mathcal{O}(n), lv_4))$
              $LOCALE(lv_5, 5thson(lv_0), ACHIEVE(p_{complexity} \succeq \mathcal{O}(1), lv_5))$
              $\emptyset)$ % no ordering on plan steps
       )
```

The $a_{mergesort}$ plan steps restrict the attention of the transformation system to the various placeholders in the code skeleton for mergesort, and ensure that each such step is implemented with the proper complexity to support the postcondition of the method[5]. For instance $4thson(lv_0)$ refers to the $merge(A, B)$ step, which is constrained to performance level $\mathcal{O}(n)$ to ensure the overall $\mathcal{O}(n \log n)$ performance of the entire *mergesort*. Notice that no checking for a correct postcondition is made, because it is a provable consequence of the plan.

Now a MEDUSA mixed specification $\langle sort, \mathcal{O}(n \log n)\rangle$ will set $f_0 = sort$, cause the the implied TCL command $ACHIEVE(\mathcal{O}(n \log n), entirestate)$ to reject $m_{sort,bubblesort}$ and try $m_{sort,mergesort}$, leading to a correct implementation.

[5]This method is one for which the method postcondition need not be evaluated, as it is provably true.

4.3 Related Control Mechanisms

TCL was influenced by a number of other control mechanisms. In this section we review the ideas behind some other control mechanisms and discuss, where relevant, how TCL implements or improves on some of those ideas. The mechanisms that we considered fell primarily into the following categories:

- Procedural control
- Production systems
- Metaprogramming
- Planners

We consider each of these in turn. Readers not interested in comparisons may skip to the summary without loss of continuity.

4.3.1 Procedural control

Purely procedural control consists of control programs using the full range of constructs (such as loops, conditional, procedure calls, etc.) from an arbitrary procedural programming language (such as LISP) not designed specifically to handle to the control problem.

Transforms may be implemented directly in the language, or may have be invoked as entities defined in a different notation. The REFINE system [SKW85, Rea86] is a good example of a transformation system with procedural control. It allows control procedures to be coded in the REFINE wide-spectrum language containing procedural primitives. REFINE transforms are coded as tree rewrites using REFINE's transform operator, applicable to a predefined tree data type, or can be implemented by direct surgery on such trees.

We have already argued against purely procedural control for metaprogramming. The fundamental reasons are that such control is hard to code and hard to reason about. Commercial vendors can perhaps afford to amortize the costs of coding a fully procedural control such as that of the REFINE language transformational compiler [SKW85] by amortizing those costs over the large customer base. However, we think that the need to mechanically reason about plans, individual transformations, their interactions, and how they justify the final artifact, especially for maintenance, will eventually force nonprocedural elements to appear in every metaprogramming language. Such a need to reason about the plans requires that those plans also be explicit, as opposed to implicit as they are in REFINE compiler.

We want to reiterate the value of a semi-procedural language, providing both the efficiency benefits of procedural execution by avoiding repeated reasoning about how to accomplish some effect, coupled with declarative descriptions of effect. This is the fundamental reasons for the notion of *METHOD* in TCL: a linkage between procedural acts and the explicit effect they are supposed to achieve. The limited set of control primitives in TCL (*ELSE*, *CALL*, *PLAN*) are an attempt to provide the needed element of procedurality without making the set too rich for maintenance analysis; further discussion of this will have to wait for Chapter 8.

4.3.2 Production Systems

Production systems consist of a fact database, and a set of inference rules used to augment the database by adding new facts or deleting existing facts [BFKM85]. A production system cycle consists of determining which rules match (perhaps, for each rule, in many ways) to which facts, performing *conflict resolution* by choosing just one rule/binding pair, and updating the database according to the consequences of the chosen rule with the chosen binding. The branching factor induced in such a space is roughly proportional to the number of facts in the database times the number of rules; with a large data- and rule- base, the production system can spend most of its time performing conflict resolution rather than actually applying inference rules. Consequently emphasis on control for production systems is on rule- and data-filtering which limit the focus of attention to a subset of the available rules and data.

A transformation system has the identical problem when attempting to determine which transformation to apply. TCL provides data filtering via the notion of locale, which restricts the attention of transformation system to semantically-defined regions of the state. Locale expressions compress or expand locales according to the semantic structure of the program being manipulated, and can be passed between methods as a means of ensuring continued focus. While locales are present in various forms in some other transformation systems, they are not first class entities, cannot be manipulated, and cannot be passed explicitly. We cheerfully admit the need for considerable further work on locales to identify the useful locale “scaling” operations.

TCL provides rule filtering via method bodies, which either explicitly specify which rules (transforms) are to be applied via *APPLY*, or implicitly specify rules via indirect applications caused by methods invoked by *CALL* or *ACHIEVE*. Hierarchical planners have long had this sort of rule filtering, but the focusing effect is not discussed.

Systems like OPS5 [For82] perform a *metamatch* of each rule to all the other rules. The value of the metamatch is that it identifies, for any particular rule, what other rules might potentially apply once this rule has been used, and which data used or generated by this rule would be relevant. Rule- and data- filtering are then achieved by only trying the rules that metamatched the previously fired rule; this shortens the conflict resolution considerably. The DRACO transformation tool [Nei80] uses metamatching in this manner. TCL does not use metamatching in any fashion. It is not clear how valuable this technique would be considering TCL's already existing focusing mechanisms.

Another approach to lowering the cost of applying the productions is to apply multiple productions in parallel. Schmolze [Sch89] suggests methods for determining sets of productions whose parallel effect is identical to their serial effect by inspecting their interactions. It might be possible to do this with TCL methods since they are really just complex transforms; the explicit postconditions should make this somewhat easier.

While both metamatching and parallel application shorten the time required to apply productions (or transformations), they do little for the problem of choosing the *right* productions, i.e., those that lead to a desired conclusion. In a Church-Rosser system, one need not choose the right productions because any sequence productions will do, but few practical systems have that property. Consequently we have not invested any energy on these techniques. Metamatching of transforms from various methods could be used to automatically identify potentially applicable "next" methods, but it is not clear how valuable this would be.

One can operate a production system in either a forward-inferencing (facts plus rules give new facts) or backward inferencing (from what facts and rules can a desired fact be deduced?). Transformation systems rarely do any backward inferencing (application of transformations in reverse) because to doing so is tantamount to looking for a specification of an artifact already possessed. While this might be valuable for design recovery of existing code, it is not the purpose of transformation systems. Backward inferencing is necessary in TCL when matching method postconditions with *ACHIEVE* actions.

4.3.3 Metaprograms

We now compare TCL to several control schemes for transformation systems, generally called metaprogramming schemes. The most influential systems were Draco (simply because of our early experience with it, and the idea of *domains*), PADDLE [Wil83] and PRESS [Sil86]. The control schemes we cover, and their fundamental ideas, are:

- Draco: Transformation by refinement through domains
- PRESS: Meta-level inference and hueristic methods
- PADDLE: Procedural metaprogramming language
- Goldberg's metaprogramming language: program regions as first-class values
- PROSPECTRA: Functional metaprograms specified algebraically
- SPECIALIST: Dynamic chaining
- Zap: Functional goals with focus by data-filtering
- Glitter: Automatic application of subsidiary transforms

Draco: Domain-oriented Transformation system

The DRACO transformation system [Nei80, Nei84a, Nei89] transforms programs written in abstract problem domains into programs written in target execution languages (domains). It does not neatly qualify as a metaprogramming system, nor as any other simply described system, because it used a variety of built-in mechanisms to control navigation:

- domain refinements
- global assertion/condition constraints on refinements
- transform priorities
- metamatching
- a primitive form of *LOCALE*
- simple user-definable tactics

Part of the original motivation for TCL was to unify and make explicit many of these mechanisms.

Domains are problem-description languages; domain *refinements* are a special kind of transform (similar to the theory morphisms outlined in Section 3.1.7) used to map programs in an abstract problem language (say, natural language query processing) into programs in a more concrete problem language (e.g., parallel LISP). Use of abstract problem domains is encouraged by the Draco paradigm in an effort to make problem specification simpler [6]. A reasonable domain network (graph of domains plus refinements directed from one domain to another) allows conversion of an abstract problem statement to be refined to concrete executable languages such as FORTRAN. Such a refinement sequence is shown in Figure 4.3. In practice, Draco applies simplifying transformations at each domain level before refining down to the next domain.

A domain network provides considerable guidance to the transformation process. If one has simply an enormous library of potentially applicable transforms, and a large specification, the branching factor at each point in the implementation space is large; since a typical derivation history is tens of thousands of steps long, the number of potential paths is overwhelming. One method for limiting search is using the notion of "islands" in the search space [BF81, Pea84]. The islands analogy likens the search space to an enormous ocean to cross; the crossing process is much easier if there are many islands scattered over the ocean, some of which are not too far out one's desired path. Islands along the path act as short term achievable goals. Each domain in the a domain network acts as an island in the search space. Navigation is aided because major steps in the implementation process are implicitly defined by directionality the domain network from abstract domains to concrete domains (the TAMPR system levels of abstractions [BM84] form such a domain network).

While the Draco tool could be told to refine to specific individual domains one at a time, there was no way to tell it to find an implementation by refining the specification "somehow". This can be done with TCL by defining a performance measure p_{domain} which determines the domain type of a program, and providing a representation of the domain network graph as a method library composed of a set of methods of the following form:

$$m_{refineJtoK} = \langle refineJtoK, a_{refineJtoK}, p_{domain}(lv) = K \rangle$$

$$\begin{aligned} a_{refineJtoK}(lv) = & SEQ(ACHIEVE(p_{domain}(lv) = J), \\ & \quad SEQ(CALL\ simplifyindomainJ(lv) \\ & \quad\quad APPLY(xfmdomainJtodomainK, lv) \end{aligned}$$

[6]It is interesting to note that other transformation systems with active research groups are evolving towards the domain notion (for the TI system, they are called *local formalisms* [Wil86]; we see them in TAMPR as the language levels between the phases of the transformation process (applicative LISP, recursive FORTRAN with local variables, recursive FORTRAN with no parameters, nonrecursive FORTRAN, etc.) [BM84, p. 580], and in REFINE in the form of encouragement to build problem-domain specific languages.

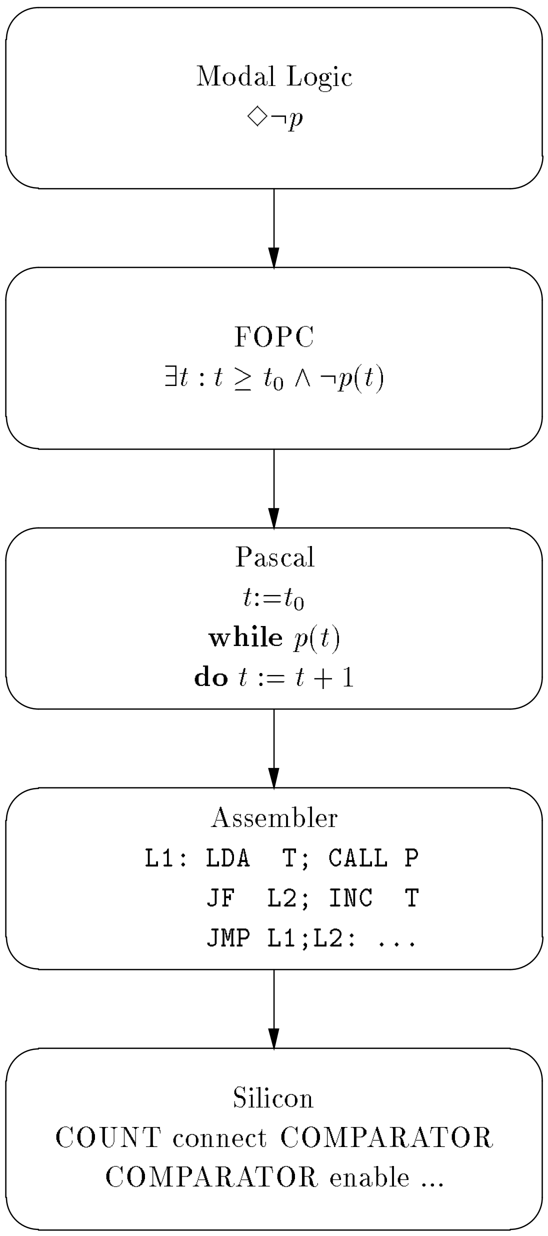


Figure 4.3: Transformation via domain refinements

One method is needed for each transformation *xfmdomainJtodomainK* that refines domain J to domain K. Invoking TCL with $ACHIEVE(p_{domain}(f) = FORTRAN)$ will then find, by a backward chaining process, a sequence of domain simplifications and refinements to produce a *FORTRAN* program. Such a set of methods implements the essential Draco implementation paradigm.

Draco refinements were actually rules that rewrote fragments of a program at one level into program fragments at lower levels. Since a fragment had multiple possible refinements to other domains, a difficulty was ensuring that several separate domain fragments at one level refined consistently with one another; this was accomplished by *assertions* and *conditions*. Assertions were declared when a domain fragment was refined in a certain way; conditions attached to a refinement could check that a consistent assertion was already established. Assertion/condition constraints are not implemented in TCL directly; rather, our belief is that the entire program representation must be refined as an entity in accordance with [ST88], and so the design of TCL assumed an algebraic view in which theory morphisms act as the mechanism for refinement [TM87]. This allows the refinement to be represented as a single transform. We unfortunately did not have time to explore this thoroughly for this thesis, but feel this is a promising avenue.

Within a Draco domain, simplifying transformations are prioritized; the priorities allow groups of transformations to be designated and applied by priority range. Metamatching caused simplifying transformations from the same domain to be matched against one another at transform definition time; this allows the runtime application of one simplifying transformation to efficiently "suggest" (because of a successful metamatch earlier) the application of other simplifying transformations, saving a considerable amount of matching time. The value of transformation priorities as priorities was not demonstrated by the Draco tool; if anything, we found the priorities ended up being used simply as a means of grouping transformations. We have already discussed how TCL can collectively apply an arbitrary group of transformations in our earlier discussion of $m_{simplify}$. Metamatching could be useful for such simplifying methods.

The Draco notion of *LOCALE*, a specified subtree of a tree program scheme, we have generalized as locaters and locales for TCL. Commands for moving up and down in a tree locale correspond to TCL locale scaling operations.

While Draco did provide choices in terms of multiple possible refinements, and multiple possible transforms, it had no ability to backtrack in case of a bad choice. TCL assumes such backtracking ability to back out of poor choices, and a controlled sequencing of alternatives via the *ELSE* action[7].

[7] We will see in Chapters 7 and Chapters 8 a kind of dependency-directed backtracking mechanism expected to complement TCL.

Finally, user-definable Draco "tactics" allow a software engineer to establish a preference for certain types of refinements over others (such as inline substitution of code (MINIMIZE-TIME) vs. creation of subroutines for instantiated code (MINIMIZE-SPACE)); this is effectively a procedural encoding of a goal predicate. The emphasis with TCL is to express the performance goal (which Draco simply cannot do) and let an appropriate method accomplish the effect. TCL carries the notion of user definable tactics quite to the extreme by being virtually a programming language in its own right.

Overall, TCL seems to be expressive enough to describe Draco's control mechanisms.

Neighbors has argued that simple control mechanisms such as Draco's simplify-in-domain, refine-to-next-domain are sufficient for transformational implementation. Glitter's order-of-magnitude reduction in manually-specified transforms [Fic80, Fic82, Fic85], the PADDLE system [Wil83] of complex, problem-specific metaprograms to control transformation sequencing, and LCF's proof plans [GMM$^+$78, Pau87] have shown that complex implementation plans *are* required to carry off complex implementations. Again, these observations lead to TCL as a necessary part of the design, and therefore maintenance, mechanisms.

PRESS: Meta-level Inference

A technique for controlling search in large search spaces, called *meta-level inference* by Silver [Sil86], groups problem-space operations having the same effects on states into "method" (this inspired the name for TCL methods, although a PRESS method corresponds to a set of TCL methods with identical postconditions). Each PRESS method becomes, in effect, an equivalence class of operators. Method postconditions specify the effects; method preconditions state necessary (but not necessarily sufficient) conditions for method to achieve its postcondition. The methods then treated as operators in an abstract space whose actions are defined by the method postconditions; this is similar to hierarchical planning (discussed later). Problem solving consists of blind forward searching by application of methods until the desired effect in the original problem space is achieved. Search savings over the original space occur because each method tried represents an entire class of operations, and the preconditions often eliminate application of a method altogether. Postconditions are checked after method application to verify that the method has truly accomplished the shared effect; Silver argues that it is often cheaper to have a weak precondition, with a dynamic postcondition check that catches those cases when the method runs incorrectly, than it is to encode a necessary and sufficient precondition.

The utility of the idea was shown by PRESS, an expert algebraic equation solver. Silver outlines a number of specific methods for solving algebra expressions; such techniques might be useful in solving in a constraint-propagation subsystem or simplifying conditions on conditional equations. Since TCL is similar to Silver's methods, we think that coding his particular methods would not be difficult, but are of no further interest for this thesis.

Silver's "meta-theory syntactic features", such as "term-occurrence count", used in his method postconditions are formalized as performance measures used in TCL method postconditions. Silver gave no justification for his choice of his syntactic features; we think they are problem-domain dependent and expect the same will be true with TCL. An unsolved domain-engineering problem is determining which performance measures and goals to define; knowing that one will be operating in the domain of algebraic equations does not obviously lead to the notion of "term-occurrence count" as useful.

PRESS actually tries its methods in a particular hand-selected hardwired order; this order presumably had to do with the probability that a particular method had of solving a random problem or leading a step closer. Ranking TCL methods for application according to a dynamically-updated ratios of past successes to failures we think would give the same result with less effort.

TCL directly incorporates the idea of "method" with its dynamically checked pre- and post- conditions. TCL allows a complex plan to form the body of a method, which generalizes PRESS view of method as "bag of equivalent-effect operations". Further, a number of TCL methods may have the same postcondition; this allows incremental addition to a knowledge base of methods. TCL also differs from PRESS in using goal-directed backward inference to select methods to apply; the post-conditions tell TCL when a method might be useful. Apparently the size of problems handled by PRESS was small enough so the notion of locale was not needed.

PADDLE Metaprogramming Language

PADDLE [Wil83] is a procedural language (and supporting system) for defining a *program developments*, i.e., a metaprogram to generate the steps used by a transformation system to implement a program scheme. A PADDLE metaprogram consists of a set of parameterized procedures called "commands"; command names consist of arbitrary English text strings which summarize the action of the procedure. Each procedure may contain program pattern-matching operations, replacement operations (which substitute new program fragments at points designated by previous pattern-matching steps), invocations of "goals" (procedure calls to other named commands), and compound operations for conditional branching and looping operators.

```
**command** divide and conquer(function,set)
  **begin**
    split set = $\{e_1, e_2, \ldots\}$ into subsets $s_1, s_2, \ldots$;
      **by**
        **choose from**
          partitioning into $s_1 = \{e_1\}$ and $s_2 = \{e_2, e_3, \ldots\}$;
          binary partitioning into $s_1 = \{e_1, \ldots, e_{k/2}\}$ and $s_2 = \{e_{k/2+1}, \ldots, e_k\}$;
          basis partition $s_0, s_1, s_2, s_4, s_{2^l}$
            where each $e_n$ is a linear combination of the $s_{2^j}$;
        **end**
    compute a related function $f_1$ on the subsets;
    combine values of $f_1$ on subsets via a new function $f_2$;
    **note** You must ensure that function applied to set = $f_2$ applied to $\{f_1(s_1), f_2(s_2), \ldots\}$;
  **end**
```

Figure 4.4: PADDLE procedure, taken from [Wil83, p. 908]

Transforms are defined by the distributed effect of pattern-matching and replacement operations. PADDLE metaprograms also appear to be augmented by procedural LISP when convenient. An example PADDLE command is shown in Figure 4.4. A PADDLE metaprogram is started by invoking a particular command.

A *goal* is traditionally an evaluable predicate that can be applied to a state, which is true in desirable states. By this definition, the term "goal" as used by PADDLE seems unconventional; it refers instead to subplans. Thus, while one of the stated intentions behind PADDLE is to capture the implementor's *goal* structure, what it truly appears to capture is the implementor's *plan* for implementing a program[8]. Goals are simply not present.

Given a specification, executing a PADDLE program generates an implementation by executing the "goal" structures in sequential order (just as in any conventional procedural language executing statements); no variability in order is allowed. Failure to successfully execute a "goal" (including failed execution of a pattern-matching primitive for which no explicit alternative has been provided in the metaprogram) halts the development process; a designer may then direct the development by hand by changing or adding new PADDLE procedures, or hand-invoking PADDLE commands. There is apparently no facility for backtracking; to "undo" the kth applied

[8]Certain planning systems such as FORBIN [DFM90] also seem to overload action names to also represent desired effects. This is reminiscent of the use of functional specifications for a true specification. This idea can work well when the functional specification is very abstract, but functional specifications in a low level language have many consequences which are usually irrelevant to the task at hand.

transform, one simply executes the PADDLE program forward from the beginning until $k-1$ transforms have been applied. Since transforms are distributed over the procedure, it is not clear just how this counting process takes place.

PADDLE provided the initial inspiration behind TCL, including the notion of subordination of "goals" (TCL *CALL* command), and conditionals (*ELSE*). TCL deviates from PADDLE by insisting on transforms as monolithic entities so that interactions between transforms can be reasoned about (see Chapter 7) without having to know how to extract a transform distributed across the body of a command. TCL postconditions express the goals that a method is supposed to achieve, so that the goal structure that drives an implementation at least has a chance of being extracted from a TCL implementation process. TCL backtracks on failed methods instead of blocking in an attempt to automate more of the implementation process. Selection of TCL methods by means-ends analysis should allow a method to be used in contexts not exactly chosen in advance, whereas PADDLE commands can only be used in circumstances designed in advance. In fact, a PADDLE metaprogram may attempt to apply a complex transformation inappropriately, simply because it has no performance goal condition to prevent it, nor any way to determine after such an inappropriate application that the attempt failed.

REFINE Language and compiler

Smith [SKW85] describes REFINE, a transformation system that, like Gist, uses a high level wide spectrum language called V. Unlike Gist, there does not seem to be an explicit requirement for executability at the specification level. The language provides declarative structures using predicate-calculus notions and sets, procedural notions (conditionals, function calls, loops, arrays, etc.), a special datatype used to construct abstract syntax trees, and a tree-transform operator. Functional specifications are written using the declarative structures where possible. Such specifications are converted directly into the tree data structures understood by the REFINE system. The tree data type and the transformation operator do not appear to be intended for general specifications; rather, they appear to be in the REFINE language only so that transformation control mechanisms can be coded in REFINE directly. This way the REFINE system need only support one language.

The pattern-directed tree-transforms allow one to state the desired form of the resulting tree, and REFINE will find some way to transform the tree to match the desired result. A typical example is to state $a \in b \Longrightarrow a \notin b$; this causes a $DeleteElement(a,b)$ operation to be inserted into the syntax tree. The ability of REFINE to satisfy such requests is unclear, but it apparently only works for very low level transformations. One is also allowed to code arbitrary conventional tree-to-tree rewrites with these transforms.

Transformation control in REFINE is essentially procedural. A single primitive transform can be applied, or a low-level REFINE procedure can be coded to apply some set of transforms in an arbitrary fashion. Special built-in procedures allow a sequence of transforms to be applied in order, or to repeat a set of transforms starting at the leaves of tree working up, or at the root working down to the leaves. The REFINE compiler is apparently coded as a very large set of such procedures that transform high-level REFINE code through several phases and ultimately into Lisp. The fact that the REFINE compiler is coded in REFINE provides a convenient bootstrap. No explicit notion of domain is used to organize the transformations, but recent work [Rea86] seems to be providing domain-tags in the form of class-entities as containing-super-types of objects to be transformed.

An interesting but apparently little-used facility in REFINE is the ability for the transformations to "explore" partial implementations and backtrack if they prove to be unpromising [Kan79].

TCL treats both state and transforms as primitive objects rather than complex data structures. Nearly arbitrary procedures can be coded using the procedural components of TCL: *APPLY*, *CALL*, and *ELSE*. We will see the utility of restricting TCL operators to a well-defined set when we attempt to reuse a design history.

Goldberg's Metaprogramming System

Goldberg [Gol89] describes a *tactics* (metaprogramming) language to be used with the KIDS [Smi89] enhancement of the REFINE transformation system [SKW85].

Primitive tactics implement the actual transformations, and are implemented as REFINE procedures. Higher level tactics consist of compositions of primitive tactics, predefined control mechanisms such as "sequence", "paralell-execution", IF-THEN-ELSE with a conventional boolean test (calling a REFINE function to get the boolean result to be tested), a failure trap (like TCL *ELSE*), looping mechanisms such as WHILE loops, and tactic procedure calls. All tactics may return multiple results for use by the caller.

Similarities to LCF are claimed, but LCF constructs tactics by use of higher-order functions, which Goldberg's tactics language does not seem to have. This tactics language seems to be most similar to PADDLE in terms of control mechanisms. Its most interesting feature is the notion of "program-part", which is apparently inherited from the underlying REFINE abstract-tree representation of the program being manipulated; a program-part represents a syntactically complete program fragment, and can be passed around as an entity from one tactic to another. A sample tactic is shown in Figure 4.5.

```
Combine-Loops(p: program-part) =
   **let** loop-1: program-part,
       loop-2: program-part,
       combined-loop: program-part
   **in while** exists-combinable-loop(p)
      **do** find-combinable-loops(p) **returns** loop-1, loop-2;
          merge-loops(loop-1,loop-2) **returns** combined-loop;
          simplify(combined-loop) **end**
```

Figure 4.5: Tactic from [Gol89, page 6]

Goldberg's tactics language, like PADDLE, is entirely procedural; unlike TCL, there are no goals to achieve or postconditions defining the effect of a tactic. Apparently the decision to avoid postconditions is conscious, as he believes that specification of postconditions is "unwieldy". Unwieldy or not, we think they are hard to live without, especially if explanation of the final artifact is desired. The notion of program-part shows up in TCL as a locale. TCL's *RETURN* construct was inspired by Goldberg's.

PROSPECTRA

The PROSPECTRA transformation system [KB88, KB89b] is intended to convert specifications in Anna [LvHKB87], a semantic annotation language, into Ada. Higher-order algebras with *functionals* (functions allowing functions as arguments and/or results) provide a unified approach used to specify modules, transforms, and control knowledge [KB89a].

Abstract data types are specified using algebraic specification techniques extended with functionals. (Ada) modules are defined using the properties of the abstract data types they manipulate. Transforms are defined as operators over abstract syntax trees (which are just an abstract data type) and can be given algebraic characterizations in their own right. Control knowledge is encoded as functionals applied to transforms and/or other functionals, in the same vein as LCF (discussed in Section 4.3.3); as an example, a MAP functional can apply a simplification functional to the enumerable components of a particular program. This scheme is much nicer than LCF's in that algebraic specification of control functionals can also be given. This offers the possibility of specifying the control knowledge algebraically, allowing one to reason about it, and even implementing functionals that meet the specification. Control knowledge treated as functionals leads to the perspective that control procedures are really just more complicated transforms. It is claimed that working with functionals leads to a higher degree of abstraction, with repetitive processes reduced to application of homomorphic extension functionals. Uniformity of definition allows both transforms and metaprograms to be defined using the same approach; in fact, the control language is a subset of the transform language.

TCL does not provide any functional features, although we have no fundamental objection to them. However, it is paramount to a control language like TCL that methods be described in terms of their effects. While the PROSPECTRA control language does not offer this facility directly, it would seem to be relatively easy to engineer using the algebraic specification tools that are integral to PROSPECTRA[9]; this would seem to be a promising avenue of research. As it stands, the PROSPECTRA control language is purely procedural. TCL allows non-procedural execution.

There seems to be nothing similar to the notion of locale in PROSPECTRA. This absence may be due only to the sketchiness of the available literature.

LCF

Any system for generating proofs is a kind of planning system; the emphasis is on the construction of a proof (a path from the antecedents to the consequent) and not on the final result, which is presumably known before the proof process starts. LCF [Pau87] is a remarkably simple proof construction system in which control procedures are built on top of a functional programming language ML [Har86, HMM86], based on the notion of *tactic* and *tactical* for backwards inferencing. It has been used to construct very large proofs, on the order to 10^6 inferences [Pau87, p. 10], so its techniques should be usable for large scale control necessary for transformation systems. We go into rather more detail because this system is so unique.

Theorems (LCF's version of program schemes) are encoded as syntax trees representing $PP\lambda$ statements, a kind of logical formalism, of a form

$$assumptions \vdash conclusion$$

An operator in this space is a logical inference rule, which is procedurally encoded as an ML function mapping theorems to theorems.

A tactic is an ML function applied to a goal theorem; it is supposed to determine a possible proof of the theorem by decomposing it. Each tactic returns two values, the first being a list of subgoal theorems, and the second being a function which combines the subgoal solutions into a complete solution (i.e., is an inference rule). The full

[9]In a similar vein, we briefly considered the notion of performance algebras to allow coupling of TCL transforms to their effects.

procedural power of ML can be used in the decomposition process provided only that the tactic's inference rule properly re-composes the decomposition to produce the argument.

If a tactic cannot decompose its argument, it can signal an exception; another tactic at a higher level can catch the exception. TCL handles failed methods via its *OR* and *ELSE* sequencing primitives, as well as by alternative methods with identical postconditions. Both LCF and TCL seem relatively unique in the planning world in having conditional plans.

Tacticals are ML functions that map tactics into tactics. An LCF tactical *ORELSE*, taking two tactics and applying either, can be implemented by applying the second tactic if applying the first tactic produces an exception. Much more complex tacticals can be built, including *REPEAT*, *THEN* and list generalizations such as *EVERY* and *FIRST* by simple variations of this idea. Much of the power of LCF tactics (as with PROSPECTRA control mechanisms) stems from the ability to pass (tactical) functions as arguments and apply them. TCL has no such ability.

LCF tactics and tacticals correspond to TCL methods, but, being totally procedural, have no postcondition stating their purpose. This is a major problem if one wants automated control, because that tactics cannot be reasoned about conveniently or, for maintenance purposes, incrementally replayed. This means that no automated tool can conveniently combine a set of tactic(al)s to provide a proof automatically; the tactics controlling an entire proof must be assembled by hand.

A big advantage to ML tactic(al)s is that new ones are easily coded, so the goal decomposition rule need not be fixed as it is in TCL.

SPECIALIST

The SPECIALIST system [Kib78] simplified Algol-like programs when given input data constraints. A typical application of SPECIALIST could reduce a general matrix multiply, with an input constraint that one argument was an identity matrix, into a matrix copy routine. Knowledge about input constraints is converted into special transformations and thereafter treated identically with other transformations. Control of the application of transformations is by dynamic chaining. Dynamic chaining requires that each applicable transformation be decorated with procedures to generate lists of other transformations that could apply if the current transformation was successful, and where they would apply, relative to the application of the current transformation (this corresponds to TCL locale moving operations). Successful application of a single transformation then suggests others to apply via dynamic chaining; SPECIALIST could apply up to 90 transformations by itself this way. Carrying this idea of pointing out potential next applications to an efficient extreme leads to the notion of metamatching as used in Draco. TCL can accomplish the same effect by defining a locale relative to the current locale in which simplifying METHODs can be applied. Implicit in control by dynamic chaining is the assumption that whatever the chain of transformations is doing is what is desired; the implicit goal for SPECIALIST is code simplification. An important difference is that SPECIALIST ties the transforms directly to their intended use, while TCL methods decouple the transforms from intent.

Zap

Feather's Zap system [Fea79] transforms a program consisting of sets of "inefficient" functional equations (a form of functional specification) into a more efficient set of functional equations. Transforms consist of equation definition *unfolding* (substitution of body for call) and *folding* (substitution of call for body) rules. The key idea for transformation control is to provide Zap with goals for the "shape" of intermediate functions, and let Zap determine the actual function by applying a number of lower-level transforms on its own. Rather than being a true metaprogramming language, the intention was to remove much of the burden of applying individual transforms manually. Goals are specified by writing a functional equation containing pattern variables with constraints over their instantiations defined by a surrounding "*CONTEXT*". Given a goal, Zap nonprocedurally finds a sequence of unfolds, built-in simplifications, and folds, that produce a functional equation satisfying the pattern constraints. Rule filtering is virtually nonexistent, because the transforms used by Zap are so few: fold and unfold. Data filtering (which equations are folded/unfolded) occurs by defining such goals in *CONTEXT*s, which allow the specification of which equations may be (un)folded, and what function (equation) names are legitimate for use in instantiating the patterns. Transformational implementation consists of Zap satisfying a series of externally-defined goal equations defined by a corresponding series of *CONTEXT*s. Feather provides some hand-hueristics for choosing the goal equations, but these are not expressible in Zap. One additional transform rule is the deletion of useless named equations from a state. One achieves the effect of a complete metaprogram by linearly reading a disk file containing *CONTEXT* and goal-equation defining commands as well as equation-deleting commands. All of the *CONTEXT*s defined seem to be very specific to the actual problem being transformed because goal equations must necessarily specify an intermediate, problem-specific equation. It is consequently difficult to believe that general-purpose *CONTEXT*s can be easily defined. The problem seems to be that goals are defined in terms of the exact function to be computed.

TCL allows the entire metaprogram to be defined as a set of cooperating methods. Locales provide data-filtering, and controlled invocation of transforms provides rule-filtering. Goals are defined in terms of ultimate problem performance. It remains to be seen whether intermediate TCL goals must be defined in terms of function.

Glitter

The Glitter system [Fic80, Fic82, Fic85], like Zap, is used to automatically apply mundane transformations needed for a major implementation steps. The intent is that the designer specifies major desired effects, and Glitter applies "conditioning" transformations as needed to make the major transformation applicable.

It accomplishes this through use of a language for stating "*transformational*" (as opposed to performance) goals such as *OPTIMIZE*, *DEVELOP*, *GLOBALIZE* and *REFORMULATE* applied to entities existing in the current *program*. Glitter satisfies transformational goals by finding *methods* which can achieve them. Each Glitter method has a goal slot (like TCL's *postcondition*), a filter slot (with effect similar to TCL's *REQUIRE*(*condition*)), and an action slot, specifying some action which will help achieve the goal. Posting a method causes Glitter to collect all methods which can possibly satisfy the goal by matching goal slots (note the similarity to the production system problem of conflict resolution). A separate knowledge base of *selection rules* chooses between the candidate methods by inspecting the current state for interesting features and voting for or against candidate methods; the method with the most votes wins and is executed. Glitter achieved an order-of-magnitude reduction in the number of designer selected transforms required for an implementation.

Glitter often had to query the designer about interesting features; TCL provides access to such features via arbitrary predicates. References in Glitter to entities in the program are by name of entity in the program; this appears to be a sort of symbolic locator. TCL allows each method to decide for itself if it is applicable, and needs no other mechanism to make the choice. Glitter requires the separate selection rules, and can choose among many methods before trying any one of them.

TCL chooses candidate methods via postconditions in much the same way as Glitter. A difference is in the vocabulary used to define the postconditions; Glitter allows certain informally defined, approximate process predicates such as *OPTIMIZE*. We feel uncomfortable with this, and have currently chosen to avoid such predicates, although they would be simple to add to TCL with the same sort of operational semantics they have in Glitter. The selection rules used by Glitter to choose between methods strike us as unneeded; they are obviously measuring something, and if what is being measured is not process information, then some performance predicate should be able to do the job.

Even assuming the existence of a tool such as Glitter, there is still a need to specify the major transforms to carry off an implementation as a metaprogram, if nothing else, for documentation purposes.

Having considered a number of metaprogramming systems, we now turn our attention to comparing TCL to planning systems.

4.3.4 Domain independent Nonlinear Planners

Classical domain-independent nonlinear planning is defined as the problem of determining a set of operations (operators plus bindings) and a partial ordering over that set specifying constraints on order of application, that changes a given initial state into a final state with specified properties (usually termed goals) [Kam89, p. 11]. An introduction to planning can be found in [CM85]. A thorough formal analysis of nonlinear planners is provided by Chapman [Cha87]. An excellent collection of papers on planning in general can be found in [AHT90].

When transformation systems are characterized via performance goals, mechanisms used in classical planners seem obviously relevant; both have goals stated in roughly the same way, and the fundamental problem for both is finding a path to reach a goal state. The notions of plans and subplans for achieving a purpose are so natural that we find it hard to imagine a control system like TCL without them, and indeed, even procedural metaprogramming languages such as PADDLE [Wil83] have the idea of subplans in the form of procedure calls. TCL was not designed with the intent of advancing the state of the art for planners, but rather with the intent of using available planning technology in a transformational context. As a result, TCL as a metaprogramming language is unique in connecting each plan explicitly with its purpose as a postcondition. Another property of planning systems is the need for replanning in the face of plan failure. Such techniques can possibly be used for transformational maintenance; we shall return to these ideas in a later chapter, but their use in conjunction with planners provided some of the impetus to define TCL in a planner-like way.

The value of a nonlinear plan is simply that unnecessary sequencing constraints are not present [Sac74, Sac77]; this is closely related to the idea of a dependency net [Fik75, Lon78]. While we do not consider dependency nets in this thesis, we expect them to prove valuable in enhancing the maintenance process as an aid for revising derivation histories. We consequently assumed that TCL should be designed so that generation of such nonlinear plans was possible. We have incorporated the notion of nonlinear plan directly into TCL as the *PLAN* construct.

Hierarchical abstract planners [Geo87, Kor87, Sac74] perform planning not only in the target problem space, but also in abstractions of the target problem space. The idea is that it is simpler to solve problems in a simpler space, and an abstract solution can be used to guide the construction of a concrete one by combining solutions in the concrete space to problems defined by pieces of the abstract solution. Usually abstract spaces are formed by weakening the goal predicates, many times by simply dropping obscure but assumed-easily-achieved terms. In a transformational context, an abstract space might be formed by simply dropping some of the performance predicates comprising G_{rest}; solving a problem in this space would produce a nearly satisfactory program. The resulting solution could perhaps be "tuned" to meet the other performance goals; this corresponds to operating in the target problem space. TCL does not implement strictly hierarchical planning, nor does any other transformation control system known to us. A key problem is identifying performance goals that are "easily achieved" so that abstraction spaces can be formed.

How Transformation system control is different than Classical Planning

Transformation system control is similar, but not identical to classical AI planning. We outline the differences and how those differences affect TCL, and transformational control in general.

Scale: Many state-of-the-art planners solve problems with only tens or hundreds of steps [CT85, Kam89, Wil88]; transformation systems must deal with tens of thousands of steps. We see that transformation systems must handle problems that are orders of magnitude larger than current ambitions for planners. Transformation systems must focus their attention more tightly to prevent scale from simply overwhelming them; thus the TCL notion of locale as a device focusing attention on a region.

Representation Change: Planners and transformation systems differ in the representations used to describe initial and final states. For conventional planners, the properties of the desired final state are usually stated in the *same* terminology which defines the initial state, and is used to describe operators: as a set of propositions about relations between objects in the world [Cha87, CM85]. Typical is the predicate $ON(x, y)$ to represent the knowledge that a block x is on top of another block y. Planning languages for such planners use this terminology directly, and are thus committed to a particular representation. For transformation systems, the representation of the initial state f_0 and the final state are likely to be very different; consider an f_0 stated in functional programming terms, with the final state satisfying $g_{FORTRAN}$.

The transformations will use terminology at the same level as the current state (as does Draco [Nei80]).

Rather than commit TCL to representations for a specific transformation system, we have chosen instead to *APPLY* the transforms by name. A consequence of the notion of monolithic transforms are monolithic locaters. In conventional planning systems, locaters and locales show up as constraints on operator arguments. For the blocks world operator $ON(x,y)$ the constraints $blue(x)$ and $y = BLOCK973$ are a locale. Use of a bound variable (say, y), in a plan corresponds to a locale expansion operator (weaken the locale by dropping the constraint $blue(x)$). Like planning systems, we do require TCL to represent goal predicates consistently across states.

The frame problem: Current planners and transformation systems differ with respect to solutions to the frame problem: controlling the ripple effect on the world of caused by changes to specific facts. Most planners operate under a STRIPS representation (usually limited to ground logical formulas) and STRIPS assumption [Lif86]: only facts changed by the operators change in the world. The world is represented by a database of currently-true relations between individual objects. Queries are almost always satisfied by direct inspection of the state for the relations in the queries. This is effective because planning situations are generally involved with the physical movements of objects, and the interesting queries are generally about how one object is placed with respect to another. Rarely are questions asked about derived properties of configurations of objects, such as "How high is this stack of blocks?"; the emphasis seems to be on objects as individuals. With transformation systems, it is not convenient to represent all the possible facts in the current state; properties of portions of the state are frequently of interest ("How fast is this subroutine?"). It is therefore difficult to retain the STRIPS assumption for transformation systems. As a consequence, conditional transforms may require considerable energy to validate. In a transformation system, there doesn't seem to be any special emphasis on objects; rather, properties of structures are of interest. If objects do exist in transformation system representations, they tend to be more anonymous (i.e., the operator "+" can be considered an object, but it is freely exchangable among all its instances). The notion of locale as choosing semantically interesting regions of a program for *REQUIRE* and *ACHIEVE* is necessary to describe structures whose properties are interesting to extract. As planning systems become more ambitious, we expect the emphasis to shift towards configurations of objects, and so the differences should diminish.

Our perspective is that transformation systems handle the frame problem by having transformations map states, and use performance measures to project the state into performance values. This is rather like Georgeff's characterization of an extended STRIPS representation [Geo87, page 15] with states containing basic facts, operators manipulating only the basic facts, and planning predicates computing derived facts from the basic facts on demand. TCL does this by referencing performance goals; costs of evaluation are partly kept down by transformation system strategy of only applying property-preserving transforms, assuring that at least one derived (very complex) property need not be recomputed at all. In our transformational model, the cache component of states is intended to keep cost of evaluating other predicates low, but TCL does not specifically help here. Other planning systems such as SIPE [Wil88] and FORBIN attempt to separate derived facts from basic facts, and provide special, eagerly-evaluated inference rules to update derived facts, that trigger on detection of changes to classes of relevant basic facts. The classic example is eager inference of the derived fact $\neg CLEAR(x)$ when an operator produces the new fact $ON(x, y)$ in the blocks world. We do not believe there is much value in this solution in transformation systems; the scale of states is likely to make such eager inference of all possibly-referenced derived properties unreasonable. An interesting unexplored possibility is how to compute derived facts on demand using what remains of a previously valid analysis based on dependency nets.

Usable Transforms/Operators: Both planners and transformation systems operate with a fixed set of operators at any instant. From the point of view of the planner and the transformation system, the set is completely arbitrary. External to the planner and the transformation system, the usable operators are limited to those that make sense. For a planning system, the operators allowed are those which model some world; a blocks-world *PUTON* operator is not expected to explode the block. For transformation systems, there is also the need to use just $G_{invariant}$-preserving transforms at any point in time.

In the face of scale, this is actually an advantage because it limits the applicable set of transforms. Planners do not have this constraint.

Often, for planning, simple means-ends analysis can compare the current state and goal descriptions to determine a likely candidate operator. For transformation systems, any means-ends analysis must check the consequences of a proposed operation on the observable effect of the other performance measures, so it is harder to determine transformations with desired effects; a theorem prover may be required to do means-ends analysis to choose plans.

Purpose of Planning vs. Purpose of Transforming: A transformation system and a planning system differ in their ultimate purpose. A planning system is given a specific, possibly partial, target world configuration, and is tasked to find a sequence of operations that when executed actually achieve that configuration; the total ending world state is usually not of any particular interest. The emphasis on applying the operations has to do with the need to truly move objects around in the world. A transformation system is not given a specific target world configuration, but only a way to recognize a desired world. It must also find a transformation sequence, but unlike planning, the sequence is not the point[10]; the complete final state is the result desired. This difference in emphasis seems have have had little effect on planning versus transforming methods, to date, but see the discussion on resource management that follows. It is the shared need to find a sequence that makes many planning mechanisms applicable to transformational control, and thus to TCL.

Resource management: Recent work in planners includes resource management: how to choose a plan that stays within problem-domain resource bounds such as total time to execute a plan, consumable supplies of objects, or total operator costs. Deadlines are handled by FORBIN [DFM90]. Consumable resource and recyclable supplies are considered by SIPE [Wil88]. Resource management problems also appear in the planning process itself.

In planning systems, problem domain resources constrain legitimate sequences of operations, whereas in transformation systems problem domain resources constrain legitimate states (programs) but not sequences of transformations. If one considers transformation systems as planning systems that produce plans (for computing values, i.e., programs), the problem domain resource constraints appear in the same place.

Describing and managing such problem domain resources requires explicit specification of those resources and how they are consumed, as well as providing special mechanisms for handling resource interactions. Conventional transformation systems do not even address the descriptive aspect of resource management. TCL handles this indirectly by performance goals built on performance measures such as $p_{complexity}$ (a time resource measure).

Process-domain resource bounds such as total planning time or external restrictions on allowable sequences of operators (such as length) are just beginning to be considered. We have avoided handling process aspects entirely in our characterization of transformation systems and TCL, but it is clearly important in the long run to be able to place bounds on resources consumed by a transformation system and yet still produce an effective program product; this is the purpose of software engineering. Because of the similarity of planning and transformation systems, we expect that planning research on resource issues will be transferable.

[10]although it is critical for maintenance purposes!

FORBIN

The FORBIN planner [DFM90] is typical of many nonlinear planners in terms of its representation of plans. The space of operators is broken into three parts:

- Task invocations (task name plus list of objects used by task)
- Task descriptors (expected postconditions of tasks)
- Task bodies (plan for accomplishing task)

Task descriptors and task bodies together act as the equivalent TCL methods. Operators (task invocations) in the abstract space seem to be poorly modeled in FORBIN, as they can match different task descriptors with different postconditions.

TWEAK

TWEAK [Cha87] is, among other things, a constraint-posting planner. The size of the search space is reduced by constraining bindings on objects referenced by operators until unique objects are selected.

The notion of "object" does not make much sense in transformation systems, so the utility of such constraints for transformational implementation is unclear. One might be able to apply such constraints to implementation domain-specific notions of reusable resources, such as variables in conventional procedural languages.

SIPE

The SIPE planner [Wil88] can handle resource management, including what amounts to cost of plan steps (a plan step that decrements a fuel resource by a plan step dependent amount). Remaining fuel is tantamount to a process measure; a predicate testing for positive fuel remaining would be a process predicate. Since our model of transformation does not consider process predicates, TCL has no support for them. SIPE's approach might be a good place to start.

As well as providing for goal decomposition by application of methods with postconditions, SIPE has a set of built-in critics for resolving plan bugs: parallel interactions, phantom goals, ordering constraints, etc. TCL has no critics, but this may be an artifact of complete versus partial states. Since critics make choices about how to resolve plan inconsistencies, they amount to implicit control.

SIPE provides methods, complete with bodies, called "plots", and postconditions, much like those of TCL. A *CHOICEPROCESS* action corresponds to TCL's *OR* node.

SIPE also allows a human agent to interactively modify a partially constructed plan via a graphical interface, with the intention that the human can redirect the planner away from poor solutions, thus making the result plan more effective. Such an interactive process is likely to be of use in a practical transformation systems. The KIDS transformation system interface [Smi89] allows interactive specification of individual transformation steps, but apparently not any overview or modification of the design plan, as there is no explicit representation of that design plan. TCL provides the basis for storing an explicit design plan (called a design history, see Chapter 5, so it is possible to contemplate such an interactive interface.

In an attempt to avoid the frame problem, SIPE operators specify only *main* effects, with *causal deductive theories* deducing *side* effects from generated main effects. SIPE's planner ensures that main effects of parallel plans are preserved, but does nothing for side effects. To define performance measures or predicates necessary for a transformation system, one would need to use side effects in SIPE. The fact that SIPE pays little attention to those effects would make it a poor system for implementing at transformation system.

4.4 Open Problems

Work on explicit control mechanisms for transformation systems is just getting started; the earliest work we know is [Wil83]. There is little real experience with metaprogramming, or understanding of what techniques will be the most useful. Consequently there a number of obvious open problems:

- What performance measures are useful in practical metaprograms?
- What are useful forms for and operations on locales?
- How can one take resource costs into account in the navigation process?
- How do we acquire metaprograms?
- How effective will functional metaprograms be?
- How can we combine functional metaprograms with nonprocedural metaprograms?

We do not address these issues further in this thesis.

4.5 Summary

In this chapter, we defined requirements for a metaprogramming language. We observed the utility of planning systems for metaprogramming, and described the essential features of a metaprogramming language, TCL, based on those ideas. Locales are identified as a means of focusing the attention of the transformation system. TCL draws on ideas from planning: nonlinear plans, and method postconditions, which in turn requires the usually-implicit G_{rest} be made explicit, The value in such postconditions is that they provide a link between the actions of the method and the interesting effects caused; this information is needed for maintenance purposes as well as controlling the metaprogramming process. Examples showing the utility of TCL were provided, including an example covering one of the few existing transformations systems that use explicit performance predicates. A comparison to navigation techniques used by other systems was made.

Contributions:

- TCL, a metaprogramming language that associates plans with postconditions
- A metaprogramming language consistent with the model of transformation systems based on performance predicates
- The notion of *locale* as first class value in TCL constraining application of transform
- A concise set of primitives for defining TCL-like languages
- An analysis of strengths and weaknesses of TCL with respect to other control systems

Chapter 5
Design Histories

Chapter summary. A trace of the execution of the transformation system, especially that of the control mechanism, provides a *design history*. This is useful for explaining how an implementation was achieved, and what role each transformation played in the process. The content of a design history is examined. These structures will guide and be modified by transformational maintenance.

Execution of the transformation system produces not only an implementation, but also a *derivation history*: the sequence of transformations that were actually applied. It is important to record, for explanatory purposes, not only the chosen derivation history, but also the motivation for each transformation in the derivation history: the design history. The design history will be of considerable use to maintainers for understanding, and to tools for revising the constructed artifact. Many transformation systems suggest the value of a similar output [Bau77, CTH79, Nei80, BMPP89] but, by and large, it is not produced in a usable form. For our work, it is essential.

In this chapter, we describe the structure of both the derivation history, and the enclosing design history, constructible by tracing the dynamic execution of TCL metaprograms.

5.1 Kinds of Design Information and Reuse

Possession of the "design" of an artifact is essential if one wishes to make changes to it. Consequently we must capture design information in some form. We must choose a particular form.

Given a definition of a design as the justification of a transformationally constructed final artifact, we perceive three different possible kinds of design information:

- Derivational: Sequence of transformations applied to achieve result
- Motivational: Structured justification of derivation
- Generative: Executed to generate a sequence of transformations

The derivational design information tells *procedurally* how the (various parts of the) final artifact were derived, by specifying exactly the sequence in which of transforms (and their locaters) were applied. Such information is essential when determining the impact of changing a transformation. It can also be used for naive replay by simply attempting to re-execute the sequence of transformations; such replay has the distinct advantage of being fast, in that no decisions regarding what transform or where to apply it need be made. *Such naive replay always works to some extent because the transformations are supposedly "correctness"-preserving.* Consequently, if a transformation from a derivation history can legally be applied, the result is by definition legitimate; there is simply no guarantee that any particular replayed transform does any useful work with respect to desiradata of the new artifact. What is missing is from such naive replay is understanding of the role the individual transformations play in achieving performance goals.

Generative design information contains not the design information for the artifact, but the *potential* to generate that design information. It usually takes the form of a *metaprogram*, which is really a mechanism for guiding the transformation system, by telling what transformations to apply where; thus it generates derivational design information. While metaprograms have the advantage that they are easily replayed, by simply re-executing the transformation system with the same metaprogram, they have the disadvantage of requiring that re-execution in order to rediscover the transforms and locaters needed. This is a major cost we wish to avoid when performing maintenance. In practice, transformation systems will require metaprograms to guide them anyway, so this is certainly an attractive form.

A design plan (or design history) includes not only the derivation history, but also structures the derivation according to the effects that the parts of the derivation are expected to achieve. It also records, along with the structuring information, the purpose of the particular structure in the form of subgoals. As it includes the derivation history, it can be used for naive replay by simply attempting to execute the derivation history. A more reasonable scheme will also re-validate the applicable transformations according to the recorded information in the design plan to ensure that they have the desired effect. A particular advantage of this form is that the scope of effect of a particular transformation is more easily determined by examining the subplan structures in which it takes part. A considerable gain over generative (dynamic) replay comes because reuse of a design plan means that the transformation system need not try to determine either the transforms or the locaters to use for much of the resulting artifact. *Use of a design plan has not been applied in transformation systems to date, and is one of our major contributions.* In fact, a metaprogram can generate design plans if properly organized; a good metaprogramming system will blur the boundary between dynamic execution of the metaprogram and and the static design history of a particular artifact, allowing design plan repair to fall back on parts of the metaprogram as needed. *Our TCL metaprogramming language was* *designed to achieve this purpose.* We will discuss the execution model for TCL in Chapter 8.

We keep all three of these types of design information for an artifact. We retain the derivation history because of our expectation that most design decisions (applied transformations) will remain intact in the face of small changes. We keep the design history so that those design decisions which appear to be unaffected by a change can be validated to ensure they still continue to serve their intended purpose. Lastly, we keep information about how the design history was generated as the TCL metaprogram in case parts of the design history need to be regenerated.

5.2 Derivation Histories

A derivation history captures the precise path through the design space traversed by a transformation system. This path *is* the construction information for the final artifact. Should we wish to construct a similar artifact, a similar path is likely to be needed. Thus considerable information is likely to be available in the current artifact's derivation history.

DEFINITION 5.1: *Derivation History.* A sequence of transformations[1]. We denote a derivation history H of length k by $H = [t_1^{\ell_1}, t_2^{\ell_2}, \cdots t_k^{\ell_k}]$. Alternatively, we may denote a derivation history by a triple $H = \langle k, H^{\mathcal{T}}, H^{\mathcal{L}} \rangle$, where $length(H) = k$ is the length of the history, $H^{\mathcal{T}} : 1..k \rightarrow \mathcal{T}$ and $H^{\mathcal{L}} : 1..k \rightarrow \mathcal{L}$, are functions which generate the individual transforms and locaters representing the history. Thus, $H = \langle k, H^{\mathcal{T}}, H^{\mathcal{L}} \rangle$ can be written as:

$$[t_{H^{\mathcal{T}}(1)}^{H^{\mathcal{L}}(1)}, t_{H^{\mathcal{T}}(2)}^{H^{\mathcal{L}}(2)}, \cdots t_{H^{\mathcal{T}}(k)}^{H^{\mathcal{L}}(k)}]$$

□

We use $\mathcal{H}$ to represent the set containing every possible derivation history.

[1]Both Carbonell [Car85] and Mostow [Mos85c] use the term "derivational" to include the notion of goals. We prefer to use it in the stricter sense of "derived from", being a purely mechanical process without motivation.

In practice, we expect each derivation history to include both property-preserving transforms and non-property-preserving transforms, because of the practice of constructing f_0 from the *empty* specification ϵ (a possible instantiation of ϵ would be **skip**) by application of non-property-preserving transform ns [JF90, Fea89a, Fea89b]:

$$f_0 = n_j^{\ell'_j}(n_{j-1}^{\ell'_{j-1}}(\cdots(n_1^{\ell'_1}(\epsilon))))\cdots)$$

The value of the ns lies in their use for "elaborating" an initial functional specification to include details not covered by the initial specification; we will see these later as functionality deltas.

An implementation f_G is achieved by applying property-preserving transforms:

$$f_G = c_k^{\ell_k}(c_{k-1}^{\ell_{k-1}}(\cdots(c_1^{\ell_1}(f_0))\cdots)$$

Consequently, a derivation history can have the structure:

$$[n_1^{\ell'_1}, n_2^{\ell'_2}, \ldots, n_j^{\ell'_j}, c_1^{\ell_1}, c_2^{\ell_2}, \ldots, c_k^{\ell_k}]$$

Current transformation systems do not produce a derivation history with this form, but are moving in this direction; the ARIES system [JF90] captures just the evolution of the functional specification.

5.2.1 Operations on Derivation Histories

It is convenient to perform various conventional operations on sequences forming derivation histories, both for mathematical description and for actual manipulation. We define the following operations:

- Length: $length(H) = k$ if $H = \langle k, H^{\mathcal{T}}, H^{\mathcal{L}} \rangle$
- Indexing: $H[i] \equiv t_{H^{\mathcal{T}}(i)}^{H^{\mathcal{L}}(i)}$
- Subsequence: $H[i..j] \equiv [t_{H^{\mathcal{T}}(i)}^{H^{\mathcal{L}}(i)}, \ldots, t_{H^{\mathcal{T}}(j)}^{H^{\mathcal{L}}(j)}]$
- Tail: $rest(H, i) \equiv H[i..length(H)]$
- Subset: $H_1 \subset H_2 \equiv \exists i, j \mid H_1 = H_2[i..j]$
- Concatenation:
  $H_1 + H_2 \equiv [H_1[1], \ldots, H_1[length(H_1)], H_2[1], \ldots, H_2[length(H_2)]]$

5.2.2 Compositions of Transformations

We shall have need for notation for the composition of transformations. Since transformations are partial functions from states to states, their compositions are well defined functions.

DEFINITION 5.2: *Composition operator* $\circ : (\mathcal{T} \times \mathcal{L}) \times (\mathcal{T} \times \mathcal{L}) \to \mathcal{T} \times \mathcal{L}$. A partial function composing two transformations:

$$t_1^{\ell_1} \circ t_2^{\ell_2} \equiv t_\circ^{\ell_\circ} \mid \forall s \in \mathcal{S} : \mathit{defined}(t_2^{\ell_2}(t_1^{\ell^1}(s))) \supset \mathit{defined}(t_\circ^{\ell_\circ}(s)) \wedge t_\circ^{\ell_\circ}(s) = t_2^{\ell_2}(t_1^{\ell^1}(s))$$

□

We assume that the representation of transforms and locaters is rich enough so such a composition is well defined.

The product composition operator defines the effect of applying a derivation history to a program.

DEFINITION 5.3: *Product composition operator* Π. Given a derivation history H, the effect of the individual transformations can be composed to form a single large transformation $\Pi(H) = H[\mathit{length}(H)] \circ H[\mathit{length}(H) - 1] \circ \cdots \circ H[1]$ □

Thus $\Pi(H)(\epsilon)$ is the program obtained by applying the entire string of transformations in the history H to ϵ.

Usually associated with each derivation history are a program f_0^H or a state s_0^H from which the derivation history was initially generated. For derivation history H starting from ϵ, $f_0^H = \epsilon$. For any $H[i..j] \subset H$, $f_0^{H[i..j]} = (\Pi H[i..j])(f_0^h)$.

5.3 Design Histories

Most extant systems that attempt any kind of replay use just a derivation history as the replayee [Gol89, MB87, SM84]. Mostow [Mos85c] implies this is not going to be greatly successful as the *justifications* for applying the individual transformations are lost.

The correctness of implementation of a specification must be justified somehow. We call a *design justification* any structure that shows precisely how each step taken by the implementing system is justified in terms of its ultimate effect. Such a structure is essentially a proof that the implementation meets the specification, derived only from the initial specification and the transformation steps used.

5.3.1 Design Histories as Unfolded Goal/Plan Structure

In practice, such a detailed proof is expensive to construct, and of little practical use. All that we really need is a justification of the implementation down to the level of believably reliable steps. With the right kind of control, plan structure comes to our rescue; *proofs of the correctness of compositions of transformations can be replaced by references to methods that achieve the effect by applying those transformations.* We can leave the proof of the method [All90], if we have it, attached to the method itself, thereby conserving on the size of a design justification. Even the hueristic nature of the method need not concern us; since TCL ensures the (untrustworthy part of the) postcondition of a method by actually testing it, we know that a successful method achieves the desired effect in the context in which it is tried, even if it does not work under all circumstances. This knowledge tells us that a proof of the value of the method in this context is possible, even if we do not have the general proof; we don't actually want to construct such proof. The mere knowledge that it is possible is sufficient justification for application of the method.

We can capture a useful part of such a design justification by tracing the execution of a goal-oriented metaprogram. A derivation history and the unfolded execution of a goal-oriented metaprogram are collectively called a *design history.*

DEFINITION 5.4: *Design History.* A structure showing how goals are achieved using plans. □

Coupled with the proofs that plans actually achieve goals, a design history provides us with indirect justification for every transformation present in a derivation history.

A design history is shown schematically in Figure 5.1. Horizontally we see the design states (minus the consequences Q_i, for clarity) coupled by the transformations produced by *APPLY* steps; the horizontal bold arrows form a single path through the design space as shown in Figure 3.7. Vertically we see the performance goal decomposition by use of plans implementing method postconditions; such decomposition is accomplished fundamentally by *ACHIEVE* steps. Each node represents a performance goal to be achieved. A set of arcs emanating from the node represent the decision to carry out that goal by the application of some plan; each arc represents a step in the plan. Dashed arcs represent untried alternatives. Links across arcs represent required sequencing of plan steps. In the interest of keeping the diagrams uncluttered, we have adopted the diagrammatic convention that, unlike interior arcs, each arc from leaf nodes to transformations in the derivation history represents an unshown node whose plan is to *APPLY* a transformation. To denote this, the notation *APPLY* is written explicitly on the leaf arcs in this diagram, but is implicit in later diagrams. Also for clarity, the diagram shows every step in a plan as having a generating goal by the simple artifice of attaching a postcondition of *true*; in a practical design history we do not do this. A successful *REQUIRE* goal or a serendipitiously *ACHIEVE*d goal[2] is treated as if it used a plan consisting of application of an identity transform to those parts of the satisfying state that imply the achieved condition. For our figure, G_3 : is effected by any-order execution of methods for achieving G_4 and G_5. G_4 is accomplished by applying $c_1^{\ell_1}$ and then $c_2^{\ell_2}$. Goal G_5 is accomplished by applying $c_5^{\ell_5}$; an unneeded alternative for solving G_5 is shown by the dashed arrow. G_7 shows a 3 step *PLAN* with only a single ordering constraint. The vertical arrows are drawn in time-order of trace generation; reversing them would produce the design justification information.

A design history in which every node has an implementing plan is called *complete*; if nodes exist which have no implementing plan, then the design history is *incomplete*.

The diagram does not show non-property-preserving transforms used to construct f_0 from ϵ. We believe that such transforms also belong in the design history, along with their justification. The only justification we can use for the collection of non-property-preserving transforms is "the system analyst says this is needed", which is essentially the goal $G_{invariant}$, with an unstated but nonetheless real plan consisting of applying all of the individual non-property-preserving transforms. Materialization of f_0 directly can be modeled as $\Pi(H_n)(\epsilon) = f_0$. An overview of the extended design history taking the non-property-preserving transforms into account is shown in Figure 5.2. Such a complete design history shows how the entire specification G is decomposed into $G_{invariant}$ and G_{rest} to effect the desired result, for those transformation systems which hold $G_{invariant}$ constant. The ordering established under $G_{invariant}$ will depend on interaction properties of the non-property-preserving transforms [JF90]. The derivation history H_c are the transformations produced by the transformation system.

5.3.2 Design History abstract representation

We now look more closely at the representation of a design history. We use the symbol $\mathcal{D}$ to represent the set of possible design histories, with D being an individual design history. While our diagrams uniformly group a goal and a plan together as a node, in practice the design history consists of nodes which may be either. We call such nodes *agenda items*[3], because each represents the potential need for work to accomplish the effect. Each design history D is a set of individual agenda items, designated a_i.

[2]Called a *phantom* goal in non-linear planning terminology.

[3]Similar nodes are generally called "task nodes" in hierarchical planning literature [Kam89]

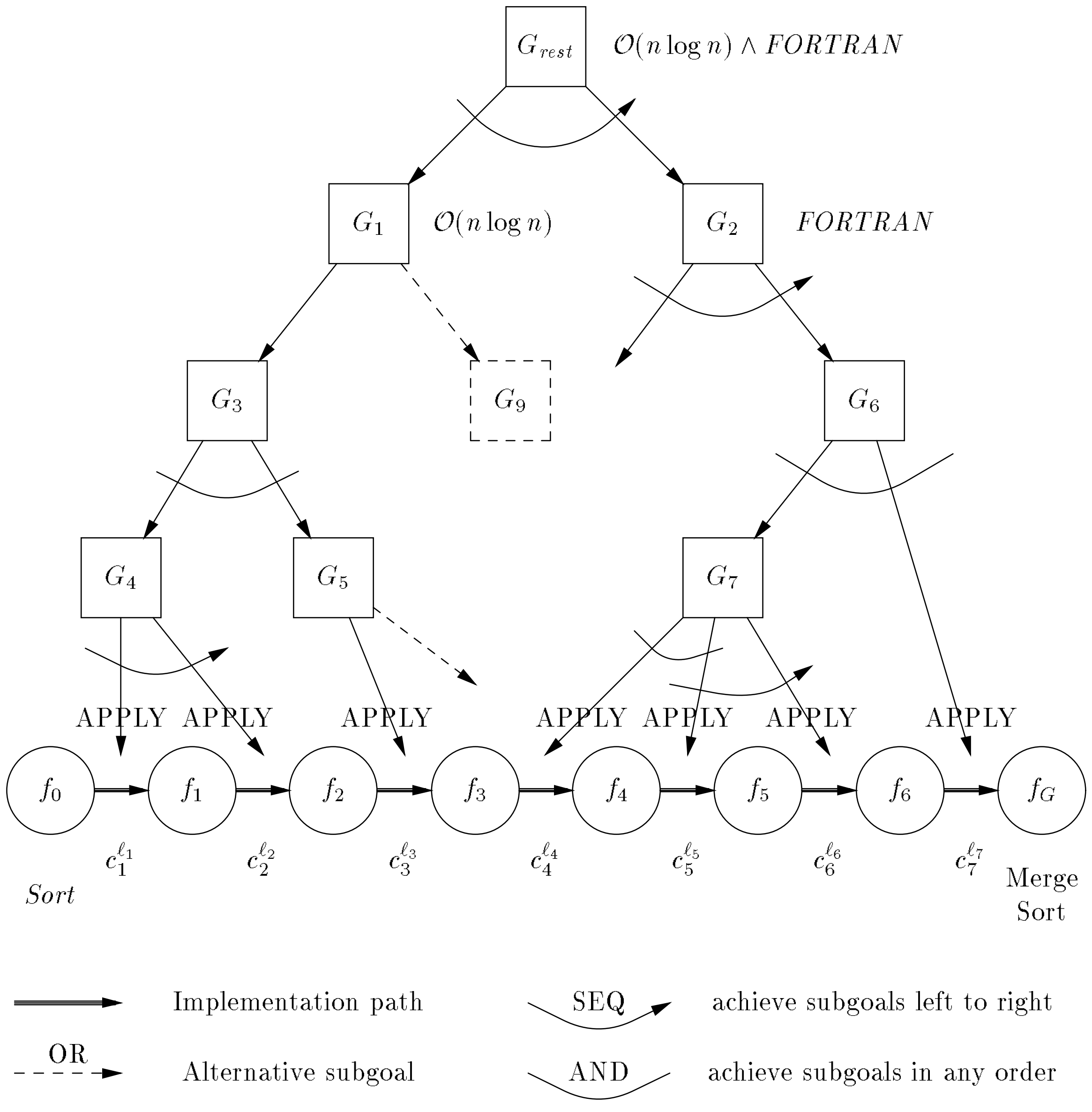


Figure 5.1: Design History as unfolded Method Execution

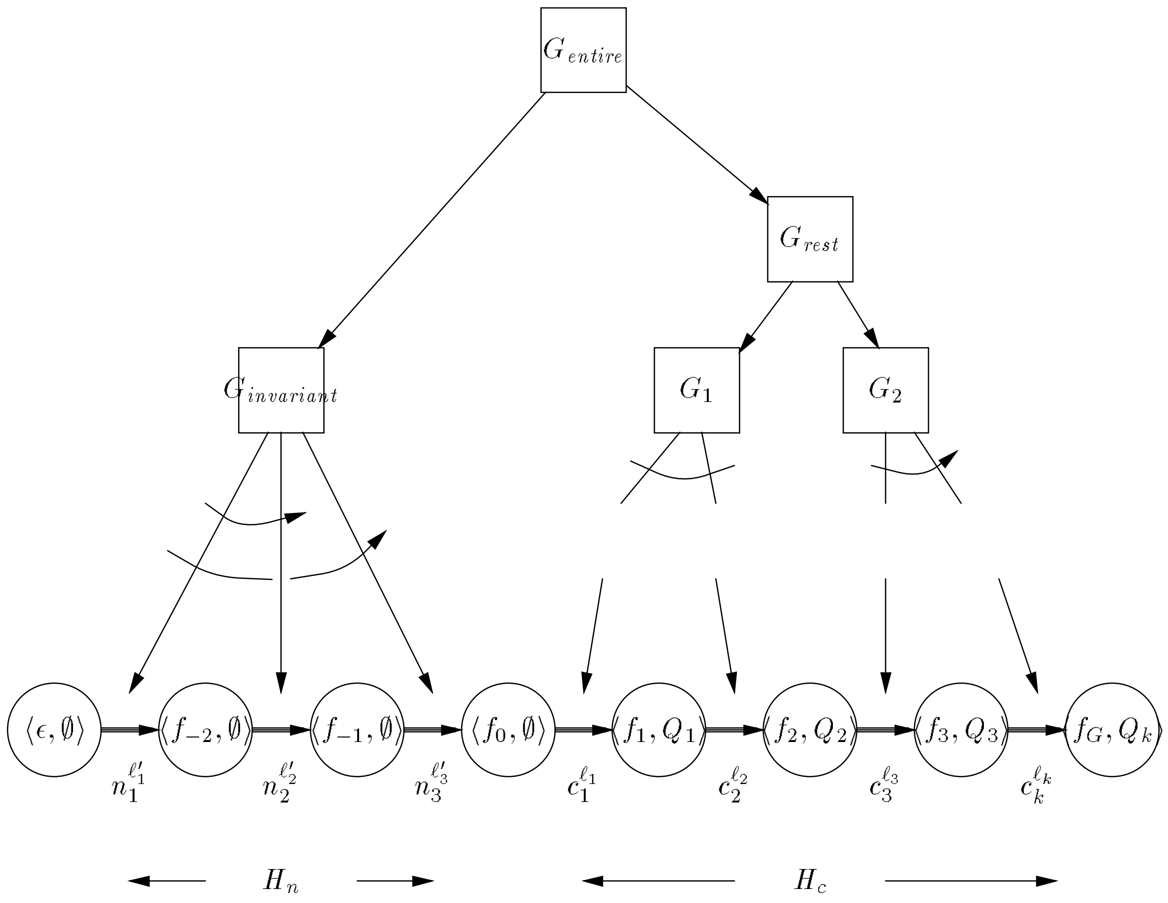

G_{entire}
G_{rest}
$G_{invariant}$
G_1
G_2
$\langle \epsilon, \emptyset \rangle$
$\langle f_{-2}, \emptyset \rangle$
$\langle f_{-1}, \emptyset \rangle$
$\langle f_0, \emptyset \rangle$
$\langle f_1, Q_1 \rangle$
$\langle f_2, Q_2 \rangle$
$\langle f_3, Q_3 \rangle$
$\langle f_G, Q_k \rangle$
$n_1^{\ell'_1}$
$n_2^{\ell'_2}$
$n_3^{\ell'_3}$
$c_1^{\ell_1}$
$c_2^{\ell_2}$
$c_3^{\ell_3}$
$c_k^{\ell_k}$
H_n
H_c

Figure 5.2: Design History including justification for functional deltas

Agenda items stand for instances of actions as outlined in Chapter 4. Each agenda item contains the following information (Figure 5.3):

- $action(a)$, indicating what needs to be accomplished, being any of the TCL actions (defined in Section 4.2.2), including *APPLY*, *ACHIEVE*, *REQUIRE*, *PLAN*, *OR*, *CALL*, etc. One can think of this as a pointer into a TCL method body formed by instantiation of the method according to its parameters ($a_{body}\sigma$). As terminology, we characterize "an a (agenda) node" as one whose *action* aspect is action a; thus an agenda node whose action is *CALL* is termed a "*CALL*" node.
- $sons(a)$, being a set of subagenda items that purportedly accomplish the effect. In the case of *APPLY*, the son is the applied transformation.
- $order(a)$, which is a partial order $>$ over $sons(a)$. If $action(a) = PLAN$, this is determined by the partial order given in the *PLAN* step.
- $completed(a)$, which is *true* if the action required by this agenda item has been accomplished, and otherwise *false*. For example, an agenda item will be marked *completed* if $action(a) = APPLY$ and the transformation has actually been applied, or if $action(a) = ACHIEVE$ and a *sons* and its *order* have been established. We say an agenda item with $completed(a) = false$ is *incomplete*.
- $parents(a)$, which lists agenda items whose sons include a. We define $ancestor(a) \equiv parent^*(a)$. The root of the design history has an empty set of *parents*.
- $symboltable(a)$, consisting of a set of triples $\langle lv, \ell, dependents\rangle$ each containing a locale-variable name (lv), a locale-value (ℓ), and a list of dependent agenda items which use the locale-variable name in a locale expression evaluated by $action(a)$. This stores named locale values for reference by necessarily-following agenda items. Nodes which always contain non-empty *symboltable*s are those with action *LET* or *LOCALE*.

Agenda Item Symbol Tables

The *symboltable* serves two purposes:

- As a mechanism for implementing *LET* and *LOCALE* constructs
- As a dependency net tracing the usage of locale values

When a locale value ℓ is bound to a variable lv by the action of agenda item a, the triple $\langle lv, \ell, \emptyset\rangle$ is added to $symboltable(a)$. For *APPLY* agenda items, a dummy variable is used to capture the locater resulting from transform application. Looking up a locale variable in the context of a particular agenda item a consists of searching

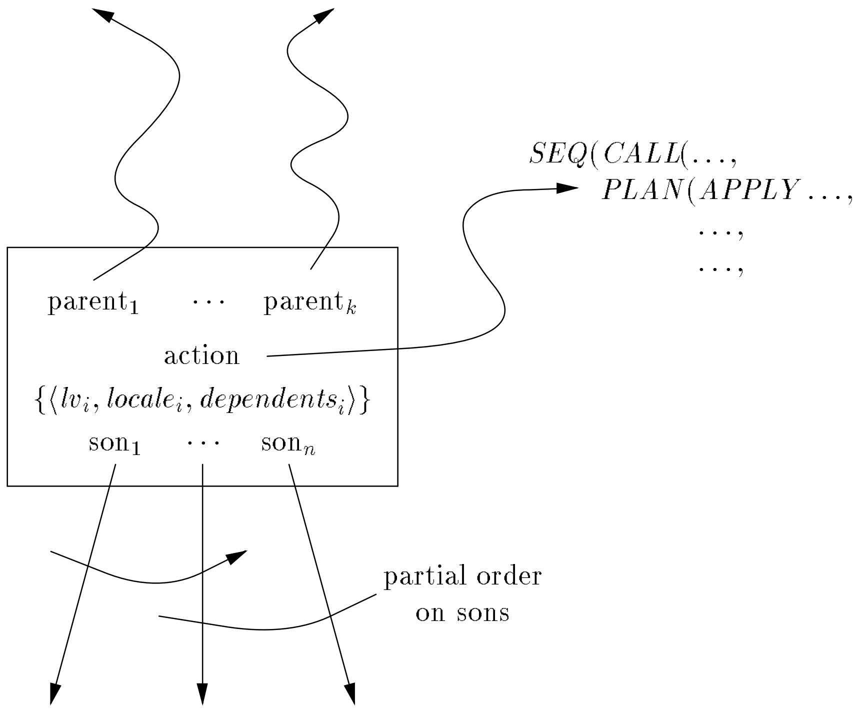


Figure 5.3: Design history agenda item

the design history in reverse order according to *order*(*parent*(a)), continuing the search upwards towards the root when there are no necessarily preceding brothers, until an agenda item containing a *symboltable* with a matching variable name is found; its associated value is the desired value. Whenever agenda item a's locale expression e resolves a variable reference to lv of agenda item a_s, a *locale-value dependency* is added to the *dependency* set for lv in node a_s. This dynamically constructs a network of locale value dependencies among the agenda items in the design history. Such a network is shown in Figure 5.4. Note that an *incomplete* agenda item has an empty *symboltable*. Because it would clutter design history diagrams, we do not show locale dependencies in them.

Shared agenda items

A design history can actually take the form of a directed acyclic graph, where certain transformations and/or methods can achieve several higher-level effects. Many conventional optimizing transformations such as $?x + 0 \Longrightarrow ?x$ both increase the speed of the ultimate program and also decrease its size. Separate goals requiring space optimization and time optimization can often be satisfied by a single optimizer, as with node G_8 in Figure 5.5.

5.3.3 Design Histories as Basis for Program Explanation

While we do not explore the subject, we believe the design history also provides a considerable amount of infrastructure necessary for *design explanation*. A derivation history is directly analogous to explanation in expert systems via fired-rule traces [WHR78, BS84, Nin89] for individual transformations. The goal structure captured by the design history corresponds more closely to the annotated derivations used in the Explanation of Expert Systems work by [NSM85].

A key difficulty in maintenance is discovering how a program works. Such understanding is a necessary precondition to any successful attempts at modifying the program's function, and many times, when attempting to enhance the performance of a program, to know where optimization can pay off. We claim that the specification plus the design history provides much of the information necessary to describe how the program achieves its purpose, by relating how the specification drove the implementation.

A crucial part of understanding a program is understanding precisely its function, and knowing just how well it was designed to perform. Conventional software implementation environments have a very strong tendency to lose even the informal

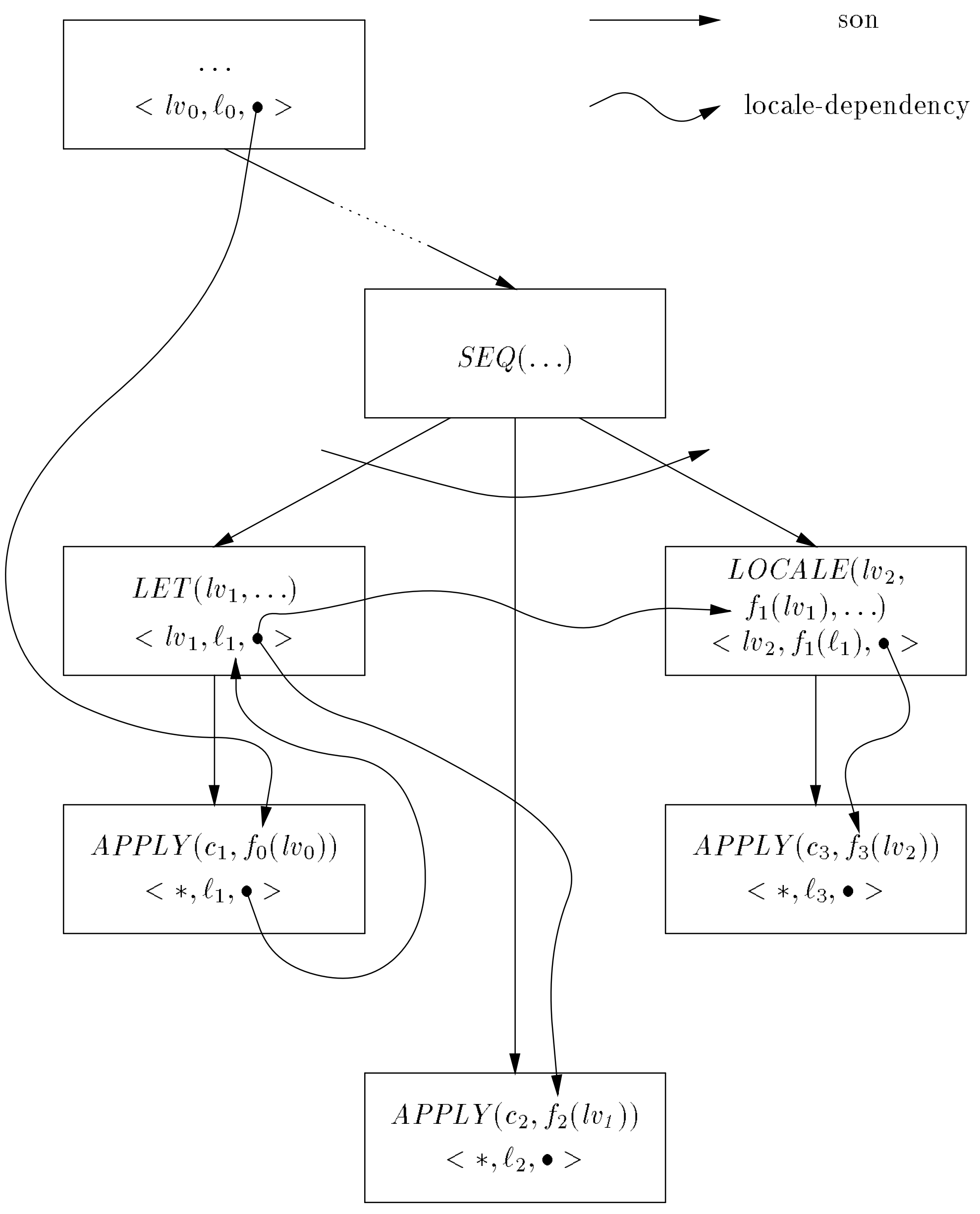


Figure 5.4: Locale-value dependency net

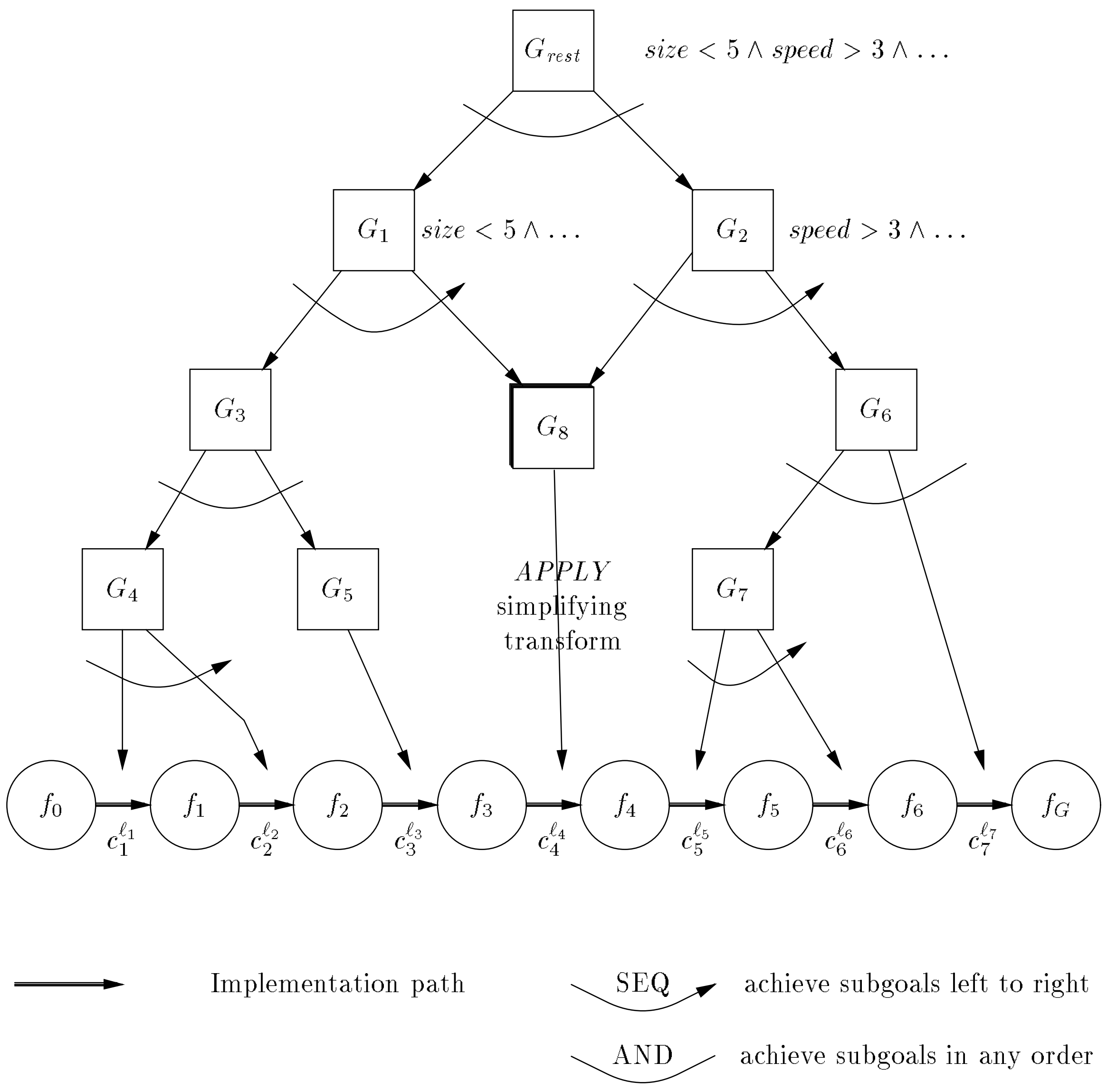


Figure 5.5: Shared agenda item in a design history

description of the software, leaving a would-be maintainer to fall back on out-of-date documentation, the shared knowledge of his co-workers, and the source code itself to divine the purpose of the program. Clearly, retaining both a formal specification and a design history would alleviate the problem of understanding the program's purpose.

Letovsky [LS86] describes a problem in conventional software maintenance he called *delocalized plans*. A fragment of the source code at one place in the text operates in conjunctions with other fragments of source code "far away" in program text. Such fragments are the consequence of a coherent conceptual plan on the part of the original implementor that has scattered by the implementation process. Such components are encountered individually by the maintainer, who assumes that the fragment currently under consideration has a purpose, but that abstract purpose, and the location and the roles of related fragments in a larger context is unknown to him, and must be rediscovered before any modifications are considered. The solution proposed was to require comments near each code fragment implementing a plan part to identify the plan explicitly and to "point" at distant parts of the of the plan implementation.

A captured design history can provide precisely that, in a more formal way. Questions of the form "what purpose does this code fragment serve?" can be answered by tracing back through the generating transformations in the derivation history to the portion of the functional specification which caused that fragment [DKMW89]. Thus this information can serve as explanation of functionality in the same sense that a rule trace can be used for expert systems [NSM85] Similarly, functionality can be traced forward to the code fragments that implement it. Each performance goal is tied explicitly to the plan(s) that achieve it, and each plan to the subgoals or transformations that accomplish the plan steps; this allows traceability from the implemented source back to the performance requirements and vice versa.

Additional information could be recorded to enhance our design history for further explanation. We only capture design choices (Section 1.2.3), intensionally as performance goals, and design selections, in terms of the actual plan to carry out a goal. Information about possible design decisions, their tradeoffs, and the costs of generating and evaluating those decisions are not captured in our framework. A simple possibility is to store the amount of energy expended on making a particular choice, as this can approximate the importance of the decision. Kant [DKMW89] marks forced decisions (those where there is only a single possible choice) specifically to document important design decisions. We also do not capture why a possible decision was eliminated; near misses could be valuable for detecting bugs or guiding the design by suggesting what needs to be changed to convert the near miss into a hit.

Of course, using this information with the intent of modifying the implemented program directly is inappropriate in a transformational context. Chapters 7 and 8 describe a better use for this information.

5.4 Related Work

Design histories have been proposed or used for a number of other systems. These can be grouped roughly as follows:

- metaprograms
- linear historical trace of actions
- tree-structured history
- nonlinear history
- heirarchical plan history
- goal-structured history

A metaprogram is a designated set of procedures (coded in a metaprogramming language) for controlling the transformational implementation of a particular specification. This really constitutes generative information rather than a design history, since it is really the potential of generating a design history. However, a metaprogram loses useful information that was is difficult to obtain: precisely how goals were decomposed, and precisely where transformations were actually applied. We include systems that record metaprograms in this section because they are often used for design replay purposes. Our notion of a Design Maintenance System records metaprograms in a library of TCL methods.

A linear history captures the actions of the development system in the time-sequenced order of occurrence. Linear histories have the virtue of being very easy to capture; a simple transaction log suffices. Our derivation history is such a linear history. When recording such a history, one should capture as much as is available from the transformation system; in our case, we capture both the transform and its locater. In some cases, notably in text derivations of algorithms, one sees only the transforms listed; presumably this is because the examples are small enough so the transform can only apply in a single place, and the authors simply leave the locater out. We note that a purely linear history cannot record that a single action accomplishes more than one useful effect, as does our notion of shared agenda item.

A tree-structured history can capture the actions of a system which always decomposes a problem into nearly independent subproblems; constraint propagation "leaks" parameter, but not structural, information from one subproblem to another. The history is organized as a tree of decompositions of the functional specification represented by the root. Tree-structured histories are easy to capture, but makes the unrealistic assumption that implementation process can always implement submodules. Our design history requires a performance specification at the root, and agenda items do allow pure decomposition (via *AND* and *ACHEIVE*), but also allow sequencing (via *PLAN*).

A nonlinear history contains only the essential sequencing constraints, usually represented by a directed graph (thus the term nonlinear); nodes in the graph represent actions, and arcs represent ordering dependencies. It is possible to construct a nonlinear history from a linear history by performing a dependency analysis, but it is simpler to insist the the mechanism generating the history supply the ordering information directly. The agenda items in our design history capture the nonlinear sequencing directly from the generating TCL *PLAN*; the convenience of such capture was the motivation for installing this construct in TCL in the first place. We were prevented by time from fully treating a nonlinear representation for the derivation history, although we think this is a useful approach.

A hierarchical history captures the breakdown of the development process in the form of hierarchical (procedural) plans. One can have hierarchical histories with either linear or nonlinear subplans. A purely hierarchical history suffers from the same defect as a purely procedural program: there is no explanation of intent. Another view is that the plan is purely operational in meaning. Our design history captures a hierarchical history via agenda items with multiple sons in a nonlinear subplan.

A goal-structured history provides motivation for a hierarchical history; in particular, it somehow provides linkage between plans used and purposes to be achieved. As stated earlier, such goal information is critical to explaining *why* an action or plan is used. Our design history provides goal structure by recording agenda item complexes for goal achievement, containing the dynamically generated subplans of $SEQ(CALL(k,\sigma), ACHEIVE(G_x, e'))$.

5.4.1 Metaprograms as design histories

PADDLE: "Program Developments"

PADDLE [Wil83] Section 4.3.3 is a procedural metaprogramming language. If the metaprogram is specific enough, it has only possible execution path for the particular program, and therefore can act as a peculiar kind of history. However, PADDLE is also apparently intended that general transforming methods be coded in PADDLE, to be reused for implementing a wide range of specifications. In this case, such methods are not likely to uniquely apply, and the ability to indirectly represent a history is weakened. We have chosen to capture both generative knowledge (in the form of TCL methods) and an artifact-specific design history to avoid this schizophrenia.

PROSPECTRA

The PROSPECTRA transformation system system [KB88, KB89b] uses transforms as functions over abstract syntax trees and higher-order functionals (Section 4.3.3) as transformational control. A "development" is defined as the composition of all the transformations/functionals used; the unevaluated composition forms a tree-like structure which we can consider to be a metaprogram. In the absence of special transformations to combine states, such developments effectively force a linear sequence on the transformations. Given the ability to reason about transforms and functionals, one can perform various operations on a development, such as optimizing out unnecessary steps, etc; the point is that tacticals and developments in this scheme can be modified using the same mechanisms as apply to the original program specification. Given such reasoning mechanisms, it is possible to discover that certain sequencing is nonessential, although this is likely to be very painful in the face of complex functionals. No performance goals are provided, so developments are unmotivated. It is not clear whether developments have actually been used in PROSPECTRA.

5.4.2 Linear histories

Zap

The fundamental control concept of Zap that of a *CONTEXT* which determines which transformations are carried out, and guiding transformation by *pattern-directed transformation*. The operation of *CONTEXT*s were described in Section 4.3.3, and can be summarized as nonprocedurally determining a sequence of transformations to achieve a state in which selected equations have a form specified by a goal in the current *CONTEXT*. The individual *CONTEXT*s seem to be very specific to the program being transformed. Sequences of contexts form a metaprogram for generating an entire implementation. Such sequences are established by constructing a script file containing a series of *CONTEXT* descriptions. The individual *CONTEXT* can be considered analogous to TCL methods, and the script file considered a high-level derivation history. Zap histories thus provide low-level goals, but no goals for the higher purpose.

Closely related, but more formalized than Zap's notion of design history, is that of [Imp86]. This defines a development history as a sequence of nodes $\langle G, R \rangle$ where G is a goal, R is representation, and a development step consists of manipulating R to achieve G. This captures Zap's approach of nonprocedurally specifying what is done at each step.

Goldberg

Goldberg [Gol89] records a linear version of a hierarchical history:

"We define a *derivation history* as a trace of the tactics invoked, either manually or as part of the execution of some other tactic, together with the values of the actual parameters passed to them."

Goldberg's "primitive" tactics (4.3.3) correspond to our transforms, whereas nonprimitive tactics are a procedural version of our methods. Because arguments to the primitive tactics are retained, Goldberg's system effectively captures each transformation c_i^ℓ including the locater. However, the relation of a high-level tactic to the lower-level tactics that it invokes and follow it in the history is not retained; in practice, Goldberg in fact only seems to capture invocations of the primitive tactics. No goal information is maintained.

5.4.3 Tree-structured histories

BOGART

The BOGART system [MB87] captured the history of the top-down refinement of a VLSI circuit functional specification. Such specifications define abstract circuit components and information flows between them, much like data flow bubbles. Abstract components are recursively refined into subassemblies of components until only primitive components remain; the refinements depend on constraints (signal timing, etc.) from sibling components. The design history forms a tree isomorphic to the component refinement structure; one can think of a refinement tree attached to each component in the original specification. The advantage of this structure is that questions about how one component is refined are decoupled from any other component which is not a parent or descendent. A severe disadvantage is that such histories cannot represent commonly found optimizations that are possible because of juxtaposition in context[4], and therefore the use of such a history is likely to prevent the construction of even a near-optimal artifact. The authors also acknowledge the absence of performance goals in this type of history.

SINAPSE

The design histories we have discussed so far provide the design history of the components making up a single artifact. The SINAPSE system [DKMW89], used to synthesize finite-difference programs, keeps a tree-structured history to record design selections (Section 1.2.3) where branches in the tree represent branches in the space of possible implementations. In essence, this is a tree "slice" of the design space shown in Figure 3.7, and is an implementation of almost exactly Parnas' notion of program families [Par76]. The tree root is the most abstract functional specification. Each tree node represents the decision to refine some component (thus capturing a locater); each tree arc represents the choice of a particular refinement. This particular model also has trouble recording juxtaposition optimizations, but that is because tree nodes are defined to be decisions to refine a component, rather than the decision to apply some transformation. The actual history is recorded as a set of pairs representing explicitly named design choices and explicitly named design selections for critical design choices. Design choices for which a selection is forced, or for which the built-in control knowledge chooses a satisfactory default result are simply not recorded. This corresponds roughly to choosing an "important" subset of our linear derivation history. Being able to explicitly name critical decisions and selections requires that such decisions and selections are known well in advance of actual transformation; for large scale implementation, we do not think this is generally possible, as as a similar decision may apply in more than one place during a derivation, and a unique name will not be able to differentiate between these. We think our directed acyclic graph like structure would been more appropriate for recording this type of history.

5.4.4 Nonlinear Histories

Cheatham's PDS

The Program Development System (PDS) [CHT81, Che84] transformationally constructed software in stages. Each stage either performed a type analysis and propagation or applied a particular set of transformations according to transformation

[4]If component A is connected to component B, A is refined to A' with an inverter on the output to B, and B is refined to B' with an inverter on the input, then a juxtaposition (called peephole for compilers) optimization removes both inverters. This cannot be recorded in a tree-structured history.

application instructions (a kind of metaprogram) to a functional specification from a previous stage. PDS recorded with each entity (stage result, and we presume, transformation sets, application instructions, and type analyzers) the tool name and parameters that generated it, and a version number equal to one plus the maximum of the version number of that entity's parents. All this information taken collectively forms a dependency network [Fik75], or a nonlinear history of the derivation process of the final stage. PDS did not allow stages to be sequenced by a higher level control mechanism; TCL methods and plans allow this.

Nonlinear Planners

Nonlinear planners (NOAH [Sac77], NONLIN [Tat77], TWEAK [Cha87], introduction [CM85], survey [Geo87] capture plans as networks of actions with a partial order on the actions. When such actions are transformations (or invocations of methods), and the networks represent workable plans for implementing a specification, then it becomes useful as a design history. The inspiration for the *PLAN* construct and the organization of the agenda items in our design history was taken from this technology. The notion of a nonlinear plan requires the notion of partial state, which is easily obtained in problem domains where the problem is conveniently described in terms of conjunctive normal form, and most terms are independent of one another, such as the archetypical blocks world. One of the difficulties with partial state representations and nonlinear plans is evaluating the truth of a predicate immediate before a particular action is applied (this is known as the modal truth criterion); one may have to enumerate an potentially exponential number of possible orderings of previous operators to determine possible full preceding states [Cha87]. In an attempt to avoid facing this problem in the short term, and not wishing to prematurely place any fixed structure on the content of a state so that our transformation model would be widely applicable, we chose to leave states as monoliths in our representation of design history.

5.4.5 Hierarchical plan histories

Hierarchical planners (NOAH [Sac77], survey [Geo87], FORBIN [DFM90], in contrast to component decomposition schemes, decompose high-level plans for accomplishing an effect into lower level plans. Essentially the lower level plans are subroutines for achieving the effect of the higher level plans. The high-level plans are simply names of the lower level subroutines, and therefore have only an operational semantics. Recording the plan breakdown produces a hierarchical plan history, but no explanation as to why the plan should should work or its purpose. TCL can generate such histories by nested *PLAN*s or making procedural *CALL*s to methods. In a hierarchical plan history, one cannot have phantom goals simply because the notion of goal is defined. However, our notion of shared agenda item, begin not necessarily non-procedural in nature, is a useful addition to hierarchical plans; this allows at least the representation of shared actions in a plan, generalizing Mostow's [Mos85b] call for shared design goals.

Abstract heirarchical planners (ABSTRIPS [Sac74], survey [Geo87], [Wil88]) first solve a problem in an abstracted problem space, and then instantiate a partial solution in the problem space and attempt to fill in the details. We have not considered carefully histories for such planners because we have not had to to consider how to abstract the performance goals in the design history.

[Mos85b] suggests that a lattice should be used to represent an idealized design history to allow shared design goals; we have chosen to use a directed acyclic graph to represent shared design goals and actions.

5.4.6 Goal-structured histories

NONLIN

The NONLIN [Tat77] planner uses goals to guide its choice of plans. As the planning process proceeds, a heirarchical task network is built up, showing how a high-level goal is achieved by some plan, possibly having subgoals and eventually terminating in primitive actions. The completed task network is a goal-structured history of the planning process. An interesting structure that appears in nonlinear goal-structured networks are phantom goals, nodes representing some desired goal effect which is serendipitiously true, either because of the initial world state or because of some necessarily-preceding action in the plan. One can represent the simultaneous achievement of separate subgoals in a nonlinear network with an action followed eventually by a phantom goal, but this places a false asymmetry into the network and therefore the algorithms that process it. A TCL design history captures the goal/plan relationships.

The SIPE planner [Wil88] uses essentially the same goal-structured history as NONLIN. However, SIPE also handles abstract plans, constraints, and extended attribute language used in goal expressions. The additions add considerable interesting detail to the design history which we do not have room to discuss here.

PRIAR

The PRIAR nonlinear planner [Kam89] starts with a plan structure essentially identical to that produced by NONLIN, and annotates it further with *validations*. Validations exist for every precondition of leaf actions a in the form $\langle C, a_1, E, a\rangle$, meaning "The action of agenda item a_1 changes the world to produce condition C, which provides necessary support for precondition E required for a to act." Each validation $\langle C, a_1, E, a\rangle$ implicitly requires that $a_1 < a$ in the nonlinear plan ordering. The entire set of validations $\langle C_i, a_i, E, a\rangle$ for E of a must have the property $\{C_i\} \vdash E$.

Each node a in the plan structure is decorated with relevant annotations of the following types:

- E-conditions (external effect conditions): validations provided by the subplan under a to parts of the plan other than the subplan at a. These are the used effects of the subplan under a.
- E-preconditions (external preconditions) are the validations required by the subplan under a from the rest of the plan
- P-conditions (persistence conditions) are validations that the subplan under a must be preserved by the subplan; these correspond to protection intervals, and are conditions that must not be disturbed by execution of the subplan under a.

Each node is also annotated with the schema which generated it, and filter conditions (those the planner will not attempt to achieve, but will simply use if present, such as BLOCK(B) in the blocks world).

Annotations on a node record validations used, generated, or preserved by the subplan below that node. This allows simplified reasoning about what the entire subplan requires, accomplishes, or must not change by virtue of simply knowing the subplan. We think this approach has considerable promise for a Design Maintenance System once we move away from monolithic states.

Carbonell's "Derivation" Histories

Carbonell [Car85] suggests that a problem solving trace capture not only the resulting goal-structured plan for a solution, but also alternatives considered and rejected, near solutions and the cause of their failure, and references to knowledge used. All of this information has potential value for explanation and in similar-problem solving situations. Carbonell only sketches of how this knowledge can be used.

We add that costs to generate and evaluate each piece of stored information would also be of great interest; a design selection achieved at great cost should generally be protected. Knowing the costs of certain design selections also allows better estimates to be made in the face of a proposed change. Finally, such costs are also a key to controlling storage costs, by using the strategy of recording only information whose cost to acquire exceeds some threshold. Cheaply acquired information is probably not worth the trouble to store.

5.5 Summary

This chapter has defined the notions of derivation history and design history, discussed how such information can aid program understanding, and compared our notion of a design history with those found in the literature. Such "understanding" can be used by tools to install changes in the artifact described by the design history. In the following chapters, we define how changes are specified and how those changes can be installed into a design history.

Chapter 6
Maintenance Deltas

Chapter summary. This chapter gives a theoretical characterization of transformational maintenance. The notions of maintenance delta and delta integration are defined. Classification and formal representations for each type of delta are provided in terms of transformation system inputs.

Given an existing artifact, and a possible modification, we would like to construct a new artifact having that modification. The process of deriving the modified artifact from the existing one is traditionally termed *maintenance*.

Transformational maintenance is using a transformation system to aid modification of an artifact. We believe there is great value in using a transformational perspective to guide a maintenance process. Such a perspective provides us with a way of classifying types of changes. The change type, in turn, leads to type-specific procedures for integrating the change into an existing artifact. Since revision of the design history is really a prerequisite to constructing the changed artifact, we call such a system a *Design Maintenance System*. Combining such procedures with the conventional transformational implementation paradigm provides one not only with a mechanism for implementing the maintenance process, but also the possibility using the identical mechanism to implement an incremental design process. The integrated process we term *Incremental Evolution*.

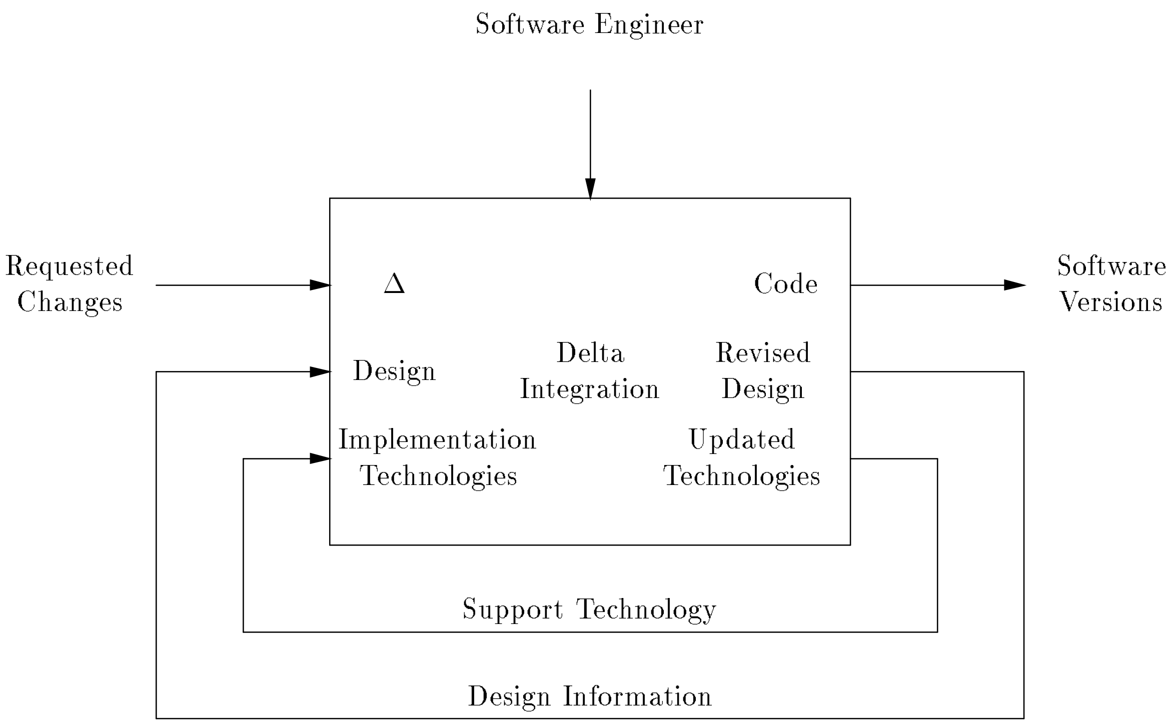


Figure 6.1: Incremental Evolution: A system for managing change

An overview of the transformational maintenance process (Figure 6.1) is described in the following procedure:

Incremental Evolution:

1. Specify an artifact formally
2. Construct a (partial) implementation using a transformation system, capturing the design information (a design history) and the technological support used to construct the implementation
3. Repeatedly
   (a) Determine a desired modification of the (partial) implementation
   (b) Specify a formal change, called a *maintenance delta*, that states the modification
   (c) Integrate the maintenance delta into the existing implementation support technology, the design information for the artifact, and artifact itself

The desired change may have some effect on the implementation technologies (property-preserving transforms and methods) used by the transformation system.

The delta integration process is roughly:

Delta Integration:

1. Determine the *type* of the maintenance delta
2. Revise the support technology if required
3. Analyze the maintenance delta with respect to the design history, using a type-specific process to determine which design history elements must be dropped
4. Regenerate the remainder of the design history by re-running the transformation system

This chapter will motivate and define maintenance deltas. We will assume the preconditions to transformational maintenance: the existence of a specification, a (partial) implementation of an artifact, a captured design history, and some new, but informal requirement desired for the existing artifact.

6.1 Using Deltas to Speed the Artifact Maintenance Process

Assume we have an implementation of a specification G:

$$f_G = \mathit{IMPLEMENT}(G)$$

and somehow managed to formulate a desired change as a formal specification delta, δ. The effect we desire from maintenance, for some specification combining operation $\oplus_{\mathcal{G}}$, is:

$$f_{G \oplus_{\mathcal{G}} \delta} = \mathit{IMPLEMENT}(G \oplus_{\mathcal{G}} \delta)$$

We argued earlier that running *IMPLEMENT* transformationally is expensive; we wish to avoid a computation of similar complexity.

Following an analogy to differential calculus, we would hope to find some incremental computation to allow use to perform this computation cheaply, assuming that the change is small. Such an approach is explored in *formal differentiation* [Pai81] in which complex computations are incrementally adjusted by applying some reduced-strength operation to a base computed value and a delta. Described in terms relevant to our problem, the base computed value is $f_G = \mathit{IMPLEMENT}(G)$. A reduced-strength operation (*INTEGRATE*) is found by considering the effects of the delta-combining operation ($\oplus_{\mathcal{G}}$), combining the delta δ with the original argument G, on the complex computation (*IMPLEMENT*). Using this approach we could ideally construct a function $\mathit{INTEGRATE} : \Delta \times \mathcal{F} \rightarrow \mathcal{F}$ so that:

$$f_{G \oplus_{\mathcal{G}} \delta} = \mathit{INTEGRATE}(\delta, f_G)$$

with $\mathit{cost}(\mathit{IMPLEMENT}) >> \mathit{cost}(\mathit{INTEGRATE})$. We call such a revision operation a *delta integration* procedure, because it knows how to install a delta into an existing implementation.

Given that a transformation system actually has multiple inputs, and that each input to the transformation system affects the final implementation and design history in a different fashion, there must be different delta integration procedures for each input, much as with partial differentials. Given $\mathit{TRANSFORM}(x, y, z)$, if input y changes by a small value Δy, then

$$\mathit{TRANSFORM}(x, y + \Delta y, z) = \mathit{INTEGRATE}_y(\Delta y, \mathit{TRANSFORM}(x, y, z))$$

assuming some analog of continuity of *TRANSFORM* in the region near y. For each different input x, y, or z, we need a different delta integration procedure $\mathit{INTEGRATE}_x$, $\mathit{INTEGRATE}_y$, or $\mathit{INTEGRATE}_z$.

Consequently, we need to identify the types of change that are possible in a transformational context, and must provide different integration procedures for each type of delta.

If we were able to simply integrate a delta directly into an implementation, without the aid of any other information, then we would have an *implementation maintenance system*. This is difficult to do in practice, because one must regenerate explanations of the roles of the parts of the program, just as in conventional maintenance. Rather than regenerating this design information, we insist that we simply not lose it. Now we must also integrate the delta into the design information $\mathcal{D}$ as well as the implementation, so that we are ready to handle a successor delta:

$$INTEGRATE_{type} : \Delta_{type} \times \mathcal{G} \times (\mathcal{F} \times \mathcal{D}) \rightarrow \mathcal{G} \times (\mathcal{F} \times \mathcal{D})$$

Just as constructing a design is most of the work involved in obtaining an implementation, so integration of deltas into a design is most of the work involved in revision. Consequently, we call a system that accomplishes this effect for many kinds of deltas a *Design Maintenance System*.

Each integration procedure can conceptually be considered independently of the others. In a practical Design Maintenance System, a number of delta integration procedures will need to be run to effect a change affecting several aspects. Those procedures should be combined so as to minimize duplication of effort. We will find that most of the procedures consist of identifying reusable portions of the design information, stripping away the reusless portion, followed by design repair (replacing the missing design information). The design repair can be delayed until all the integration procedures have had their chance to strip away reuseless design information. Understanding this proviso, we will show the integration procedures separately.

6.2 Classification of change: Types of Deltas

Traditional maintenance classifies change types into *perfective*, *adaptive* and *corrective* [LS80, Wed85]. *Perfective* changes are those that improve a software system somehow without affecting its existing capabilities, i.e., decreasing resource utilization costs, etc. *Adaptive* changes are those that allow the software to operate in a newly changed context. *Corrective* changes include bug fixes.

These classifications of change are unfortunately not only informal, but they only label the work or the end product; as classifications, they provide very little help in actually accomplishing the desired change.

A viewpoint based on a model of a formal transformation system classifies changes in terms of entities involved in the transformation system, and can do so in a formal fashion[1]. Our approach to discovering change classifications is to inspect all the inputs, outputs and structures of the transformation system and to propose a change type for each. The value in this approach is that for each delta type, there is some hope of producing a transformation system specific procedure for handling that type of delta by inspecting the transformation system itself, rather like a generalization of the notion of *finite differencing* [Pai81, Pai86].

An earlier work [ABFP86] classified transformational changes into *performance change*, *environmental change* (as a subset of performance change), *functional change*, and *design error correction*, and provided informal methods for managing change on each of these categories. This work follows in the same vein, but classifies the the change types more carefully, and provides concrete procedures for managing change.

Obviously, the more detailed the transformation system model, the more delta types we can propose. From the point of view of what programs can be produced by the transformation system, proposing a delta type for each possible input is sufficient to cover all possible types of deltas. Allowing changes to the other aspects of the transformation system can at best provide additional convenience, but not greater theoretical power. Similarly, for any particular input, one can further classify the input values, leading to even more detailed delta characterizations; an example of this can be found in Section 9.4.7. We have chosen a set that we think constitute the major classifications, recognizing that further work may identify interesting subclassifications.

[1]Balzer [Bal85b] classifies structural changes to domain models into one of 15 types. While the changes are formal, they are only to one aspect of software construction, and so his classification is much too limited for our purposes.

We observe that a transformation system (Figure 4.1) has the following inputs:

- G, the entire performance specification, usually composed of the parts:
  - f_0, the initial program satisfying $G_{implicit}$
  - G_{rest}, composed of
    - pure predicate specifications g_i
    - performance bound specifications via values $v_{i,j}$
- P, the set of available performance value observation functions p_i, indirectly defining $\mathcal{V}_i$, the set of performance values
- Definitions of the various performance predicates G_j (usually in terms of some $\succeq_i$)
- $C = \{C_j\}$, the approximate set of G_j-preserving transforms, especially C_{base} for those transformation systems with fixed p_{base}
- $M = \{m_k\}$, the available methods for navigating the design space
- The software engineer

A change to any of these inputs gives rise to new potential implementations. Consequently we define a delta type for each input as shown in Figure 6.2. Any object representing such a change we call a *maintenance delta.*

It is tempting to collectively call these changes "specification deltas", but we do not, because not all of the deltas apply to the specification; some apply to the implementation knowledge that the transformation system possesses that is independent of the specification. Furthermore, not all the changes to the specification G are actually made directly to it; some are made to the base specification represented by the program f_0, and so affect the total specification G only indirectly. We use the term *specification delta* to refer to any of the specification changing operations Δ_G, Δ_v, Δ_f. We use the term *support delta* to refer to any of operations affecting the supporting databases used by the transformation system: $\Delta_{\mathcal{C}}$, $\Delta_{\mathcal{G}}$, $\Delta_{\mathcal{P}}$, $\Delta_{\succeq}$, $\Delta_{\mathcal{V}}$, $\Delta_{\mathcal{M}}$, $\Delta_{\mathcal{E}}$.

In the balance of this section, we describe each type of delta. We give explicit definitions for each type of delta in Section 6.4.

Since we do not know how to represent changes to software engineers, $\Delta_{\mathcal{E}}$, we ignore them in this thesis. We expect this problem to remain unsolved for a very long time.

Performance deltas (Δ_G) to G are the essential specification changes. Such changes affect the performance aspects of the desired program. They are generated

- Specification deltas:

  Δ_G : Change of performance (modification of specification G)

  Δ_v : Change of performance bound (modifications to performance bounds v_i)

  Δ_f : Change of functionality (modifications of f_0)

- Support deltas:

  $\Delta_{\mathcal{C}}$: Change of technology (modification of C_i)

  $\Delta_{\mathcal{G}}$: 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

  $\Delta_{\mathcal{E}}$: Change of software engineer

Figure 6.2: Types of delta induced by structure of transformation model

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 $G_{FORTRAN}$ to G_{PROLOG}.

Performance bound deltas (Δ_v) are a special kind of performance deltas. 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 $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. 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.

So-called functional deltas (Δ_f) occur because of the practice of providing mixed specifications containing a base specification f_0 to the transformation system instead of G_{entire}. Such changes are generated whenever the expected performance $p_{base}(f_0)$ of the initial "specification" f_0 does not meet the requirements. The term "functional" delta comes from the common practice of defining $G_{invariant}$ in terms of $p_{meaning}$, but is not limited to this case. An example functional change would be modifying the functional program $f_0 \equiv sin^2(x-3)$ to be $sin(x+1)$. The evolution transforms of Johnson [JF90, Fea89a] 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 non-property-preserving transforms (Δ_f's) to convert the initial f_0 into functional specifications believed to better suited. In practical systems, we would expect the bulk of changes made to be functional changes; performance tuning usually comes after achieved functionality.

Technology deltas ($\Delta_{\mathcal{C}}$) occur when software engineers realize that a desired transform does not exist in the library of property-preserving 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 of engineering a complete transform library in advance of use of the system [Bal85a, CHT81]. 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. Errors in transforms can even be introduced at implementation time, if one allows incremental domain engineering as in the CIP system [BEH$^+$87, BMPP89]; a designer can define and use transforms whose validity he will verify later, and a transform must be retracted when its validation later fails to go through. Even correct transforms can be invalidated if the problem domain to which they apply changes, as is expected in the domain engineering process [Ara88]. Rarely-used property-preserving transforms might be retracted if the transform library gets too large to manage conveniently; this is a tradeoff between power of transforms and the branching factor of the design space. We consequently expect that technology changes will be necessary both during program construction and during maintenance. However, we expect that the rate at which technology changes are generated will drop as the transform library matures and becomes validated; all users of the transformation system will benefit from such changes.

Method deltas ($\Delta_{\mathcal{M}}$) capture knowledge of new implementation techniques, or fix errors in existing techniques. An error in the action $a_{mergesort}$ (Section 4.2.5), for example, a complexity goal of $\mathcal{O}(n^2)$ performance in locale lv_4, would require a method delta to correct it. Such changes take place for the same kind of reasons that technology changes occur. Some methods will be applicable over a broad range of programs, but, unlike technology changes, our expectation is that for each program implemented, some new methods will be generated, mostly due to our inability to encode effective heuristics for every possible program [Bal85a], and because of limits on the completeness of the control mechanisms. Such program-specific methods we do not expect to augment the general utility of the transformation system, and so we would expect them to be stored with each individual design history [2].

[2]This implies that the transformation system has two sets of method inputs, one for generically useful methods, and the other for methods specific to the problem at hand. We ignore this distinction in this thesis.

Performance predicate library deltas ($\Delta_{\mathcal{G}}$) 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. Such deltas would occur when an incorrectly implemented predicate defining $G_{fits-in-one-page}$ was fixed.

Deltas relating to performance measurement functions $\Delta_{\mathcal{P}}$, changing a subsumption ordering over a set of performance values $\Delta_{\succeq}$, or to the range of performance values $\Delta_{\mathcal{V}}$ are possible but expected to be rare; such changes indicate an error on the part of the domain analyst defining these entities, or incorrect implementation of these in the transformation system proper. We do not address them further in this thesis, but we think that the techniques outlined for the other deltas can be adapted to handle them.

There is yet another class of changes which are beyond the scope of this thesis: changes to *process* performance predicates, or constraints over resources consumed by the transformation system while constructing an artifact. First, our transformation model does not account for process costs or predicates; augmenting it to do so would be a necessary first step. Secondly, there is a conceptual problem with process predicates with respect to maintenance: given that an existing implementation has achieved some process predicate, what does it mean to change that process predicate? The resources have already been consumed. What we currently expect is that a new process predicate will be supplied for each installed change.

6.3 What is a meaningful unit of change?

A unit of change is one for which the changed entity is well-defined, and for which it is worth investing significant energy to install.

We distinguish between *micro-changes* and true change. A micro-change modifies a specification, leaving it in a possibly ill-formed state. A true change to a specification must leave the specification well-formed, and must achieve some useful goal desired by the designer. Micro-changes occur as a consequence of using tools that manipulate the representation for a specification without regard to whether the manipulation leaves the specification in a consistent state with respect to the semantics of the specification, or a useful state from the point of the designer. No effort should be expended attempting to handle a change until a true change has been made; the micro-changes installed by the tools must be composed[3] into a unit change.

[3]This is similar to to the problem of composing a property-preserving transform from a bundle of non-property-preserving transforms. The difference here is that the composition of the deltas need

Consider using a text editor to modify the textual representation of a PASCAL program. Adding an additional test to a subroutine will likely require several editor commands to accomplish the desired effect; the editor commands would effect the micro changes. It is important that any system for installing changes do so only when a complete set of such micro-changes has been made; otherwise, considerable effort may be wasted. There is no point in compiling the program until the test statement is completely coded and the declarations which support it are adjusted properly.

An example from [GKS86] shows that several unit changes made to a specification may work together to accomplish a desired effect. Consider a sequence of changes to a particular BNF production rule:

1. X ::= Y Z

2. X ::= Z

3. X ::= Z Y

In **1.**, we see the original specification for a particular rule. In **2.**, the designer has deleted the first component of the right hand side (this is a valid change from the point of consistency of the "spec"). In **3.**, the designer inserts a new second component. The desired result took *two* steps for the designer to state, neither of which was a micro-change. Clearly, installing change immediately after the first step is complete is inappropriate.

Secondly, there is potential ambiguity introduced: is the Y introduced at step **3.** the same Y deleted in step **1.**? Answering YES or NO leads to two different specified changes. We see that the individual steps must compose unambiguously or the user must specify which is intended when more than one composition is possible.

We avoid this problem by requiring specification of the entire change desired as a single entity. Any practical system performing incremental evolution must handle the composition of the micro-changes made to obtain the change specification we require for our approach. It is likely that the interface to the software engineer specifying a change will need to be made on a database-like atomic transaction basis; this is probably necessary anyway in any environment where a number of software engineers can simultaneously be working on a project. Johnson's system for applying evolution transforms [JF90] provides this effect by offering the specifier a menu of specification-changing operations which leave the specification well-formed; collection of a unit change is handled by requiring the specifier to explicitly invoke the transformation system on the specification, which effectively signals the end of a transaction.

not be a property-preserving transform in any sense of the word; simply that the composition be interesting.

In practice, a unit change will be a number of deltas of each type assembled to form a composite δ to be applied:

$$\delta = \{\delta_1, \delta_2, \ldots, \delta_k\}$$

There are some obvious consistency requirements for a composite delta. The foremost is simply that no δ_i and δ_j in the composite delta should conflict; it is not meaningful for a composite delta to simultaneously delete a transform from the transform library, and also add a method that applies that transform. We note that independent delta types do not necessarily imply independence of deltas; a technology delta $\Delta_{\mathcal{C}}$ may require method deltas $\Delta_{\mathcal{M}}$ for those methods using the affected transforms. Producing a detailed model of delta consistency is beyond the scope of this thesis.

The composite δ is given to a composite *INTEGRATE* procedure to be integrated into the design, the implementation, and the technology support, as shown in Figure 6.3. The composite *INTEGRATE* procedure must decompose the composite δ into its constituent parts, shunt each part δ_{type} to an appropriate $INTEGRATE_{type}$, and combine all the results. A more detailed overview of the process of breaking up the composite delta and shunting it to appropriate procedures was shown in Chapter 1.

6.4 Form of Deltas

Delta types only allow us to classify. To process deltas, we need concrete definitions of their form. We give domain definitions for each type of delta as a set of values, as well as the effect of "applying" individual deltas. Such definitions provides us with a means for representing changes as formal entities, and allowing tools to inspect the deltas for interactions with the existing specifications and artifacts.

In general, since changes to an object can be captured as a function from objects to objects, an instance of each delta type is a parameter to a revision function appropriate to that kind of delta:

$$REVISE_{type} : \Delta_{type} \times object \rightarrow object$$

Given a particular delta instance δ of type $\Delta_{type(\delta)}$ and object instance ω, we define

$$\delta(\omega) \equiv REVISE_{type(\delta)}(\delta, \omega)$$

as a convenient notation.

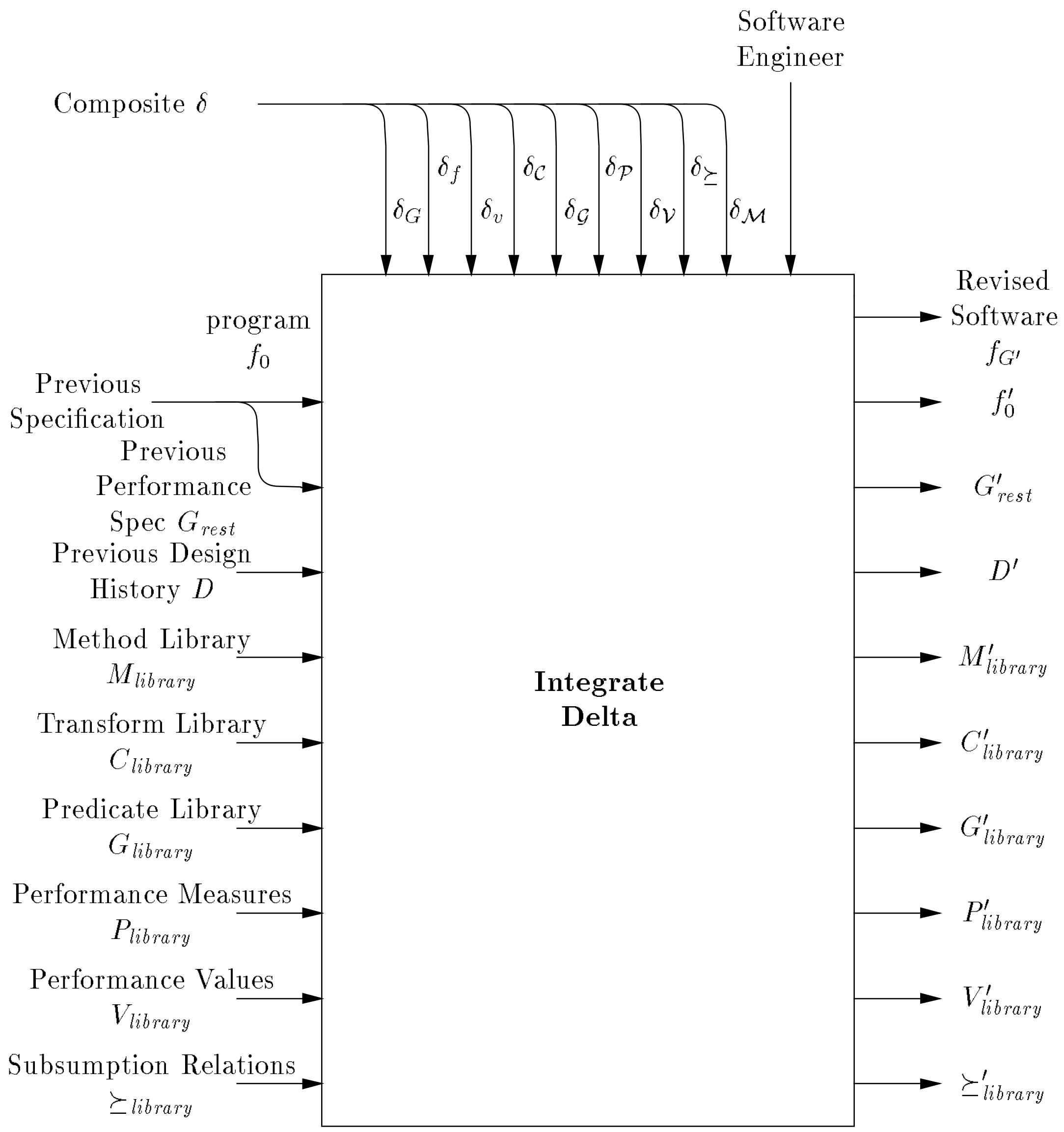


Figure 6.3: Model of transformational change

Our current approach can be characterized as defining a fixed set of specialized definitions for certain interesting subtypes of each change type. The subtypes are, like the types themselves, determined by specialized techniques for handling the subclass. An unexplored possibility is to consider using transformations to represent changes to all input entities of the transformation system, as we have with method bodies in the following.

6.4.1 Performance Deltas Δ_G

For performance deltas, any representation for Δ_G requires that we place some structure on G_{entire}. Fortunately, a natural structure suggests itself due to the usually-conjunctive nature of $G_{entire} = G_1 \wedge G_2 \wedge \cdots G_n$: represent a conjunctive predicate specification as a set of individual predicates G_i. Δ_G then becomes a means of mapping sets (of predicates) to sets (of predicates). In practice, we expect that specific predicates will be added, deleted, or replaced; replacement can be handled by deletion followed by addition. We define:

$$\Delta_G = powerset(\mathcal{G}) \times powerset(\mathcal{G})$$

For each $\delta_G = \langle G_\ominus, G_\oplus \rangle \in \Delta_G$

$$\delta_G(G) = (G - G_\ominus) \cup G_\oplus$$

Here, $-$ and $\cup$ stand for set difference and set union respectively.

6.4.2 Performance Bound Deltas Δ_v

Performance bound deltas, being a special kind of performance delta, have a more specialized form. We do not need to handle deleted performance bound goals; those can be handled by Δ_G. We only need to worry about added or revised performance bound deltas. We need to capture which performance goal i is being changed, and the replacement value $v_{i,j}$:

$$\Delta_v = powerset(V_{current}) \times powerset(V_{current})$$

where $V_{current} = \{\, v_{i,j} \mid p_i \in P \wedge v_{i,j} \in \mathcal{V}_i \,\}$. For each $\delta_v = \langle V_\oplus, V_\Delta \rangle \in \Delta_v$:

$$\delta_v(G) \equiv (G - G_\ominus) \cup G_\oplus \cup G_\Delta$$

with $G_\ominus$ being performance goals to delete, $G_\oplus$ being new performance goals, and G_Δ being revised performance goals:

$$\begin{aligned} G_\ominus &\equiv \textstyle\bigcup_{v_{i,j} \in V_\Delta} \{\, p_i(f) \succeq v_{i,x} \mid p_i(f) \succeq v_{i,x} \in G \,\} \\ G_\oplus &\equiv \textstyle\bigcup_{v_{i,j} \in V_\oplus} \{ p_i(f) \succeq v_{i,j} \} \\ G_\Delta &\equiv \textstyle\bigcup_{v_{i,j} \in V_\Delta} \{ p_i(f) \succeq v_{i,j} \} \end{aligned}$$

Performance bound deltas that revise provide more information than performance deltas. This information is the relation between the revised performance bound and the old performance bound, which is one of the following:

- $v_{i,revised} \succeq v_{i,original}$
- $v_{i,original} \succeq_i v_{i,revised}$
- $v_{i,revised} \succeq v_{i,original} \wedge v_{i,original} \succeq_i v_{i,revised}$
- $v_{i,revised} \not\succeq v_{i,original} \wedge v_{i,original} \not\succeq_i v_{i,revised}$

This additional information can make the integration procedure for performance bound deltas potentially more efficient than that for performance deltas.

6.4.3 Functional Deltas Δ_f

Functional deltas Δ_f are simply maps from a specified f_0 to a revised f_0'; these turn out to be precisely our definition of transformation, including a locater value. We thus assume that whatever form the transformation system uses for transformations will be used for Δ_f's.

$$\Delta_f = \{\, t^\ell \mid t \in \mathcal{T}, \ell \in \mathcal{L} \,\}$$

For each $\delta_f \in \Delta_f$:

$$\delta_f(f) \equiv t^\ell(f)$$

Such deltas may be either property-preserving transforms or non-property-preserving transforms with respect to $G_{invariant}$, although the interesting ones are non-property-preserving transforms. We allow property-preserving transforms for δ_f's only for generality. In normal practice, one does not expect to see specification changes which have no effect; a property-preserving transform δ_f is probably an error if produced by a system analyst[4].

Because of limitations on what transforms are representable in a particular system (simple tree-transforms don't handle global changes well) it is possible that the form used for deltas may not be able to express a desired change concisely. This is merely a shortcoming of the chosen representation for programs and transforms, not our methods. We shall assume that a larger-grain transform that includes a desired precise change is always possible to construct; in extreme cases of representational weakness, we can always fall back on total state transforms[5] like $f_0 \Longrightarrow f_0'$.

6.4.4 Method Deltas $\Delta_{\mathcal{M}}$

Method deltas $\Delta_{\mathcal{M}}$ can affect an existing library of methods M in a number of ways:

- add new methods
- delete existing methods
- revise existing method postconditions
- revise existing method procedure body

More detailed characterizations of method changes are possible due to their rich internal structure (cf. discussion on TCL), such as changing parameter lists, etc., but we shall model such changes using the above list[6], as the utility of finer grain forms is currently unclear.

[4]Johnson [JF90, p. 241] seems to think differently; his "evolution transforms" include "reorganizing" transforms, whose purpose is simply to shuffle the functional specification around, and "data-flow modifying" transforms which apparently insert buffers between agents. To us, these appear to be early implementation decisions. We can see some utility for functional deltas produced by a software engineer.

[5]An interesting alternative representation is to allow Δ_f to be *sets* of transformations; then assuming that the transforms can collectively effect any set of local changes, any global change can be represented.

[6]Parameter list modification can be modeled by method replacement. Revising a method is a special case of revising its postcondition and revising its procedure body.

We define:

$$\Delta_{\mathcal{M}} = powerset(\mathcal{M}) \times powerset(\mathcal{I}) \times powerset(\mathcal{I} \times \Delta_G) \times powerset(\mathcal{I} \times \Delta_f)$$

with $\mathcal{I}$ being the set of possible identifiers. Each

$$\delta_{\mathcal{M}} = \langle M_{\oplus}, \Delta_{\mathcal{M}\ominus}, \Delta_{\mathcal{M}postcondition}, \Delta_{\mathcal{M}action} \rangle \in \Delta_{\mathcal{M}}$$

has the the following parts:

- $M_{\oplus} = \{\langle i, a, G \rangle\}$ is a set of methods to be added.
- $\Delta_{\mathcal{M}\ominus} = \{i\}$ is a set of identifiers of methods to be deleted.
- $\Delta_{\mathcal{M}postcondition} = \{\langle i, \delta_G \rangle\}$ is a set of method postconditions to be revised.
- $\Delta_{\mathcal{M}action} = \{\langle i, \delta_f \rangle\}$ is a set of method actions to be revised.

Since a postcondition is a performance predicate, we represent a change to a particular postcondition as a performance delta δ_G or its specialization δ_v; each such performance delta must be associated with a method identifier to indicate which method postcondition is to be changed. $\Delta_{\mathcal{M}postcondition}$ is then a set of pairs $\langle i, \delta_G \rangle$ of method identifiers and performance deltas.

Method bodies can be treated as a kind of program, so a change to the procedure content of a method can be captured by a transformation[7] δ_f. Similarly, each such delta must be paired with an identifier indicating to which method in the library that it applies. $\Delta_{\mathcal{M}action}$ is a set of pairs $\langle i, \delta_f \rangle$ of method identifiers and functionality deltas. For this thesis, we shall ignore the possibility that method bodies as programs require different representations than the objects the transformation system is intended to manipulate, with the consequent problem that action transforms might require different representations than are normally used by the transformation system.

[7]This does not mean that other kinds of deltas are necessarily transformations.

We define application of method deltas:

$$REVISE_{\mathcal{M}} : \Delta_{\mathcal{M}} \times powerset(\mathcal{M}) \rightarrow powerset(\mathcal{M})$$

as

$$\delta_{\mathcal{M}}(M) = (M - M_{\ominus}) \cup M_{\oplus} \cup M_{\oplus postcondition} \cup M_{\oplus action} \cup M_{\oplus both}$$

with

$$\begin{array}{lcl} M_{\ominus} & \equiv & \{\, \langle i, a, G\rangle \mid i \in \Delta_{\mathcal{M}\ominus} \,\} \cup \\ & & \{\, \langle i, a, G\rangle \mid \langle i, \delta_f\rangle \in \Delta_{\mathcal{M}action} \,\} \cup \\ & & \{\, \langle i, a, G\rangle \mid \langle i, \delta_G\rangle \in \Delta_{\mathcal{M}postcondition} \,\} \\ \\ M_{\oplus postcondition} & \equiv & \{\, \langle i, a, \delta_G(G)\rangle \mid \langle i, \delta_G\rangle \in \Delta_{\mathcal{M}postcondition} \wedge \\ & & \qquad \langle i, \delta_f\rangle \notin \Delta_{\mathcal{M}action} \wedge \\ & & \qquad \langle i, a, G\rangle \in M \,\} \\ \\ M_{\oplus action} & \equiv & \{\, \langle i, \delta_f(a), G\rangle \mid \langle i, \delta_f\rangle \in \Delta_{\mathcal{M}action} \wedge \\ & & \qquad \langle i, \delta_G\rangle \notin \Delta_{\mathcal{M}postcondition} \wedge \\ & & \qquad \langle i, a, G\rangle \in M \,\} \\ \\ M_{\oplus both} & \equiv & \{\, \langle i, \delta_f(a), \delta_G(G)\rangle \mid \langle i, \delta_G\rangle \in \Delta_{\mathcal{M}postcondition} \wedge \\ & & \qquad \langle i, \delta_f\rangle \in \Delta_{\mathcal{M}action} \wedge \\ & & \qquad \langle i, a, G\rangle \in M \,\} \end{array}$$

While this looks formidable, all it really says is that the set of methods is updated by deleting unwanted methods, adding new methods, and revising methods that need to be changed.

We limit changes to method bodies to transforms rather than allowing application of methods, to allow us some hope of eventually analyzing the effect of the changes.

The richness of the delta for methods stems from the need to save work in the maintenance process; we can use the additional detail to avoid re-executing parts of the method later.

6.4.5 Technology Deltas $\Delta_{\mathcal{C}}$

Technology deltas $\Delta_{\mathcal{C}}$ are changes to the sets of available property-preserving transforms:

$$\Delta_{\mathcal{C}} = powerset(\Psi \times \mathcal{T}) \times powerset(\Psi \times \mathcal{T})$$

where $\Psi = \{\, p \mid p \in P_{library} \,\} \cup \{\, g \mid g \in G_{library} \,\}$ is a set of property names for the sets of property-preserving transforms.

Remembering that transforms actually available to the transformation system are packaged as sets of approximations of property-preserving sets of transforms, i.e.,

$$C = \{C_1, C_2, \ldots C_n\}$$

we define, for each $\delta_{\mathcal{C}} = \langle \Delta_\ominus, \Delta_\oplus \rangle \in \Delta_{\mathcal{C}}$:

$$\delta_{\mathcal{C}}(C) = \left\{ C_i' \mid C_i \in C, C_i' = C_i - \{ t \mid \langle i, t \rangle \in \Delta_\ominus \} \cup \{ t \mid \langle i, t \rangle \in \Delta_\oplus \} \right\}$$

Notice that several sets of property-preserving transforms may be updated at once.

As a consistency requirement, technology deltas are assumed to be presented in advance, or coupled with, method deltas that change the set of transforms used (*APPLY*'d) by a method.

6.4.6 Form of other Deltas

Performance library deltas $\Delta_{\mathcal{G}}$, performance measurement deltas $\Delta_{\mathcal{P}}$, subsumption-ordering deltas $\Delta_{\succeq}$ and performance range deltas $\Delta_{\mathcal{V}}$ are all similar in structure: a list of identifiers for those which are being deleted, and $\langle$*identifier*, *value*$\rangle$ pairs for those being revised. The *value* portion of $\Delta_{\mathcal{G}}$ and $\Delta_{\mathcal{P}}$ consist of functions that can be applied to states to extract qualities (booleans and performance values, respectively). For $\Delta_{\succeq}$, *value* is the replacement boolean function comparing two values.

6.5 Acquiring Deltas

We do not intend to solve the problem of acquiring particular deltas for a given program; for this work, simple possession of a desired set of deltas is sufficient. However, we outline some methods for obtaining the desired changes for the sake of completeness.

One general requirement is shared by all of the delta collectors: the ability to inspect the aspect of the transformation system affected by the delta type. In the case of functionality deltas, inspection of the supplied value f_0 by conventional pretty-printing techniques is well understood. For libraries of methods and transforms, some means for selecting and displaying the objects of interest needs to be provided.

Acquiring technology changes $\Delta_{\mathcal{C}}$ are relatively straightforward; a tool for defining new transforms to add to the transformation library, as well as designating the set of disallowed transformations is required. The essential parts of such a tool must have been present when the transformation system was constructed. The CIP system [BEH+87] uses the transformation system itself to construct new transforms, to ensure that the constructed transforms are property-preserving transforms.

Since specifiers may wish to make arbitrary modifications Δ_f to functional specifications, some means of directly entering non-property-preserving transforms is needed; portions of the same mechanism which allows definition of new transforms can likely be pressed into service for this. Changes to functionality could also be captured by use of a *program editor*, a special tool to allow a designer to edit the representation of a specified program f_0. Structure editors for programs could provide a convenient basis [Rep84]. Upon completion of an editing session, the editor would compose the individual edits to obtain a specific δ_f. The ARIES system [JF90] provides a different approach: a designer selects "evolution transforms" from a set of those found to be generically useful in the past, and selects bindings by pointing with a mouse at a graphic display of the program. The selection process occurs either by pointing at a menu item, or by specifying some desired effect on the program, such as "promoting a type declaration" to encompass a larger type using a predefined type lattice. Even for ARIES, it seems clear that a way of defining deltas not present in the set must also exist. A poor third approach, standard in conventional software engineering environments, is to allow arbitrary text editing of a linear text representation of a functional specification, and to generate a functionality delta[8] by comparing the resulting f_0' with the original f_0.

An interesting possibility is the generation of functional deltas that enable method application at some later stage of the transformation process. The idea is that at some point during transformational implementation, a particular method achieving some interesting performance result (via some available set of transforms) may not quite apply. The failing part of the method postcondition may be satisfiable if the initial specification is changed appropriately. This obviously will generate maintenance deltas. This is reminiscent of *goal regression* [Wal77], for which tools are necessary. We shall say a little more in Chapter 7.

A difficult open problem is that of generating deltas at a abstraction level consistent with the specifications given to the transformation system. Observation of failures at the level of the running program does not necessarily translate easily into the abstractions the specify the program.

[8]The transformation replay scheme used in [Gol89], and conventional software development paradigms only allow such edits; no delta is ever generated.

6.6 Summary

This chapter has provided:

- A means for defining a complete set of *maintenance deltas* based on the possible inputs of a transformation system
- Motivated the real utility of such maintenance deltas in terms of delta-type specific integration procedures for integrating the delta into an existing implementation
- A specific list of maintenance deltas defined by our model of transformational implementation
- Defined forms for each maintenance delta type
- Considered mechanisms for acquiring such maintenance deltas

We are now ready to consider delta-specific integration methods.

Chapter 7
Integrating Maintenance Deltas into Derivation Histories

Chapter summary. A revised artifact can often be efficiently constructed by reusing parts of a derivation history from an existing artifact, and integrating a formal delta. This chapter provides procedures for technology and functional delta integration based on commutativity in the design space. A number of arguments for the presence of significant commutativity are considered.

An implementation is found by a difficult search of the design space for a path leading from f_0 to some implementation f_G. Given a maintenance delta, and a desire for a new implementation $f_{G'}$ that takes that delta into account, we could search the design space again, but that is expensive. If the change is relatively small, the derivation history for f_G may not be far from the correct one. We hope to reuse significant portions of the derivation history, avoiding much of the search involved in a pure reimplementation.

Reuse of the derivation history implies that we can somehow start the transformation system up after applying the transformations contained in the reusable derivation history. The transformation system must continue as though it had generated those transformations itself, adding new transformations to complete an implementation and/or backtracking to repair the partial derivation history it has as needed. While we have not discussed this, changing a transformation system to continue in this fashion is trivial enough so we will simply assume this ability.

Reuse of the derivation history implies reuse of the individual transformations. We can do this if we can somehow validate the effect of individual transformations with respect to the specification we are trying to implement. Such validation is only possible if we possess the original specification, the revised specification, some notion of their difference, and the design history, to tell us the role each transformation played in achieving the original result. We shall pursue this approach in Chapter 8, but we can get considerable mileage out of a first approximation:

Assume, unless it can be easily shown otherwise, that every transformation in the original derivation history will serve a useful purpose in a new one, and attempt to use it again.

This approximation is effective because of scale-induced commutativity in the design space; we expect that maintenance deltas will generally only have a small effect on our desired artifact. Most replay schemes (including ours) [MB87] make this assumption, and then use various strategies to clean up errors induced by the assumption.

A particularly simple scheme is *naive replay*. For a functional delta $\delta_f \in \Delta_f$, naive replay sequentially tries to apply the transformations $t_j^\ell = H[i]$ from the old derivation history H, in order of application $i = 1..length(H)$, to the revised specification $f_0' = \delta_f(f_0)$. Successful application causes t_j^ℓ to be retained; failed application causes that transformation to be dropped. Such a scheme has the disadvantage of blindly trying transformations without considering the effect of the change. We provide an analogy to show the flaw: naive replay is like hammering a nail into wood-block coordinates $(5, 12)$ to get a first implementation, deciding, next time, to move the nail to $(6, 9)$, and then trying to hammer again at wood-block coordinates $(5, 12)$ simply because that worked last time. Our heart is in the right place, but the hammer is not. We did not take into account the effect of the change on the locaters.

We have a different approach to derivation history reuse, which integrates the maintenance delta. *A key insight is based on the observation of commutative paths in the design space; often, the derivation history can be locally rearranged without affecting the end result.* It is important to notice that *such local rearrangements may retain the transforms, but change the locaters*, and still achieve the exact same result. This allows us the theoretical potential to rearrange a derivation history for our convenience into two parts: a part which we want to save, and a part which we do not know how to save. The rearrangement is determined by the maintenance delta. Reuse then consists of performing this rearrangement, and simply throwing away the part we do not know how to save. We replay the saved portion H_{saved}, constructing an end state $s_{saved} = \Pi H_{saved}(f_0')$ for the saved portion. Finally, we turn the transformation system back on to regenerate the tail of the derivation history from s_{saved}. In fact, we must allow the transformation system to attempt implementation by extending and/or revising the "reusable" part of the derivation history; so "reuse" really consists of purging the obviously *unusable* portions of the derivation history.

Delta Integration then consists of the logical steps:

1. Adjustment of specification or support technology according to the delta
2. Derivation history rearrangement consistent with the delta
3. Truncation of rearranged derivation history
4. Direct reuse of truncated derivation history
5. Completion of implementation from the end point of derivation history.

In practice, these logical steps may be interwoven. The directly reused part has many of the original transforms, with different locaters; we are reusing something more than just the transforms, but something less than the actual transformations[1].

The challenge:

- How do we know which transforms can be preserved?
- How do we change the locaters on preserved transforms?
- How do we rearrange the history before truncating?

We can determine transformations that are problematic by inspecting their interaction with a given delta; the transformations that can be saved are the ones without troublesome interactions. Rearranging consists of taking advantage of local commutativity in the design space to change the order in which transforms are applied; this will often dictate how to change the locaters. In practice, it is more efficient to truncate the derivation history during the rearrangement process.

Without the design history, it is difficult to detect interactions of most types of maintenance delta with individual transformations. Consequently, this chapter is mostly about integrating Δ_f. This is expected to be one of the more common types of deltas used in practice; the ARIES system [JF90] for managing evolution transforms implicitly assumes that Δ_f are the only interesting kinds of deltas, and most current transformation systems cannot even express other types of deltas because their performance goals are implicit. Furthermore, the basic techniques we use to handle functional deltas will turn out to be very useful for managing the other types of deltas. We will return to the other maintenance deltas in Chapter 8.

In this chapter, we provide methods for preserving portions of derivation history in the face of functional deltas, a theoretical basis that justifies the method, and some evidence that the method will work well in practice. A detailed example is provided to illustrate the method.

[1] Y. V. Srinivas (personal communication) has suggested that in a properly chosen topology, the transformations are indeed preserved intact. This idea has not been pursued in detail.

7.1 Delta Integration Overview for Various Types of Change

We first provide an overview of certain delta integration processes. We examine the effect of the delta integration process on the design space, emphasizing the key role commutativity plays (refer back to Figure 3.7). For a related view of how maintenance can take place in a conventional software construction process, see [ABFP86], in which alternative paths through the design space are stressed, but commutativity does not play a key role.

We consider this for the following types of delta:

- Performance: Δ_G
- Technology: Δ_C
- Functionality: Δ_f

This order of presentation is chosen because each one has successively larger effects on the shape of the design space. The other support deltas have an effect on the design space similar to that of performance deltas, so we do not examine them here.

We will formalize Δ_C-integration and Δ_f-integration in Sections 7.3 and 7.4. Formalizing Δ_G-integration must wait until Chapter 8 where we have access to design goals, although it is conceptually the simplest.

7.1.1 Effect of Performance Delta Δ_G

We restrict our attention to performance deltas applied to G_{rest}, as performance deltas applied to $G_{invariant}$ are usually cast as functional deltas δ_f, which we will discuss later.

Figure 7.1 shows a design space, and a particular implementation f_G found traversing a path from f_0 of property-preserving transforms. Now, f_G satisfies the remainder of the performance predicate, G_{rest}. In fact, there is a set of nodes in the design space satisfying G_{rest}, of which f_G is only one; Steier [SA89, p. 106] makes this same observation after examining 7 different algorithm syntheses. A delta $\delta_G : G_{rest} \rightarrow G'_{rest}$ results in a new performance goal $G_{invariant} \wedge G'_{rest}$, which picks out another set of implementations in the same design space.

The revision procedure in this case can retain much of the design history (i.e., $c_1^{\ell_1}$ and $c_2^{\ell_2}$) if it discovers implementation $f_{G'}$. Transformations $c_3^{\ell_3}$ and $c_4^{\ell_4}$ must

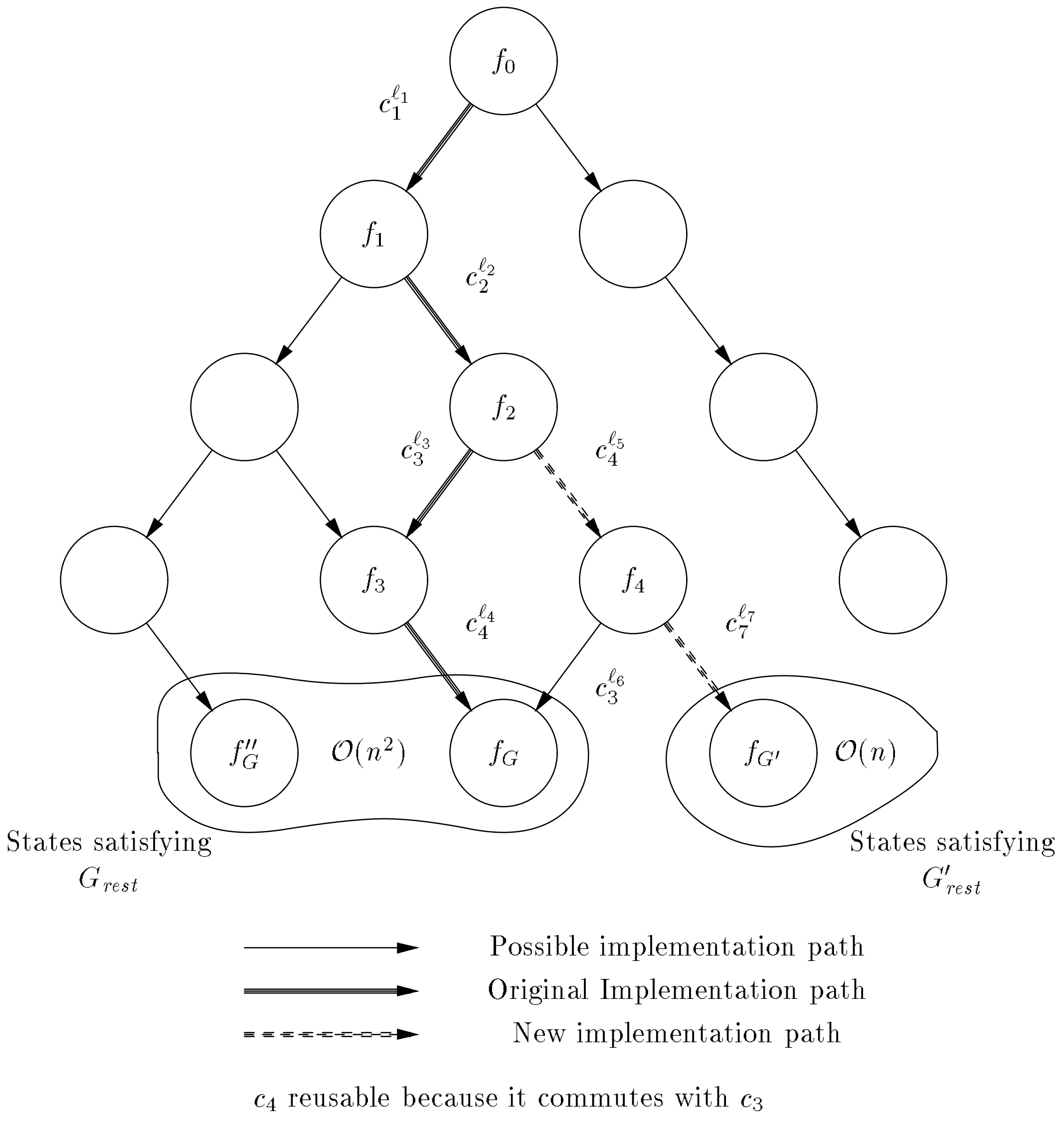


Figure 7.1: Changing performance: find new path in space

be removed from the derivation history by virtue of being the last steps leading to the now undesirable implementation f_G. The derivation history can be repaired by choosing $c_4^{\ell_5}$ and $c_7^{\ell_7}$, leading to $f_{G'}$.

The repair process must somehow choose these new transformations. We first observe that transform c_4 can be reapplied (assuming that it eventually leads to $f_{G'}$) because $f_G = c_4^{\ell_4}(c_3^{\ell_3}(f_2)) = c_3^{\ell_6}(c_4^{\ell_5}(f_2))$ implies that $c_4^{\ell_5}(f_2)$ is well defined; we loosely say that transforms c_3 and c_4 *commute*. What is not reusable is the locater for c_4. $c_7^{\ell_7}$ must be generated fresh by the repair process; there is no hint of it in the derivation history.

The interesting problems here are:

- which transformations must be dropped?
- which transformations can be preserved intact?
- which can be preserved with new locaters?
- what should be the value of the new locaters?
- when should new transformations be generated?

Making the desired change explicit (δ_G) will provide us with the needed answers. We will take this up in detail later.

The support deltas other than Δ_C only affect the set of implementations presumed desirable. At the level of the design space, they are indistinguishable from the overall effect of Δ_G on G_{rest} because they only change the the performance goal G_{rest}, so we do not consider them further.

7.1.2 Effect of Δ_C

In Figure 7.2, we see the effect of changing the set of property-preserving transforms C_i usable by the transformation system. The only changes one can make to a set are to delete elements (as shown for c_2), and to add new elements (c_5, c_6). Changing the set of property-preserving transforms changes the shape of the design space. Old possible paths ($c_4^{\ell_4}(c_3^{\ell_3}(c_2^{\ell_2}(f_1)))$) and implementations ($f_G$) disappear; new potential paths ($c_6^{\ell_6}(c_5^{\ell_5}(c_3^{\ell_7}(f_1)))$) and implementations f'_G satisfying the performance predicates appear.

Even though c_2 is no longer legitimate, we can use commutativity in the original design space with respect to $c_3^{\ell_3}$ to note the potential reusability of c_3. We do this by noting that in the original design space, $f_3 = c_3^{\ell_3}(c_2^{\ell_2}(f_1)) = c_2^{\ell_8}(c_3^{\ell_7}(f_1))$, which implies that $c_3^{\ell_7}(f_1)$ is well defined *even in the revised design space*. Consequently we can

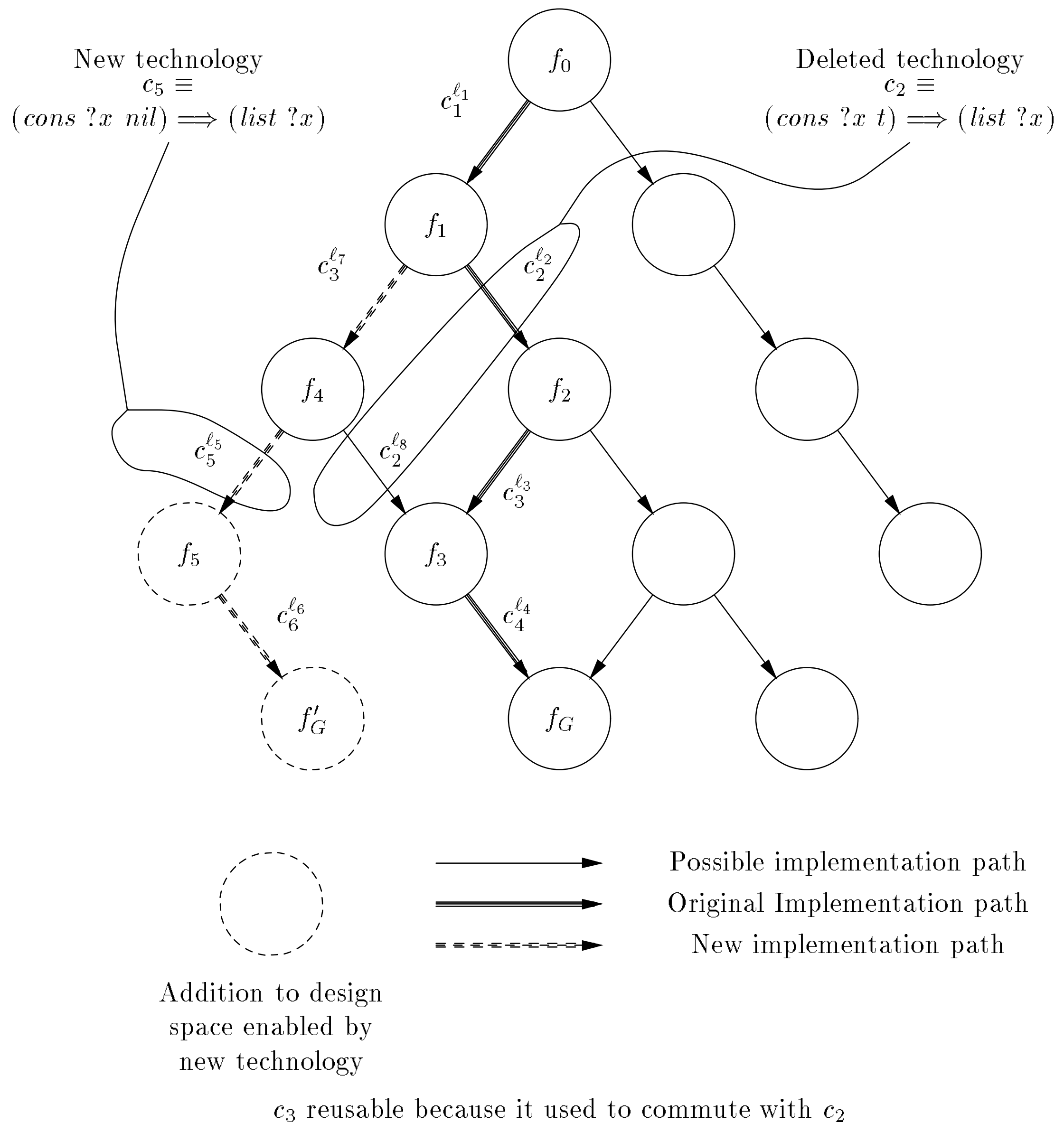

New technology
$c_5 \equiv$
$(cons\ ?x\ nil) \Longrightarrow (list\ ?x)$
Deleted technology
$c_2 \equiv$
$(cons\ ?x\ t) \Longrightarrow (list\ ?x)$
f_0
$c_1^{\ell_1}$
f_1
$c_3^{\ell_7}$
$c_2^{\ell_2}$
f_4
f_2
$c_5^{\ell_5}$
$c_2^{\ell_8}$
$c_3^{\ell_3}$
f_5
f_3
$c_6^{\ell_6}$
$c_4^{\ell_4}$
f'_G
f_G
Possible implementation path
Original Implementation path
New implementation path
Addition to design space enabled by new technology
c_3 reusable because it used to commute with c_2

Figure 7.2: Changing technology: reject an old path or enable a new path

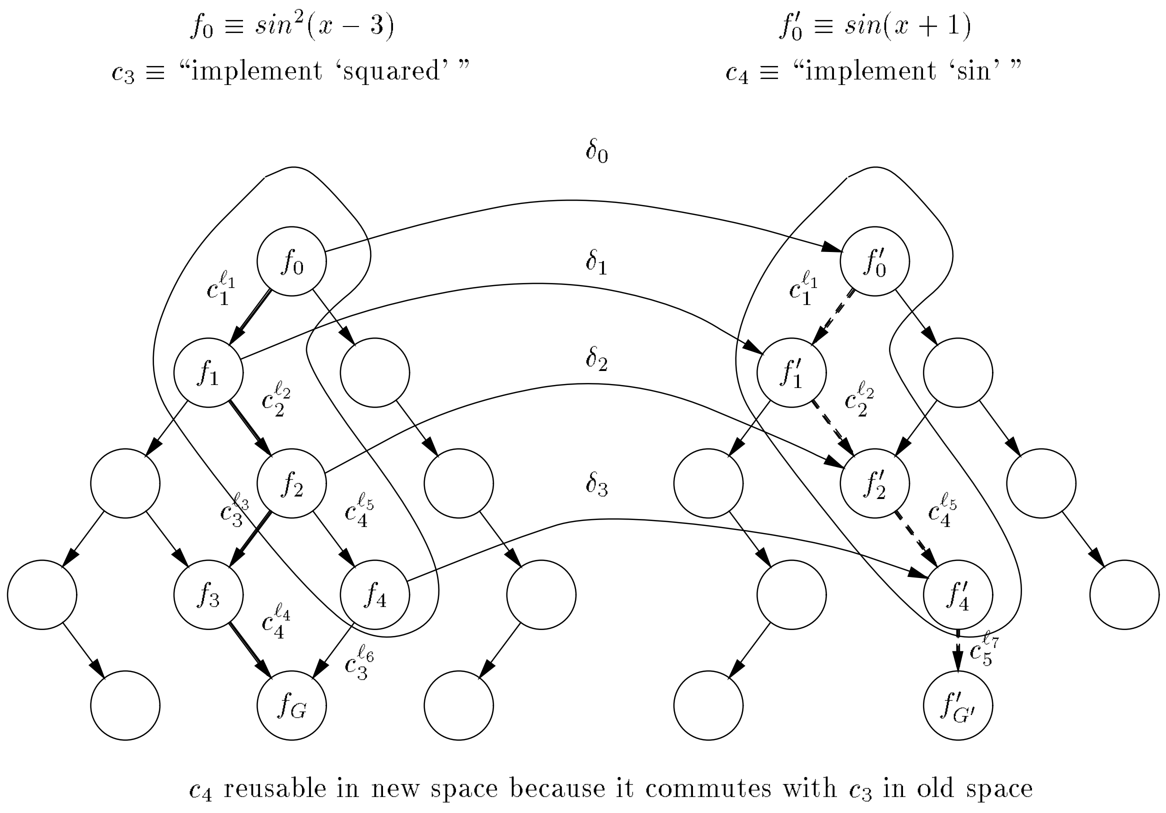


Figure 7.3: Changing functionality: preservation of path across design spaces

reuse transform c_3 with the new locater ℓ_7. The transformations $c_5^{\ell_5}$ and $c_6^{\ell_6}$ must be generated as repairs, as there is no hint of them in the original derivation history.

7.1.3 Effect of Δ_f

Changing the functional part of a specification (δ_f) completely changes the design space from that of f_0 to f'_0, in which the new implementation must be found (Figure 7.3). In one sense, the original path is entirely irrelevant, and so an entirely new path must be constructed in the new space. In another sense, there should be a close analog of the original path in the new space.

Applying δ_f changes $G_{invariant}$ to $G'_{invariant}$. Now, any preservable transformation c^{ℓ} must have the property $c \in \mathcal{C}_{G_{invariant}} \wedge c \in \mathcal{C}_{G'_{invariant}}$, for otherwise it would be a non-property-preserving transform for one of the two spaces and could not be preserved. For delta types which do not change $G_{invariant}$, this is trivially true, and we therefore need not check this condition. With δ_f, we must check this condition, with one exception. If we know that $G_{invariant}(f) \equiv f \succeq_{invariant} p_{invariant}(f_0)$, as it is with most transformation systems, and the transforms c used are all $p_{invariant}$-preserving, then the condition is always true and we can avoid the check. We call such a legitimately applicable transform δ_f-preservable.

Given a δ_f-preservable transformation, it can be tried in the new design space. If it doesn't fail to apply, it is at least safe to use, even if it does not help with G_{rest}. (We remind the reader that a functional delta does not affect G_{rest}).

From the beginning of the new path (f_0'), the old transformations can be tried sequentially. Each transformation which is applicable can still be legally applied (as exemplified by $c_1^{\ell_1}$ and $c_2^{\ell_2}$); transformations which no longer apply (such as $c_3^{\ell_3}$) can simply be dropped (naive replay). We depart from naive replay by using a more sophisticated technique to save inapplicable transformations: if an undesirable transformation commutes with its successor, we can delay the undesirable one and attempt to preserve the successor instead (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). Once again, commutativity rescues us.

7.2 Basic Mechanisms for Rearranging a Derivation History

In each case where we wish to reuse a derivation history, we find it valuable to rearrange that derivation history, leaving the net effect alone, before trying to apply it to the new problem. This rearrangement is usually necessitated by the presence of a transformation which will be inappropriate in a solution to the new problem. We have shown in Section 7.1 that "commutative" transformations play a key role in such rearrangements.

We categorize some basic mechanisms, based on commutativity, for exiling an unwanted transformation as follows:

- Delay: delay application of a transformation until later
- Swap: exchange the order of two transformations in a derivation history (a special case of Delay)
- Banish: push a transformation down to the end of a history (and delete it)

These mechanisms are used in virtually all of the delta integration procedures as removing an inappropriate transformation is a fundamental need. We will discuss each of these in turn before turning to specific delta integration procedures.

We assume that we have an existing derivation history $H = [t_1^{\ell_1}, \ldots, t_k^{\ell_k}]$, and some predicate:

$$undesirable_H : \{1..k\} \rightarrow \mathbf{Boolean}$$

where $\{1..k\} \subset \mathbf{Nat}$. The predicate *undesirable* specifies which transformations in H are no longer appropriate. (Its complement identifies transformations which are not known to be undesirable, as opposed to known to be definitely reusable). This predicate is the result of some analysis of a delta with respect to the derivation history. A sample *undesirable* useful for pedagogical purposes designates the first transformation as undesirable, and the rest as acceptable, i.e., $undesirable(1) = true$. We will see some actual definitions of *undesirable* later.

7.2.1 Delaying an undesirable transformation

We *delay* undesirable transformations, by taking advantage of commutative paths of the design space. The idea is to revise the original derivation history H in such a way that the original program $f_{length(H)} = \Pi(H)(f_0^H)$ is not affected, but application of the undesirable transformation is delayed until a later time, and is replaced by a transformation which is not undesirable. In this section, we characterize an idealistic *DELAY* procedure to help us accomplish this. It is difficult to construct such a general *DELAY* procedure in practice for a number of reasons we will outline, but we can construct interesting specializations using related procedures called *SWAP* and *DEFER*. Thus *DELAY* provides theoretical motivation.

Our ultimate intention is to delay application of undesirable transformations until all the acceptable transformations have been applied. We do this by repeatedly replacing a subsequence $H_{replaced} \subseteq H$ of transformations by another sequence $H_{replacement}$ with equivalent effect, but different initial transform $H_{replacement}[1]$ (Figure 7.4).

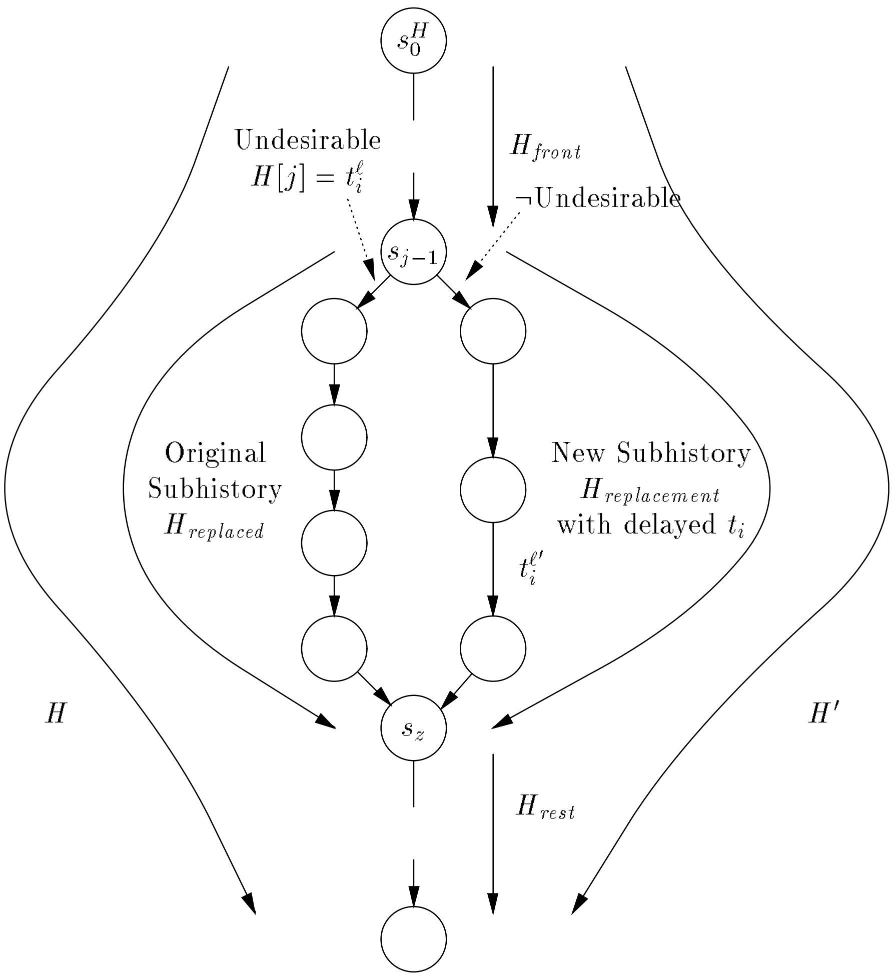


Figure 7.4: Delaying undesirable transformation $t_i^{\ell'}$

We define the function

$$DELAY : \mathcal{S} \times \mathcal{H} \times (\mathbf{Nat} \rightarrow \mathbf{Boolean}) \times \mathbf{Nat} \rightarrow \mathbf{Boolean} \times \mathcal{H} \times (\mathbf{Nat} \rightarrow \mathbf{Boolean})$$

acting on a state s_0^H, a derivation history H, a predicate *undesirable*, and an index j such that

$$DELAY(s_0^H, H, undesirable_H, j) = \langle b, H', undesirable_{H'} \rangle$$

where b is a boolean signifying success in delaying $H[j]$, H' is a derivation history in which application of $H[j]$ has been delayed, and $undesirable_{H'}$ marks undesirable transformations in the revised history.

After invocation of $DELAY$, the following will be true:

$$\begin{array}{l} undesirable(j) = true \wedge b = true \supset \\ \quad \exists H_{front}, H_{replaced}, H_{rest}, H_{replacement} : \\ \quad\quad H[1..j-1] + H_{replaced} + H_{rest} = H \\ \quad\quad H[1..j-1] + H_{replacement} + H_{rest} = H' \\ \quad\quad s_{j-1} = \Pi(H[1..j-1])(s_0^H) \\ \quad\quad s_z = \Pi(H_{replaced})(s_{j-1}) = \Pi(H_{replacement})(s_{j-1}) \\ \quad\quad length(H_{replacement}) \geq 2 \\ \quad\quad undesirable_{H'}[1..j-1] = undesirable_H(1..j-1) \\ \quad\quad undesirable_{H'}(j) = false \\ \quad\quad undesirable_{H'}[j + length(H_{replacement})..length(H')] = \\ \quad\quad\quad undesirable_H[j + length(H_{replaced})..length(H)] \end{array}$$

If $b = false$, then there is no way to delay $H[j]$ further.

The $DELAY$ operation allows us to push a single undesirable transformation $H[j]$ "deeper" into H', and temporarily allows us to avoid dealing with it. This pushing process can technically make $H[j]$ disappear as a recognizable entity, but this is unimportant to us, as we are only interested in equivalence of effect. It is entirely possible that $H[j]$ and even $H^{\mathcal{T}}[j]$ are not present in $H_{replacement}$ (we will discuss an example of this in Section 7.2.2).

We leave open precisely how $undesirable_{H'}[j + 1..j + length(H_{replacement})]$ is defined and/or computed. One could repeat the delta-versus-H analysis process to fill it in, or one could very conservatively define $undesirable_{H'}$ to be *true* over this entire interval, depending on the cost of the analysis. *Irrespective of how $H_{replacement}$ is defined, at least one of its transformations must end being marked as undesirable if we assume that the reason that a transformation is undesirable is its effect. DELAY* doesn't remove the problem; it merely delays it.

Because delaying a transformation does not depend on any property-preservation effects, $DELAY$ may be applied to any transformation in the derivation history, including the evolution transformations between ϵ and f_0.

A suitable $H_{replacement}$ may not exist, that is, it may not be possible to delay $H[j]$ any further in the history (i.e., $b = \mathit{false}$). We shall address this topic further during the discussion on the *BANISH* procedure.

We note that the multiplicity of paths in the design space allows the *DELAY* function to produce any of several possible results, of which we arbitrarily accept any one. Future research is needed to determine how to choose a potential $H_{replacement}$ that maximizes reusability.

Given *DELAY* and the predicate *undesirable* it is easy to construct a conceptual procedure to rearrange a derivation history into reusable and reuseless parts:

$$\mathit{PARTITION} : \mathcal{S} \times \mathcal{H} \times (\mathbf{Nat} \rightarrow \mathbf{Boolean}) \rightarrow \mathcal{H} \times \mathcal{H}$$

The procedure operates by scanning the derivation history from beginning to end, and delays undesirable transformations until some undesirable but undelayable transformation is found. Code for such a procedure is shown in Figure 7.5. Since the procedure only applies *DELAY* to the history, the resulting histories, concatenated, are an equivalent path to the original history. The procedure *PARTITION* cannot fail; at worst it will produce an empty reusable history. Thus a *DELAY* procedure provides us with a way to determine potentially reusable portions of a derivation history. When attempting to reuse portions of a derivation history, one can run the *PARTITION* procedure and discard the second (reuseless) result immediately; the first result consists only of transformations that are not *undesirable* and are therefore likely reusable. An obvious optimization is to simply drop the reuseless result.

In general it is difficult to construct a *DELAY* procedure to find a suitable $H_{replacement}$ that satisfies the required properties; we consequently fall back on a number of heuristics to make this computation easier.

One complication is that even when $H_{replacement}$ theoretically exists, it may not be practical to compute; in the face of conditional transformations, one might need a full theorem prover to determine path equality. A simple cure is for *DELAY* to declare failure if the computational energy to compute the correct answer exceeds some arbitrary bound; we call this heuristic a *conservative cutoff*. Such cutoffs can at worst prevent the *PARTITION* procedure from saving as much of the derivation history as theoretically possible, but, like a conservative data flow analysis [Kil73, ASU86] it cannot make the result incorrect. We currently have no specific suggestions as to how to choose the computation bound, although we are inclined to be generous under the assumption that *IMPLEMENT* will likely have to generate roughly one transformation for each one lost by *PARTITION*, and *IMPLEMENT* is expected to be expensive.

```
Procedure PARTITION(State: StartState, DerivationHistory:History,
    Boolean Function: undesirable)
  Returns DerivationHistory, DerivationHistory
  Declare Integer: j, Boolean: SuccessFlag
  RevisedHistory:=History
  j:=1
  While j< length(RevisedHistory) do
    If undesirable(j) Then
      ⟨SuccessFlag,RevisedHistory,undesirable⟩:=
        Delay(StartState,RevisedHistory,undesirable,j)
      If ¬SuccessFlag Then
        % RevisedHistory[j] cannot be delayed any further
        Return ⟨RevisedHistory[1..j-1],rest(RevisedHistory,j)⟩
      Fi
    Fi
    j:=j+1 % Continue scanning towards end
  End While
  % This place not normally reached.
  Return ⟨ RevisedHistory,EmptyHistory⟩
End PARTITION
```

Figure 7.5: Procedure to partition derivation history using *DELAY*

A second complication is the expense involved in validating the *DELAY* output requirement:

$$\Pi(H_{replaced})(s_j^H) = \Pi(H_{replacement})(s_j^H)$$

A considerable portion of this cost can be traced to the involvement of the state value, which, by assumption in the software construction environment, is likely to be bulky. A simple heuristic to lower this cost is to compare the composed transformations directly, avoiding use of the state, by validating:

$$\Pi(H_{replaced}) = \Pi(H_{replacement})$$

A failure to prove equality can fall back on a state-based computation, or simply apply conservative cutoff.

A third complication is the necessity to find candidate arbitrary chains of transformations $H_{replacement}$. A related problem is justifying the transformations in a nearly arbitrary $H_{replacement}$; validating such transformations would be much simpler if the transforms involved were already justified by the design history for H.

A heuristic for handling both the cost of validations involving state and the difficulty of locating arbitrary equivalent chains is to specialize *DELAY* to simply exchange two sequential transforms, called *swapping transformations*, which proves to be relatively easy in practice.

7.2.2 Swapping two sequential Transformations

A *DELAY* procedure is difficult to implement in practice. However, *SWAP*, a specialization of *DELAY*, can often be implemented relatively easily. This procedure exchanges the order of two sequential transformations. We define the function

$$SWAP : \mathcal{S} \times \mathcal{X} \times \mathcal{X} \rightarrow \textbf{Boolean} \times \mathcal{X} \times \mathcal{X}$$

such that

$$SWAP(s, t_1^{\ell_1}, t_2^{\ell_2}) = \langle b, t_2^{\ell_2'}, t_1^{\ell_1'} \rangle$$

with the constraint that

$$b = true \supset t_2^{\ell_2}(t_1^{\ell_1}(s)) = t_1^{\ell_1'}(t_2^{\ell_2'}(s)).$$

What *SWAP* does is to commute the transformations ($b = true$), possibly revising the locaters, or complain that it cannot effect the exchange ($b = false$).

Given a *SWAP* procedure, a *DELAYBYSWAP* procedure can be implemented for $undesirable(j)$ by swapping transformations $H[j]$ and $H[j+1]$ after checking that $undesirable(j+1) = false$. We will later discuss a procedure, *BANISH*, that handles the case of $undesirable(j+1) = true$.

Implementing *SWAP*

We do not define precisely how *SWAP* works. The key problem is generating the revised locaters and testing whether the resulting transforms meet the desired results. While we model of *SWAP* as British Museum algorithm, in Appendix B, which simply enumerates locaters to try, knowledge of the structure of the program and transform representation should allow one to build more efficient procedures that can decide this very quickly for most transformations (as well as computing the revised locaters), or report "Unknown" for the rest. The "Unknown" answer can be conservatively treated as "Transformations do not commute". We therefore think that implementing *SWAP* with moderate efficiency is not difficult. Since *SWAP* is a specialization of *DELAY*, remarks about heuristics to make the computations more tractable equally apply.

There is a special case that is common enough so that every implementation of *SWAP* is likely to handle it. When the locaters act as geometric constraints and specify places that are "far apart", *SWAP* can report success and literally just copy the locaters, as the "distance" between the binding sites of the transforms is enough so the transformations have no effect on one another. *We expect this case to be very common because of scale*: the size of the state for interesting programs is likely to be large, and so most randomly chosen places are "far apart"[2]. We believe that dependency networks [Fik75, Lon78] are a promising way to detect this case.

We consider tree transforms to demonstrate that it is possible to implement for *SWAP* for some representations, and to provide some examples. For tree transformations, locaters are geometric constraints. Two tree locaters specify places that are far apart if the paths they select diverge, i.e., one path is not a prefix of the other. Figure 7.6 shows two such tree transformations and their swapped equivalents. Note that the locaters do not even change; truly the transformations swap in this case. This is the case we expect to be common due to scale. For trees, divergent paths for locaters ensures that the transformations commute, and so no dynamic test for equivalence of result is needed.

When one tree transformation locater is the prefix of another, often a rather messy but straightforward analysis of how subtrees (or leaves) are rearranged by each

[2]Here is an example of how the constraint aspect of locaters can be used to advantage. If one interprets a tree path locater as "apply the transform in the only place it is valid in the selected subtree", then one can actually abbreviate path locaters. This saves space in a derivation history. Under the assumption of large states, most path locaters select "places" that are far apart. Since two transformations may be swapped if their locaters are mutually inconsistent, this abbreviation still allows most *SWAP*s to go thru as though the abbreviation had not occurred.

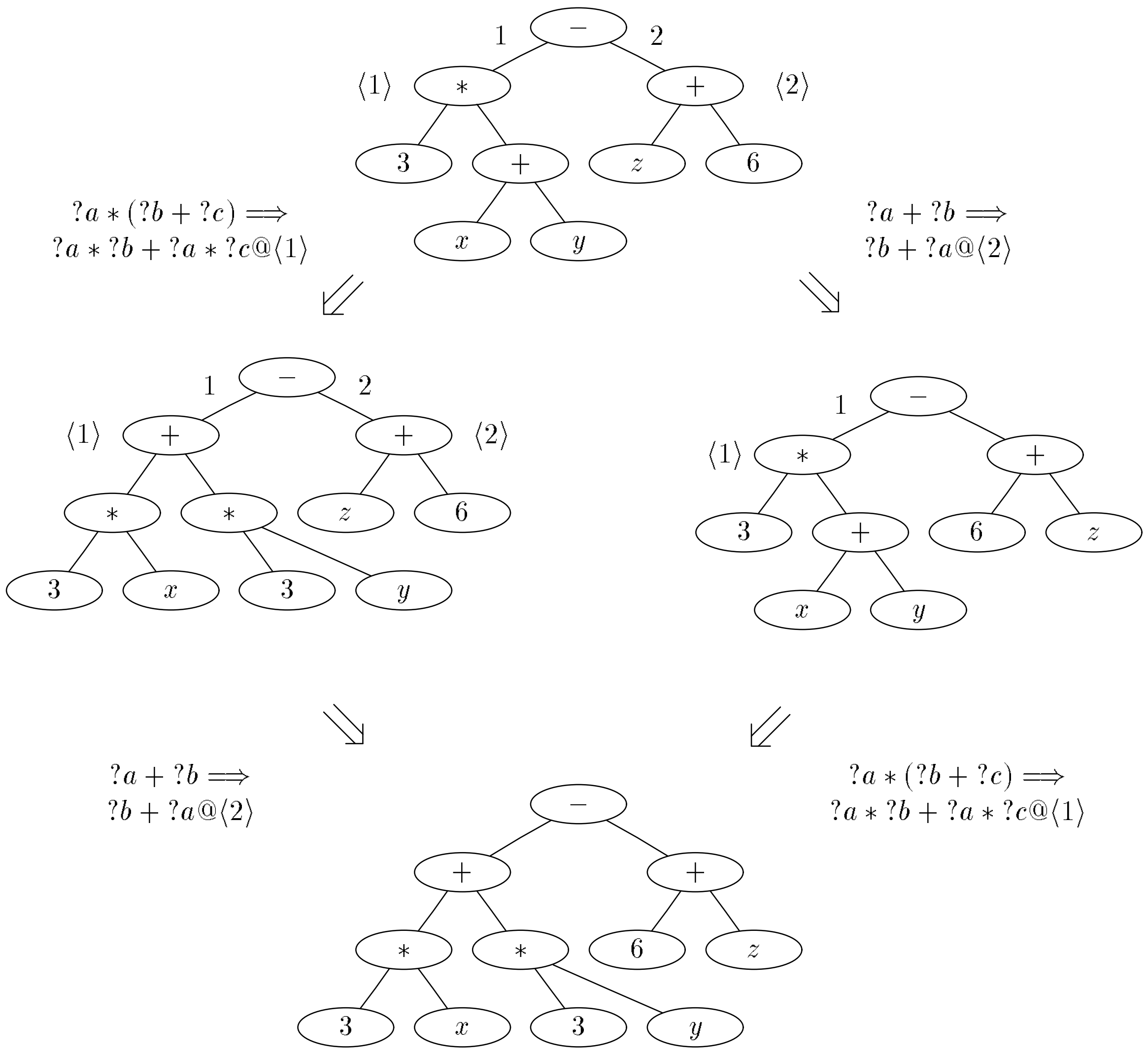
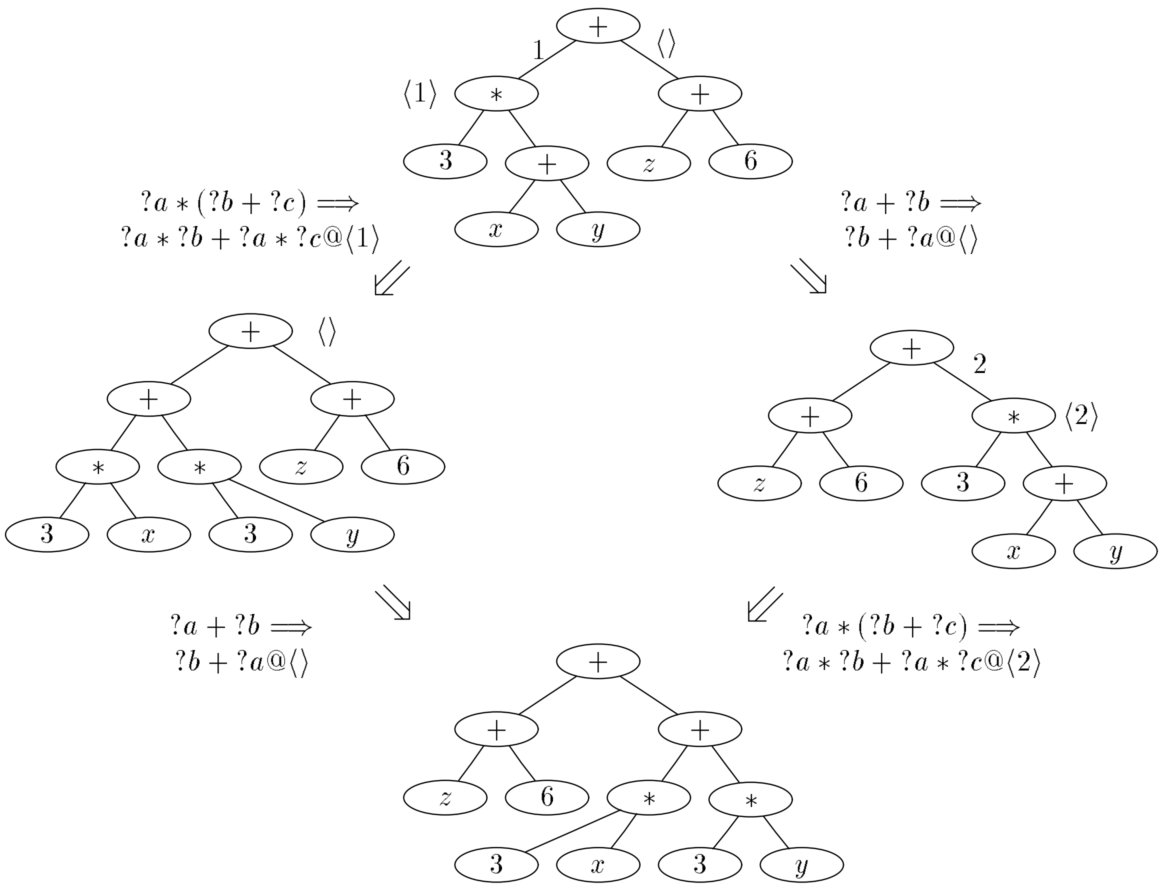

$?a * (?b + ?c) \Longrightarrow$
$?a * ?b + ?a * ?c@\langle 1\rangle$
$?a + ?b \Longrightarrow$
$?b + ?a@\langle 2\rangle$
$?a + ?b \Longrightarrow$
$?b + ?a@\langle 2\rangle$
$?a * (?b + ?c) \Longrightarrow$
$?a * ?b + ?a * ?c@\langle 1\rangle$

Figure 7.6: Swapping order of two independent sequential transformations



$?a * (?b + ?c) \Longrightarrow$
$?a * ?b + ?a * ?c@\langle 1\rangle$
$?a + ?b \Longrightarrow$
$?b + ?a@\langle\rangle$
$?a + ?b \Longrightarrow$
$?b + ?a@\langle\rangle$
$?a * (?b + ?c) \Longrightarrow$
$?a * ?b + ?a * ?c@\langle 2\rangle$

Figure 7.7: Swapping order of two overlapping sequential transformations

tree transform can provide most of the information needed to determine commutability and revised locaters. Figure 7.7 shows how the leaves of one transform move the entire effect of another.

Sometimes the applications of the transformations to be swapped overlap. When this occurs, one may have to enumerate the places in one transform result as possible points of application of the other transformation in order to generate candidate locaters. The resulting proposed transformations may actually need to be dynamically composed to verify equality (check that $t_1^{\ell_1} \circ t_2^{\ell_2} = t_2^{\ell_2'} \circ t_1^{\ell_1'}$). Figure 7.8 is such a case.

Johnson [JF90] provides another concrete example of an implementation of *SWAP*, for a more complex representation for programs, a semantic network. He determines if two evolution transforms (non-property-preserving transforms) affect one another by considering which semantic links they affect. If t_1 inspects only semantic links of type A and affects only links of type A', and t_2 inspects only links of

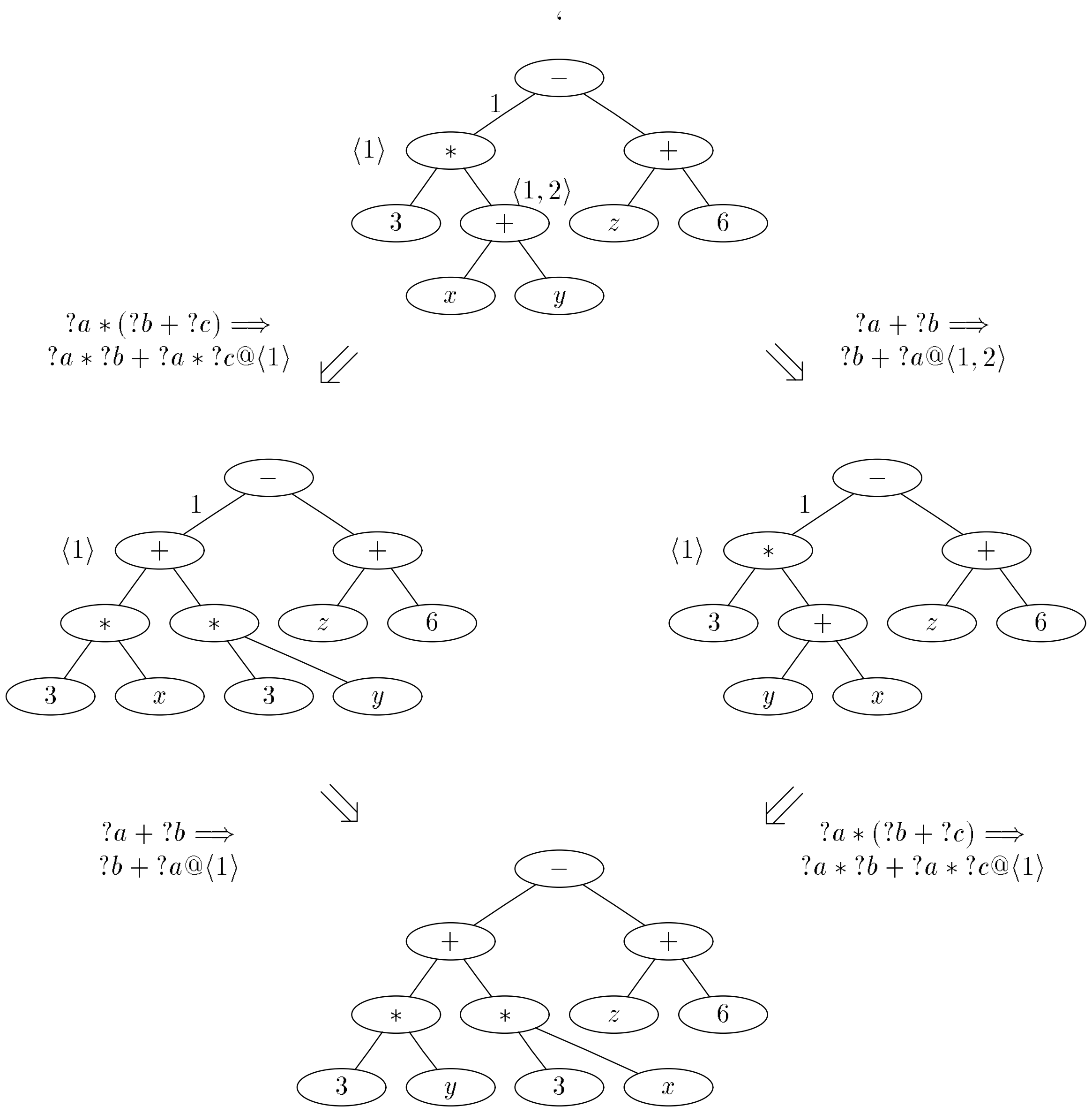


Figure 7.8: Swapping two transformations with considerable overlap

type B and affects only links of type B', and $A \cap B' = \emptyset$ and $B \cap A' = \emptyset$, then the two transforms are independent. This type of shallow analysis is sufficient to decide if two transformations trivially commute, because the indirect performance values are determined only by the state, and not by the transforms or their order.

Deferring the application of a transformation

A generalization of *SWAP* allows the swapped transforms to also be changed. We define:

$$\mathit{DEFER} : \mathcal{S} \times \mathcal{X} \times \mathcal{X} \rightarrow \textbf{Boolean} \times \mathcal{X} \times \mathcal{X}$$

such that

$$\mathit{DEFER}(s, x_1, x_2) = \langle b, x_2', x_1' \rangle$$

with the constraint that

$$b = \mathit{true} \supset x_1' \neq x_1 \wedge x_2(x_1(s)) = x_2'(x_1'(s))$$

We call x_1 the *deferred* transformation, and x_1' the *promoted* transformation. The boolean signals whether x_1 was successfully deferred. Either or both resulting transforms may be different than the originals. This generalization can be used when delaying the application of a domain-specific transformation past a theory morphism (see Section 3.1.7 for definitions of these), or vice-versa, as shown in Figure 7.9.[3]

One would expect that computing *SWAP* would generally be simpler than computing *DEFER*, because for *SWAP* the transforms are constant and need not be recalculated. Since *DEFER* is more general than *SWAP*, we would prefer to use it instead. These facts suggest that an implementation of *DEFER* would actually try to perform a *SWAP* first, and failing that, would fall back on the more general computation. With this in mind, we will use the term *swap* to refer to the action *DEFER*.

We have implemented both *SWAP* and *DEFER* for conditional tree transformations (as well as the examples shown in this thesis) by using subtree-tracing and a simple theorem prover to validate equivalence of compositions.

[3]Deferring an optimization past a refinement should always be relatively easy to do, because such an optimization will always have a corresponding optimization in the lower domain, effectively generated by applying the refinement to the optimizing transform itself. Deferring a refinement past an optimization cannot always be done; the source domain may simply not have the vocabulary. See the Δ_f integration example for such a case.

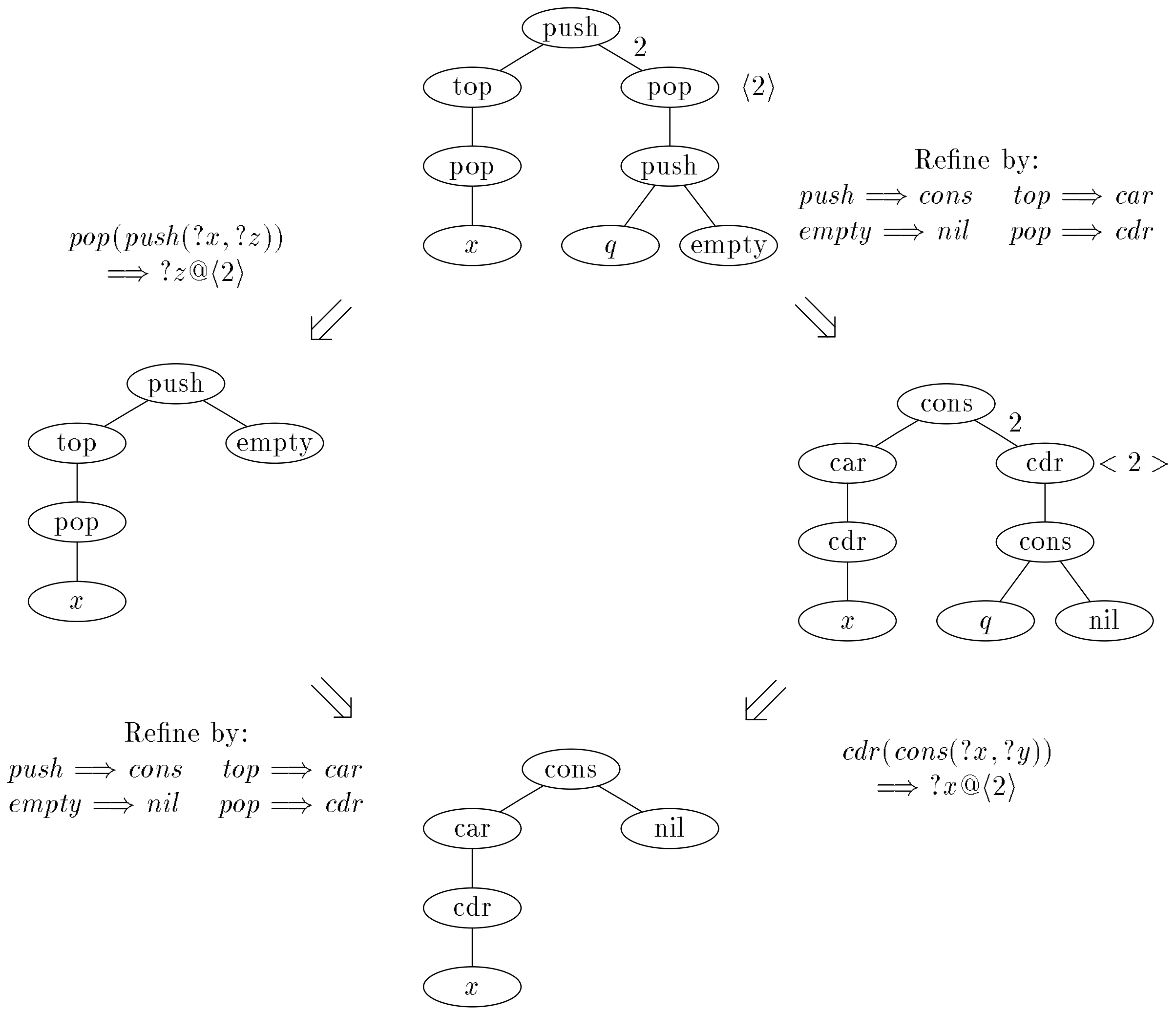


Figure 7.9: Deferring a transformation

Derivation History permutations

We define permutations of derivation histories to allow us to demonstrate equivalences of certain kinds of histories.

Given a derivation history H, we define a transposition of H to be those H' in which a pair of transformations have been swapped, i.e.,

DEFINITION 7.1: *transp*(H). A relation $\mathcal{H} \times \mathcal{H}$:

$$\begin{array}{ll}\{\,\langle H, H'\rangle \mid & H' = H[1..j-1] + x_1 + x_2 + rest(H, j+2) \wedge \\ & j < length(H) - 1 \wedge \\ & DEFER(H[j], H[j+1]) = \langle true, x_1, x_2\rangle\}\end{array}$$

□

A *permutation* of H_{base} is any member of the transitive closure of the transpositions of H_{base}.

DEFINITION 7.2: *permutation*(H). Any member of the set of permutations,

$$HPERMS(H_{base}) = \{\, H' \mid \langle H, H'\rangle \in transp^*(H_{base})\,\}$$

□

It should be obvious that every permutation H' of a derivation history H_{base} leads to exactly the same implementation, given the same initial state, i.e.,

$$\Pi(H_{base})(s_0^H) = \Pi(H')(s_0^H)$$

One typically applies *DEFER* to a pair of transformations $H[j]$ and $H[j+1]$ when $undesirable(j) = true$, exchanging $H[j]$ and $H[j+1]$ to produce a revised H'. The marking function $undesirable_{H'}$ corresponding to H' must be changed to reflect the new position of the exchanged transformations, i.e.,

$$undesirable_{H'}(i) = \begin{cases} undesirable_H(j+1) & \text{if } i = j \\ undesirable_H(j) & \text{if } i = j+1 \\ undesirable_H(i) & \text{if } i \neq j \wedge i \neq j+1 \end{cases}$$

We will assume that the function *undesirable* is revised in this fashion whenever *DEFER* or *SWAP* is applied.

7.2.3 Banishing a Transformation

The *DEFER* procedure is useful only for delaying an undesirable transformation momentarily. To effectively get rid of it, we push the undesirable transformation as far towards the end of the the derivation history as possible by repeated application of the *DEFER* procedure, and removal of unnecessary stumbling blocks. We call this *banishing*. A transformation is unlikely to be banishable to the far end of the derivation history as intermediate transformations may depend upon it.

We define

$$\mathit{BANISH} : \mathcal{H} \rightarrow \mathcal{H}$$

so that it banishes the first transformation $H[1]$ of an argument H, producing a revised derivation history H'. We assume $\mathit{undesirable}_H(1) = \mathit{true}$, for otherwise we would have no reason to run *BANISH*.

We will define *BANISH* in terms of an auxiliary,

$$\mathit{BANISH0} : \mathcal{H} \rightarrow \mathcal{H} \times \mathbf{Nat}$$

such that if

$$\mathit{BANISH0}(H) = \langle H', j \rangle$$

then the following are true (see Figure 7.10):

$$\begin{array}{l} H' \in \mathit{HPERMS}(H) \\ \forall i : 1 \leq i < j \supset \mathit{undesirable}_{H'}(i) = \mathit{false} \\ j \leq \mathit{length}(H') \supset \mathit{undesirable}_{H'}(j) = \mathit{true} \\ \forall i : j < i < \mathit{length}(H') \supset \mathit{DEFER}(H'[i], H'[i+1]) = \langle \mathit{false}, x_1, x_2 \rangle \end{array}$$

These conditions tells us that H' is truly just a rearrangement of H with the same resulting state, that some of the undesirable effect of $H[1]$ has been moved to $H'[j]$, and that all $H'[k] \in \mathit{rest}(H', j)$ are dependent on $H'[j]$. We call the index j the *blocking point* of H', because the undesirable transformation $H'[j]$ cannot be deferred any further in the history in a useful way.

The original point of banishing a transformation was to get rid of it. The transformations with larger indexes than the blocking point j are all reuseless because of their dependency on $H'[j]$. *BANISH0* thus computes a partition of the original derivation history. Rather than retain the reuseless portion, we can simply truncate H' at the blocking point j computed by $\mathit{BANISH0}(H)$. So we define

$$\mathit{BANISH0}(H) = \langle H', j \rangle \supset \mathit{BANISH}(H) = H'[1..j-1]$$

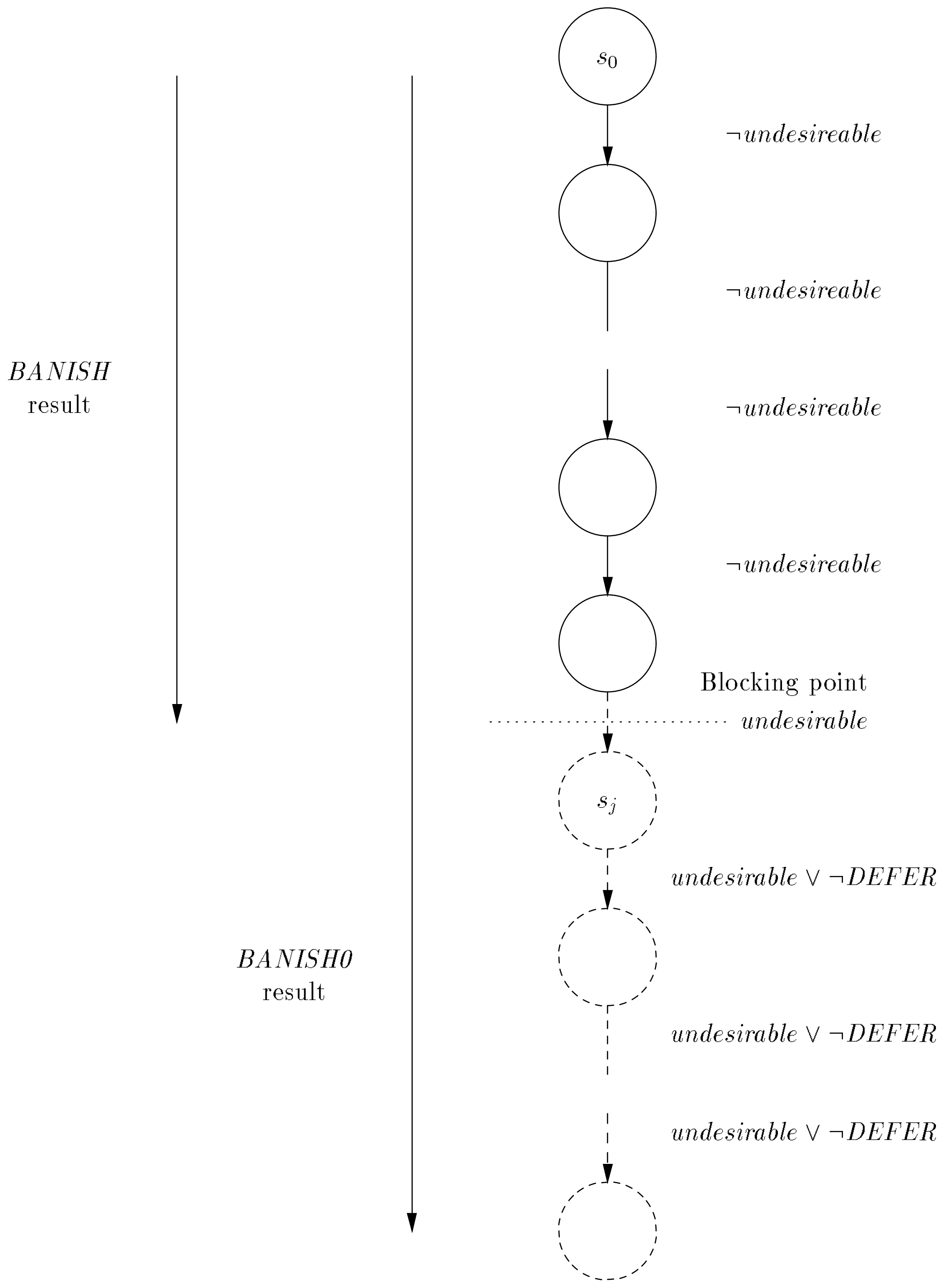


Figure 7.10: Banishing a transformation

Rather than actually compute the partition by executing *BANISH0*, we implement *BANISH* by incrementally dropping transformations which will provably be at or beyond the partition point. If an undesirable transformation is at the end of a history, we can simply drop it. If the undesirable transformation can be successfully deferred, then the promoted transformation is transferred to the reusable portion of the history. If it cannot be successfully deferred, then the following transform is an obstacle; we simply *BANISH* the obstacle, and attempt to defer the original again. Code for *BANISH* is shown in Figure 7.11. The implementation lowers costs in two ways: the derivation history shrinks by a least one transformation per call, so successive calls are cheaper by at least unit energy, and it prevents a second banished transform from being unnecessarily pushed into the dependents of the first, which does not improve the reusability of the second. As long as *BANISH* is invoked on a nonempty history, it cannot fail.

On close examination, one can see that *BANISH* combines the steps rearrange, truncate, and replay, partly out of necessity; the internal routine *DEFER* requires a state, which is most easily obtained by replaying the previously saved transformation.

Note that *BANISH* does not require that any participating transforms be property-preserving transforms. This means that *BANISH* can be applied to Δ_f (evolution transformations) as well as property-preserving transforms selected by the transformation system during its normal course of operation. This observation is used by the derivation history replay mechanism in Appendix B.

BANISH deletes the first transformation and its dependents from a derivation history. To banish a transformation $H[i]$ in the middle of a derivation history we can use a function *BANISHATPOINT* to split the history before i, banish from that point, and combine the pieces. We define:

$$\mathit{BANISHATPOINT} : \mathcal{H} \times \mathbf{Nat} \to \mathcal{H}$$

such that

$$\mathit{BANISHATPOINT}(H, i) = H[1, i-1] + \mathit{BANISH}(rest(H, i))$$

We can thus use *BANISH* to get rid of any undesired transformation anywhere in an existing derivation history, given its index.

BANISH as dependency-directed backtracking

The *BANISH* procedure can be useful during transformational implementation as a form of dependency directed backtracking. Conventional (chronological) backtracking during implementation requires that the last applied transformation be

```
Function BANISH(Program: CurProgram,
    DerivationHistory: History)
  Returns ⟨Program,History⟩
 % This function pushes History[1] as deep into the history as possible,
 % chops the history off at that point, and returns the revised history.
 % Because we always chop the history off, banishing cannot fail;
 % at worst it returns an empty history.
 % Complication: History[1] may conflict with History[2], so we can't always
 % immediately get rid of History[1]; we solve this by (recursively)
 % getting rid of History[2] and then proceeding.
 % This procedure costs O(length(History)^2) to run.
 Declare Program: PartialImplementation, Boolean: SuccessFlag
 Declare DerivationHistory: RevisedHistory
 Declare Transformation: PromotedTransformation, DeferredTransformation
 Assert length(History) ≥ 1 % Or there's nothing to banish!
 If length(History)=1 Then Return EmptyHistory
 ⟨SuccessFlag,DeferredTransformation,PromotedTransformation⟩:=
    DeferTransformation(CurProgram,History[1],History[2])
 If SuccessFlag Then
   % We can move transformation to banish to History[2].
   % Pretend we did that, and (eagerly) banish it from there.
   ⟨PartialImplementation,RevisedHistory⟩:=
     BANISH(ApplyTransformation(PromotedTransformation,CurProgram),
            DeferredTransformation+rest(History,3))
   Return ⟨PartialImplementation,PromotedTransformation+RevisedHistory⟩
 Else
   % Transformation we wish to banish is blocked by rightmost neighbor.
   % So banish rightmost neighbor, shortening history, and try again.
   % Safe to banish rightmost neighbor for two reasons:
   % 1) This procedure can be conservative (because the Revise
   %    procedure will work even if Banish throws away everything!
   % 2) The rightmost neighbor depends on transformation we are trying to banish;
   %    if we succeed in banishing it, the rightmost neighbor's preconditions
   %    will not be present, and the rightmost neighbor can't be saved either.
   ⟨PartialImplementation,RevisedHistory⟩:=
     BANISH(ApplyTransformation(History[1],CurProgram),rest(History,2))
     % ignore PartialImplementation
   Assert length(RevisedHistory)<length(History)-1
   Return BANISH(CurProgram,History[1]+RevisedHistory)
 Fi
End BANISH
```

Figure 7.11: *BANISH* procedure

undone, whether it was the essential cause of the backtracking step or not. If the cause of backtracking can be traced to a particular transformation in the derivation history, then that transformation can be *BANISH*ed, thereby possibly preserving the work (transformations) accomplished between its point of application and the end of the derivation history. A mechanism to pinpoint a defective transformation must of course exist; this is similar to the problem of explaining failure in machine learning. We have not actually utilized this idea.

We note that if *DEFER* simply fails whenever the locaters are close, *BANISH* acts almost exactly like a dependency network [Lon78]. It is better than such a dependency net because *DEFER* can swap transformations that directly affect one another, as long as the end result is equivalent. A dependency net can indicate, at best, that two transformations somehow interact. In fact, dependency nets fail for remarkably simple cases. Consider an identity transform $t_I : I(?x) \rightarrow ?x$. One can apply *BANISH* to a derivation history $t_I^{\langle\rangle}(t_I^{\langle\rangle}(I(I(z))))$ to get rid of the first application. A dependency net will suggest that the sequential applications of t_I overlap, and are therefore dependent.

Cost to execute BANISH

The cost to banish a transform initially looks quite high. In this section, we analyze various costs to run *BANISH*. We show that worst case costs are not terribly expensive, and argue that the average costs are quite good.

We measure running time of *BANISH* in terms of the number of swap-attempts (calls to *DEFER*). We pidgeon-hole each swap-attempt by its left-hand argument. Each transformation in a history of length k is swapped only with transformations to its right, of which there are at most $k-1$. Further, a transformation x_i can be swap-attempted with x_{i+1} on its immediate right at most once; after such check, the x_{i+1} transformation is either swapped to left of x_i (where it will not participate further in the *BANISH* process) or x_{i+1} is banished (deleting it from the remaining history, so it can't be swap-attempted with any transformation, let alone x_i, again). So x_1 can participate in at most $k-1$ swap-attempts; x_2 with at most $k-2$, and x_{k-1} with at most 1 swap-attempts. The number of swap-attempts is then at most $\sum_{i=1}^{k-1} k-i = \frac{(k-1)*k}{2}$. Since (except for trivial cases) there is one swap-attempt per call to banish, the cost to banish must be at most $O(k^2)$.

If we have a design space in which every decision depends on every other decision (i.e, highly constrained) the cost of banishment is $O(k-1)$: we attempt to swap each transformation its right-hand neighbor, failing each time, and then the history is truncated. If we have an extremely commutative design space (close to what we expect in practice) then the cost of banishment is also $O(k-1)$: the offending transformation is repeatedly swapped with its right-hand neighbor until it reaches the end of the derivation history, and is then truncated. We speculate that a dependency network would lower the average cost in this case to $O(1)$.

We defined costs in terms of history length and the number of swap-attempts. Each swap-attempt can be expensive in its own right if the transforms are complex. If executing *DEFER* on a particular pair of transformations should exceed a predefined threshold, one can conservatively assume that they do not commute. If this happens often, then the cost to banish will drop as more of the history is lost; one pays the price of losing that potentially preservable history later when the conventional transformation system will have that much more work to do reconstructing a new tail for the derivation history.

7.2.4 Banishing batches of transformations

Sometimes we can simultaneously identify a number of transformations in a derivation history H which we are sure are undesirable. In this case, we can save effort by *banishing in a batch*. The idea is simple: mark each $H[i]$ that is undesirable in H; then scan H from left to right, looking for a marked transformation. For each marked transformation, apply the *BANISH* procedure, with one additional proviso: before attempting a swap, if the righthand transformation is also marked, first banish it. Marks must obviously swap when their corresponding transformations swap. We call this process *BATCHBANISH* (Figure 7.2.4). The savings occur in that no transformation with index i in the batch is bubbled-right into a block of transformations which are dependent on some later to-be-banished transformation with index $j > i$.

We define

$$\mathit{BATCHBANISH} : \mathcal{H} \times (\mathbf{Nat} \rightarrow \mathbf{Boolean}) \rightarrow \mathcal{H}$$

in terms of an auxiliary function *BATCHBANISH0*, paralleling the definition *BANISH*. We construct *BATCHBANISH0* to have the same effect as *PARTITION*. By dropping the tail of the derivation history produced by *BATCHBANISH0*, we obtain *BATCHBANISH*:

$$\mathit{BATCHBANISH0}(H) = \langle H', j \rangle \supset \mathit{BATCHBANISH}(H) = H'[1..j-1]$$

We define

$$BATCHBANISH0 : \mathcal{H} \times (\mathbf{Nat} \rightarrow \mathbf{Boolean}) \rightarrow \mathcal{H} \times \mathbf{Nat}$$

such that $BATCHBANISH0(H, undesirable) = \langle H', j \rangle$ satisfies:

$$\begin{array}{ll}
H' \in HPERMS(H) & \\
\forall i : 1 \leq i < j \supset undesirable_{H'}(i) = false & \\
j \leq length(H') \supset undesirable_{H'}(j) = true & \\
\forall i : j \leq i < length(H') \supset & DEFER(H'[i], H'[i+1]) = \langle false, x_1, x_2 \rangle \vee \\
 & undesirable_{H'}(i+1) = true
\end{array}$$

As with *BANISH0*, the index j is the blocking point of some undesired transformation, with the additional provision that *all* the undesired transformations and their dependents are at or beyond the blocking point. Truncating at the blocking point throws away all the unwanted transformations and dependents. The implementation of *BATCHBANISH* truncates incrementally, like *BANISH*.

The cost to perform *BATCHBANISH* on a derivation history of length k is identical to the cost to perform *BANISH*. Remembering that that we counted swap-attempts for *BANISH*, if we simply treat a swap-attempt on $H[i]$ as also including a check for $undesirable(i+1)$, then the code structure for *BANISH* and *BATCHBANISH* become identical, and thus have identical worst-case running times of $O(k^2)$. Since we expect more than one undesirable transformation to be present during a *BATCHBANISH*, its average costs should be a little higher than *BANISH*ing just a single transformation, but it is clear that one should *BATCHBANISH* rather than *BANISH* when possible. The lower bound on the cost to *BATCHBANISH* is obviously $O(k)$ because the derivation history must be scanned to find marked transformations. If one has a small list of undesirable transformations and uses a dependency net, it may be more efficient to individually *BANISH*.

The relative efficiency of *BATCHBANISH* over individual *BANISH* suggests that banishing transformations should be delayed as long as possible in order for the batch to grow to maximum size, and then applying *BATCHBANISH*. This delaying hueristic accounts for the order in which deltas are processed in Figure 1.13. Functional deltas are processed last because they must be applied to a clean derivation history.

When producing a monolithic derivation history free of *undesireable* transformations, *BATCHBANISH* seems reasonable. Under circumstances in which the elements of a derivation history are enumerated in order, and those elements may indirectly force marking other elements as *undesirable* (see Section 7.4 for an example of this), it may be better to banish lazily. By this we mean deferring application of an *undesirable* transformation as little as possible to reveal a potentially reusable transformation, in

```
Function BATCHBANISH(Program: CurProgram,DerivationHistory: History,
  Returns ⟨Program,DerivationHistory⟩
 % This function delays application of marked (undesirable) transformations
 % as long as possible, chops the history off at the earliest delayed transformation,
 % and returns the revised history.
 % Because we always chop the history off, batchbanishing cannot fail;
 % at worst it returns an empty history.
 % This procedure costs O(length(History)$^2$) to run.
 Declare Program: PartialImplementation, Boolean: SuccessFlag
 Declare DerivationHistory: RevisedHistory
 Declare Transformation: PromotedTransformation, DeferredTransformation
 If length(History)=0 Then Return ⟨CurProgram,EmptyHistory⟩
 If ¬undesirable(History[1]) Then
  ⟨PartialImplementation,RevisedHistory⟩:=
   BATCHBANISH(ApplyTransformation(History[1],CurProgram),rest(History,2))
  Return ⟨PartialImplementation,History[1]+RevisedHistory⟩
 Fi
 % History[1] is undesirable, delay its application
 If length(History)=1 Then Return ⟨CurProgram,EmptyHistory⟩
 If ¬undesirable(History[2]) Then
  ⟨SuccessFlag,DeferredTransformation,PromotedTransformation⟩:=
   DeferTransformation(CurProgram,History[1],History[2])
  If SuccessFlag Then
   % We can move undesirable transformation to History[2].
   % Pretend we did that, and (eagerly) banish it from there.
   undesirable(DeferredTransformation):=true
   ⟨PartialImplementation,RevisedHistory⟩:=
    BATCHBANISH(ApplyTransformation(PromotedTransformation,CurProgram),
     DeferredTransformation+rest(History,3))
   Return ⟨PartialImplementation,PromotedTransformation+RevisedHistory⟩
  Else undesirable(History[2]):=true
  Fi
 Fi
 % Transformation we wish to banish is blocked by rightmost, undesirable, neighbor.
 % So banish rightmost neighbor, shortening history, and try again.
 ⟨PartialImplementation,RevisedHistory⟩:=
  BATCHBANISH(ApplyTransformation(History[1],CurProgram),rest(History,2))
 % ignore PartialImplementation
 Assert length(RevisedHistory)<length(History)-1
 Return BATCHBANISH(CurProgram,History[1]+RevisedHistory)
End BATCHBANISH
```

Figure 7.12: Batch Banish procedure

an effort to allow the supply of *undesirable* transformations to grow as large as possible in the tail of the derivation history before processing the tail. Procedures to accomplish this are shown in Figure 7.2.4.

7.3 Integration of Technology Deltas $\Delta_{\mathcal{C}}$

We now have enough mechanisms defined to integrate $\Delta_{\mathcal{C}}$s into a derivation history. Our original approximation for reusing a derivation history was to assume that all transformations were reusable unless easily shown otherwise. Technology deltas directly provide information to the effect that certain transforms, and therefore their derivative transformations, are no longer valid. Remembering that a $\delta_{\mathcal{C}} = \langle \Delta_{\ominus}, \Delta_{\oplus} \rangle$, we see a direct identification of transforms which are no longer legitimate to use: $\Delta_{\ominus}$. The procedure is straightforward:

1. Mark transformation $H[i] = t^{\ell}$ as *undesirable* in the existing derivation history if $\langle i, t \rangle \in \Delta_{\ominus}$.
2. Apply *BATCHBANISH* to remove the undesirable transformations and any dependencies thereof, producing a truncated history H_{saved} as well as $s_{saved} = \Pi(H_{saved}, s_0^H)$.
3. Update the set of usable transforms by computing $C'_{library} = \delta_{\mathcal{C}}(C_{library})$
4. Restart the transformation system at state s_{saved} with H'. This provides an opportunity to use the new transforms represented by $\Delta_{\oplus}$.
5. Output resulting derivation history H' and f^H.

Restarting the transformation system with H_{saved} allows it to backtrack (perhaps using *BANISH* or *BATCHBANISH*) and revise other parts of H_{saved} if needed. Such backtracking may be required as our scheme for integrating $\delta_{\mathcal{C}}$ is conservative; we only remove transformations which are obviously bad. None of the information provided by a $\delta_{\mathcal{C}}$ can assure us that the program specification is achievable via H_{saved}.

```
Function $ENUMERATEHISTORY()$
  returns boolean, transformation
  % Produces next derivation history element on each call
  % Lazily banishes undesirable transformations as they are encountered.
  declare global DerivationHistory: $H$, integer: $next$, program: NextProgram
  $next := next + 1$ % advance history scan pointer on each call
  if $next > length(H)$ then return $\langle false, dummy \rangle$
  if $\neg undesirable(H[next])$ then
    NextProgram:=ApplyTransformation($H[next]$,NextProgram)
    return $\langle true, H[next] \rangle$
  else
    $H := H[1..next-1] + BANISHLAZY($NextProgram,$rest(H, next))$
  if $next > length(H)$ then return $\langle false, dummy \rangle$
  NextProgram:=ApplyTransformation($H[next]$,NextProgram)
  return $\langle true, H[next] \rangle$
end $ENUMERATEHISTORY$


Function $BANISHLAZY($Program:CurProgram,DerivationHistory:$H)$
  returns DerivationHistory
  % Returns $H' : length(H') = 0$ or $undesirable(H'[1]) = false$
  declare program: NextProgram
  assert $length(H) > 0$ and $undesirable(H[1]) = true$
  if $length(H) = 1$ then return $emptyhistory$
  % Try to defer $H[1]$ until after $H[2]$
  NextProgram=ApplyTransformation($H[1]$,CurProgram)
  if $undesirable(H[2]) = false$
  then $H' := rest(H, 2)$
  else $H' := BANISHLAZY($NextProgram,$rest(H, 2))$ fi
  loop
    if $length(H') = 0$ then return $emptyhistory$
    assert $undesirable(H'[1]) = false$
    $\langle successflag, x'_1, x'_2 \rangle = DEFER($CurProgram,$H[1], H'[1])$
    $undesirable(H'[1]) := true$ % mark $H'[1]$ as (transitively) undesirable
    if $successflag$ then return $x'_1 + x'_2 + rest(H', 2)$
    $H' := BANISHLAZY($NextProgram,$H')$
  endloop
end $BANISHLAZY$
```

Figure 7.13: Banish Lazily procedure

The following process is germane:

```
**Procedure** Integrate$\Delta_{\mathcal{C}}$(StartState:State, DerivationHistory:History, $\Delta_{\mathcal{C}} : \delta$)
  **Returns** $\langle$State,History,State$\rangle$
  **Declare** RevisedHistory,AdditionalHistory: History, EndState:State
  **for** $i := 1$ **to** $length(H)$
    **if** $H^{\mathcal{T}}[i] \in \delta.\Delta_{\ominus}$
    **then** $undesirable(H[i]) = true$
  **endfor**
  $\langle$RevisedHistory,EndState$\rangle$:=BatchBanish(StartState,DerivationHistory)
  $C_{library}$:=$\delta(C_{library})$
  $\langle$RevisedHistory,Implementation$\rangle$:=ImplementContinue(Endstate,RevisedHistory)
  **Return** $\langle$StartState,RevisedHistory,Implementation$\rangle$
**End** Integrate$\Delta_{\mathcal{C}}$
```

In practice, we delay restarting the transformation system until we have also adjusted the derivation history to account for the other deltas present in a composite delta. This allows use of a single pass of as *BATCHBANISH* to remove all of the undesirable transformations.

7.4 Integration of Functional Deltas Δ_f

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[4]. When an existing transformation interferes with the desired change, we can simply banish the offending transformation. When a transformation does not interfere, we *preserve* that transformation, i.e., save it for use in the revised derivation history. We scan the original derivation history from beginning to end, checking the delta for interference with each transformation. When such a checking process is complete, the remaining derivation history is compatible with the desired delta; all we need to do is apply the delta and finish the implementation.

What we will 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 start with a functional delta δ_0, a functional specification f_0, and a derivation history H. We want to produce a

DEFINITION 7.3: *Ladder.* A triple $\langle H_{reusable}, H_{revised}, H_\delta \rangle$ with the property:

$$\forall i \leq length(H_{reusable}) : \Pi(H_{revised}[1..i])(H_\delta[1](f_0)) = H_\delta[i+1](\Pi(H_{reusable}[1..i])(f_0))$$

$\square$

$H_\delta[1]$ will contain δ_0, the given δ_f. We call this a ladder because of the resemblance of diagrams of this object to a ladder, with $H_\delta[i]$ forming the rungs and $H_{reusable}$ and $H_{revised}$ forming the left and right sides of the ladder, respectively (Figure 7.14). The ladder component $H_{reusable}$ must be a prefix of a member of $HPERMS(H)$. The dashed arrows shown at the end of $H_{reusable}$ are the transformations banished from H because of their conflict with the effect of δ_f; $H_{reusable}$ plus the dashed arrows ($H_{rearranged}$) is a member of $HPERMS(H)$.

If we can construct a ladder, then $H_{revised}$ is a derivation history for for $\delta_f(s_0)$. If all the members of $H^{\mathcal{T}}_{reusable}$ are property-preserving transforms, then all the members of $H_{revised}$ are also, and so the state $s_{end} = \Pi(H_{revised})(\delta_f(s_0)))$ is a correct partial implementation of $\delta_f(s_0)$. We can pass this derivation history plus the state s_{end} to the transformation system for completion. Thus integrating a Δ_f can be accomplished

[4]Should a software engineer wish to insert a property-preserving transform in the middle of an existing derivation history, the Δ_f integration technique can be used without significant change.

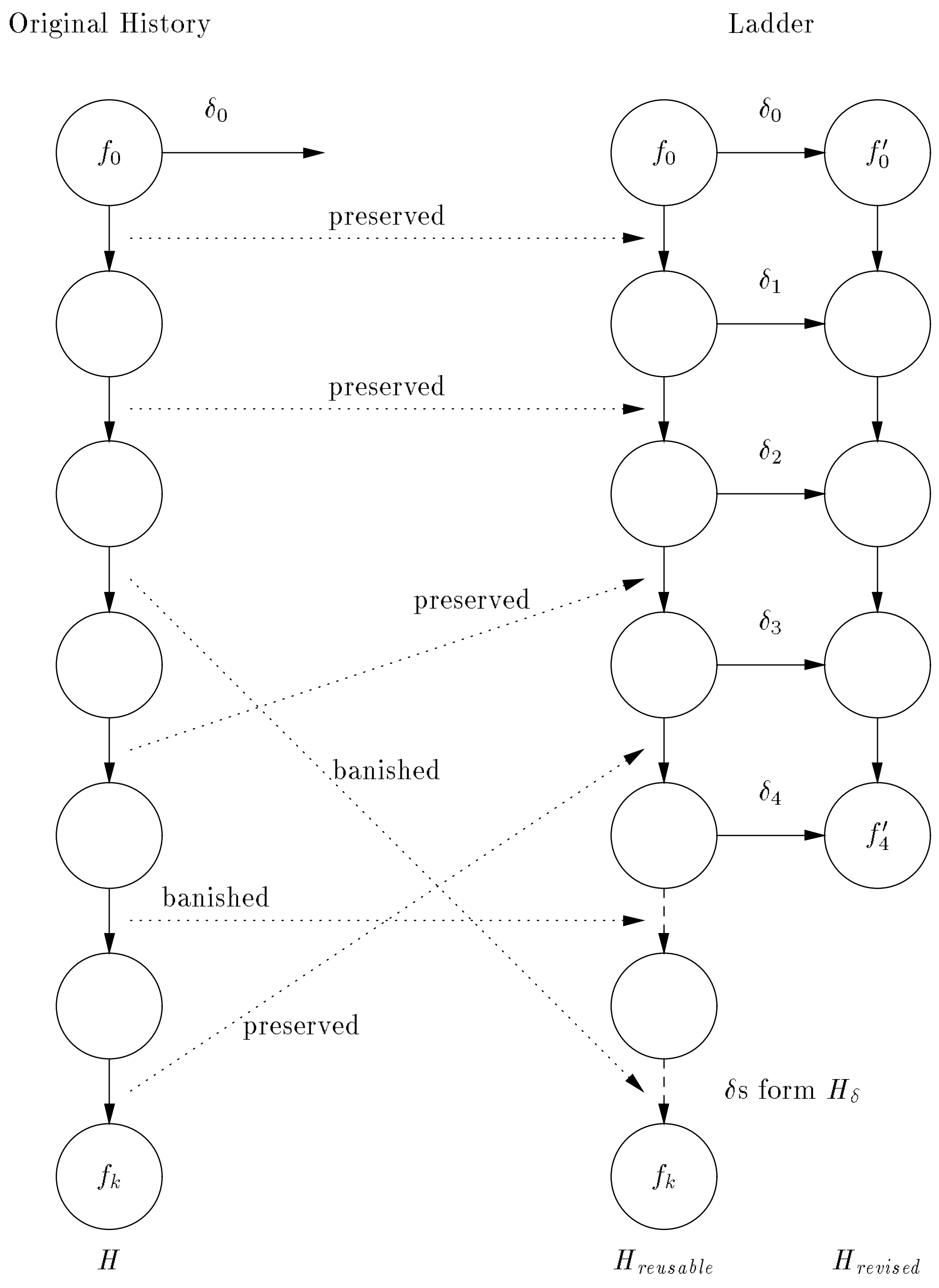


Figure 7.14: Producing a ladder from a Δ_f

by building a ladder. In practice, we don't actually build the ladder as a monolith; it serves as a conceptual device. Production of $H_{revised}$ is sufficient.

We can form the ladder incrementally by an iterative procedure which scans the original derivation history H. At each step, a δ_i derived from previous steps causes either the transformation $H[i]$ to be banished, or to be preserved by forming a new rung of ladder. Failure to preserve transformation $H[i]$ is the signal that $H[i]$ must be banished.

Let us consider how a rung is formed.

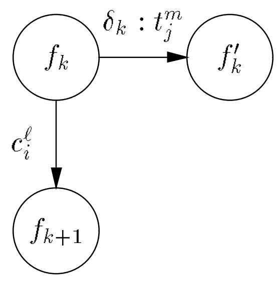
7.4.1 Preserving single transformations

The starting condition when generating the $k+1$th rung shown in Figure 7.15A. To be successful, we must actually find *two* transformations: one to output to $H_{revised}$, and another to serve as δ_{k+1} for the next step. Fig 7.15B shows a plethora of possibilities. How can we choose such a pair? We note that $\delta_k = t_j^m$ for some arbitrary transform t_j with locater m.

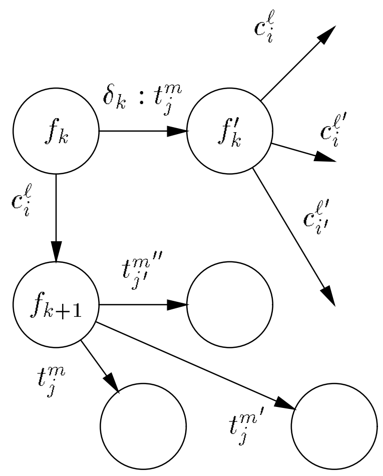
The first constraint is that the transform $H^{\mathcal{T}}_{revised}[k]$ applied to $\delta_k(f_k)$ must generally be a property-preserving transform; after all, it will form part of a new derivation history and therefore must act as though the transformation system generated it. If we can find $t_{j'}, \ell', m'$ such that $t_{j'}^{m'}(c_i^{\ell}(f_k)) = c_i^{\ell'}(t_j^m(f_k) = f'_{k+1}$ (see Fig 7.15C), then the new functionality desired, $f'_k = \delta_k(f_k)$, is preserved by application of $c_i^{\ell'}$, because c_i is, by definition, a property-preserving transform. We can then safely reuse c_i in a new derivation history for $\delta_k(f_k)$. Because a derivation history may contain non-property-preserving transforms, we relax the requirement to $H^{\mathcal{T}}_{reusable}[k] \in C_i \supset H^{\mathcal{T}}_{revised}[k] \in C_i$, usually for $C_i = C_{implicit}$, chosen by the transformation system invariant.

A second constraint on $H^{\mathcal{T}}_{revised}[k]$ is that it ideally should be $t = H^{\mathcal{T}}_{reusable}[k]$; this constraint comes not from anything in the derivation history, but from a desire to be able to continue using the justification from the design history for t; we will discuss this further in Chapter 8. So we need only pick a new locater.

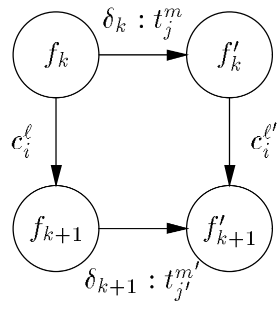
The last constraint comes from the ladder itself. If $H_{revised}$ is to truly be analogous to $H_{reusable}$, then there must be a constructive analogy between each parallel state generated by the derivation histories. Consequently, we require that:



A: Before preserve step: we possess δ_k and c_i^ℓ



B: Possible Choices for δ_{k+1} and t'_i



C: Best Choice for δ_{k+1}

Figure 7.15: Preserving a transformation

$$H_{revised}[k](\delta_k(f_k)) = \delta_{k+1}(H_{reusable}[k](f_k))$$

We interpret this as "applying an implementing (technology) transform to the changed state is the same as implementing the state, and then applying a change to it."

We define a function, *PRESERVE*, to form a rung and push a delta through it:

$$PRESERVE : powerset(\mathcal{T}) \times \mathcal{S} \times \mathcal{X} \times \mathcal{X} \rightarrow \textbf{Boolean} \times \mathcal{X} \times \mathcal{X}$$

such that

$$PRESERVE(C, s, t_i^\ell, t_j^m) = \langle b, t_{i'}^{\ell'}, t_{j'}^{m'} \rangle$$

has the properties:

$$\begin{array}{l} b = true \supset t_{j'}^{m'}(t_i^\ell(s)) = t_{i'}^{\ell'}(t_j^m(s)) \\ t_i \in C \supset t_{i'} \in C \end{array}$$

Failure to *PRESERVE* a transformation is signaled by returning $b = false$. We banish transformations that cannot be preserved. Since banishing also truncates transformations dependent on the unpreservable one, we will not have to deal with complications. We do not hunt for substitutions or long chains of other transformations, because we want to preserve transforms that can be traced back to *APPLY* steps in methods; more on this in Chapter 8.

Implementing *PRESERVE*

Similar to the discussion about implementing *SWAP* in Section 7.2.2, we do not provide precise details on how to implement *PRESERVE*, because they depend on the transformation system, its representation, and the sets of property-preserving transforms. Much of the discussion in that section applies. In particular the following points are still relevant:

- A theorem prover is necessary in general
- Conservative cutoff can conservatively signal failure of *PRESERVE*
- Having proposed a pair of resulting transformations, it is often sufficient for validation purposes to compare compositions of the transformations rather than comparing applications of the proposed transformations to states, i.e., simply to check that: $t_{j'}^{m'} \circ t_i^\ell = t_i^{\ell'} \circ t_j^m$
- Locaters which are "far apart" are common and provide a special case which is easily implemented, by simply returning the argument transformations as the results.

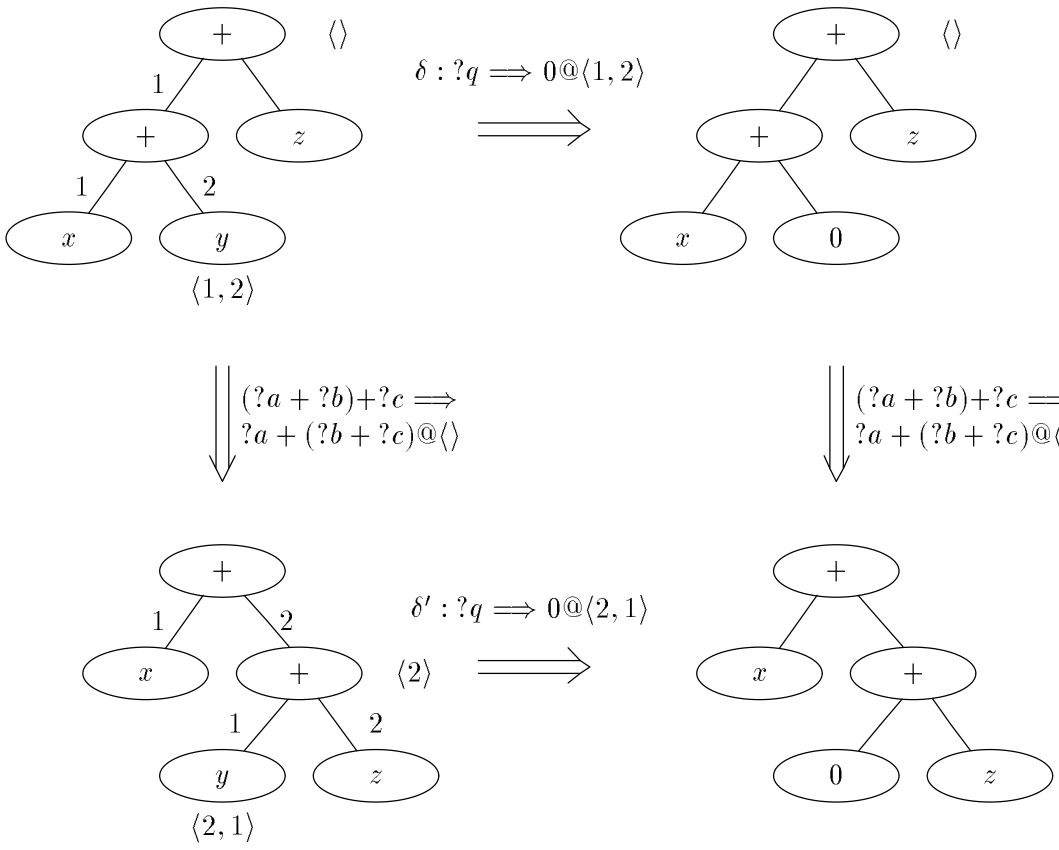
It may be that there is more than one way to produce resulting transformations satisfying *PRESERVE*. Carrying t_i through intact (i.e., $t_{i'} = t_i$) is one way to eliminate multiple results, motivated by possible reuse of design history justifications. Similarly, one wants to keep $t_{m'}$ as small as possible; gratuitous expansions of the delta simply make computations of following ladder rungs slower. We conjecture that implementing *PRESERVE* as a pushout (if it exists) in the underlying category of transformations is probably best, since that places the fewest constraints on f'_{k+1}. If f'_{k+1} is a pushout, it implies that $p_{implicit}(f''_{k+1}) \succeq_{implicit} p_{implicit}(f'_{k+1})$ for any f''_{k+1} which is not the pushout. Fewer constraints on intermediate states mean fewer commitments to carry through to an implementation, and therefore probably a shorter final derivation history.

Statistically, the revised δ is equal to the original δ. Sometimes $t_m \neq t_{m'}$, i.e., the delta transform changes. This is no cause for concern; the System Analyst will never be confused by it because he will never see such intermediate deltas, and the Software Engineer will see a delta which is relevant to the program f_{k+1} he is inspecting.

We must still validate that each preserved transformation still achieves some desired effect, but that is a topic for Chapter 8. For this chapter, mere generation is sufficient.

We provide some examples of the *PRESERVE* step with tree transforms. We do not show the common case, where the locaters of the transformation to be preserved and the delta do not overlap. When the locaters select nearby regions, as with *SWAP* implemented for trees, an analysis of how the overlapping transformations rearrange their subtrees can lead to simple proposals for how to revise the locaters. Figure 7.16 shows a case in which preserving the transformation causes the application point of the delta to be moved; the delta's locater is revised accordingly. Figure 7.17 shows the opposite case; the delta moves the point of application of the transformation to be preserved, causing revision of the locater of the preserved transform.

Building an efficient implementation of a specification is fundamentally accomplished by spreading information. This effect can be seen in Figure 7.18, in which the delta is spread out by application of the transformation to be preserved. This suggests that the deltas forming ladder rungs are likely to grow monotonically in size as we move down the derivation history, perhaps to the point where the delta can become significant in size relative to f_k. This can be handled, if necessary, by the simple device of forcing *PRESERVE* to fail whenever the delta becomes inconveniently large. Considering that the transformations to be preserved are likely to stay small, we think that manufacturing ladder rungs even with large deltas should be cheaper than trying to regenerate those transformations.



$\langle\rangle$
$\delta : ?q \Longrightarrow 0@\langle 1,2\rangle$
$\langle 1,2\rangle$
$(?a + ?b)+?c \Longrightarrow ?a + (?b + ?c)@\langle\rangle$
$\delta' : ?q \Longrightarrow 0@\langle 2,1\rangle$
$\langle 2\rangle$
$\langle 2,1\rangle$

Figure 7.16: Preserving transformation intact by revising delta locater

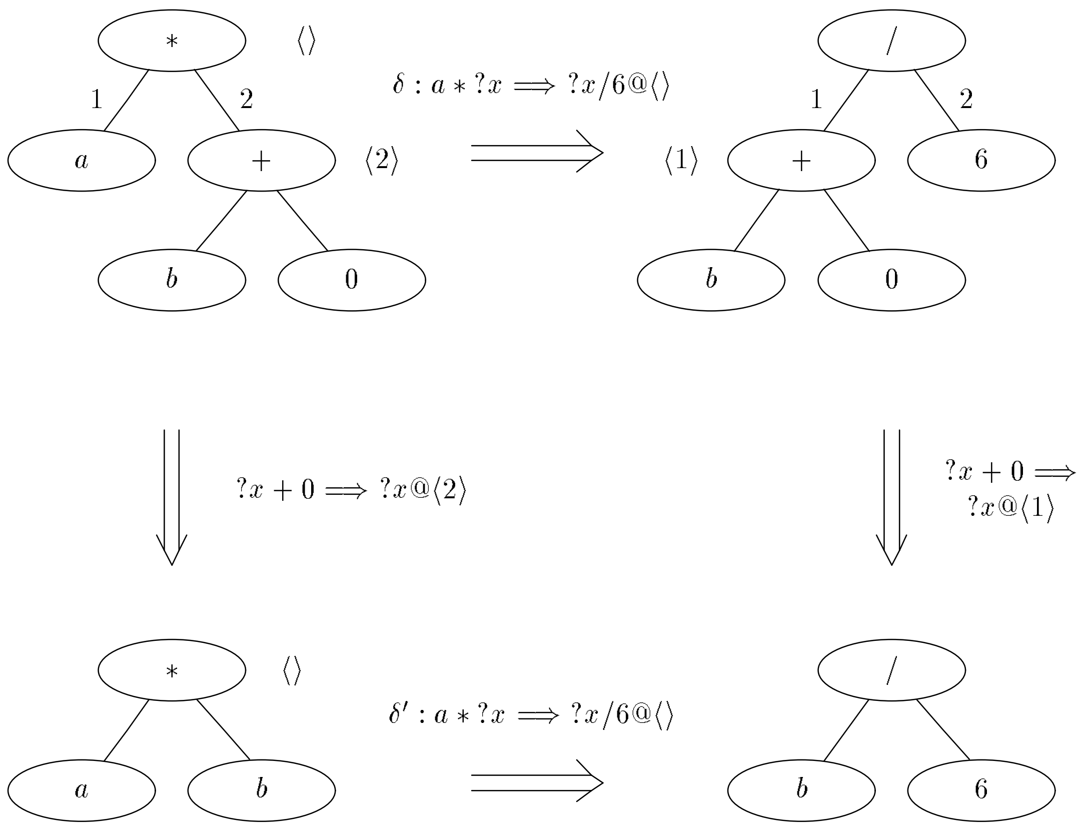

$*$
$\langle\rangle$
1
2
a
$+$
$\langle 2\rangle$
b
0
$\delta : a * ?x \Longrightarrow ?x/6@\langle\rangle$
$/$
1
2
$\langle 1\rangle$
$+$
6
b
0
$?x + 0 \Longrightarrow ?x@\langle 2\rangle$
$?x + 0 \Longrightarrow ?x@\langle 1\rangle$
$*$
$\langle\rangle$
a
b
$\delta' : a * ?x \Longrightarrow ?x/6@\langle\rangle$
$/$
b
6

Figure 7.17: Preserving a transform by revising its locater

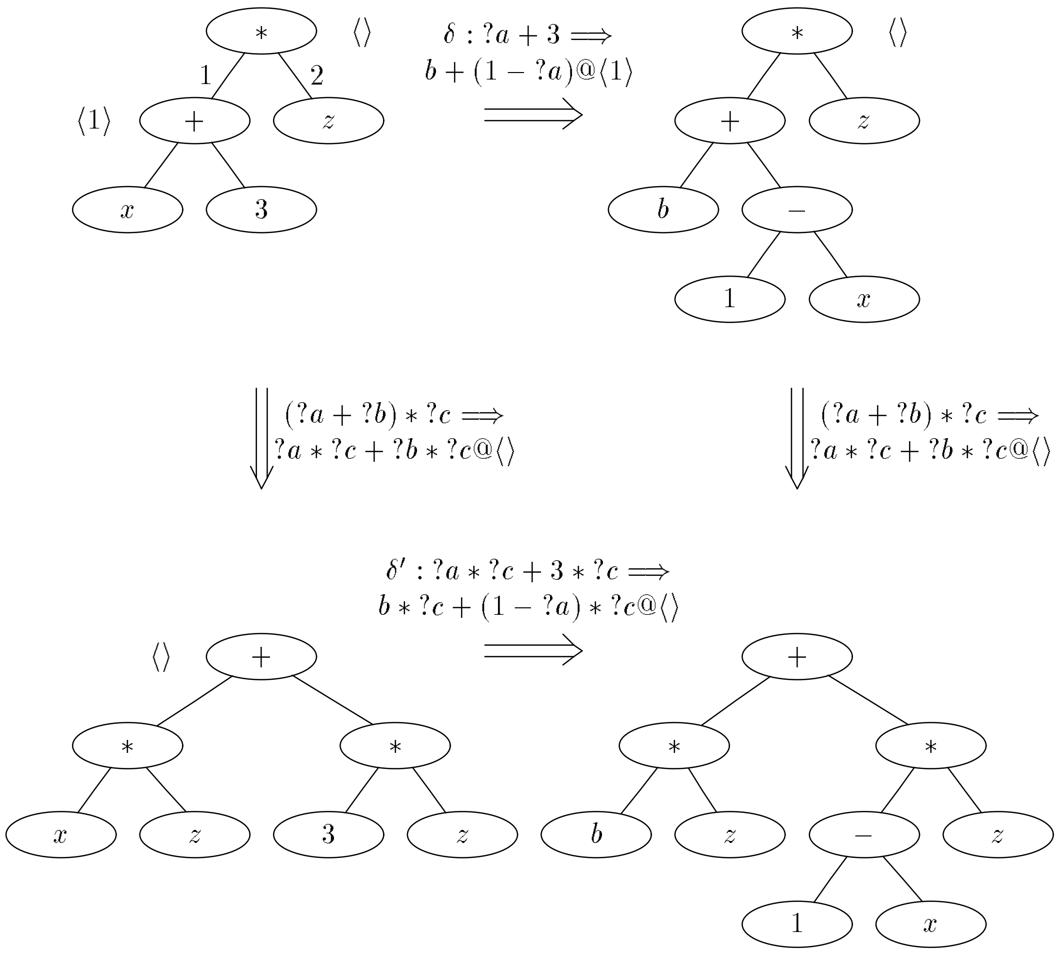

$\langle\rangle$
$\langle 1\rangle$
1
2
$\delta : ?a + 3 \Longrightarrow$
$b + (1 - ?a)@\langle 1\rangle$
$(?a + ?b) * ?c \Longrightarrow$
$?a * ?c + ?b * ?c@\langle\rangle$
$\delta' : ?a * ?c + 3 * ?c \Longrightarrow$
$b * ?c + (1 - ?a) * ?c@\langle\rangle$

Figure 7.18: Growth of delta by information spreading

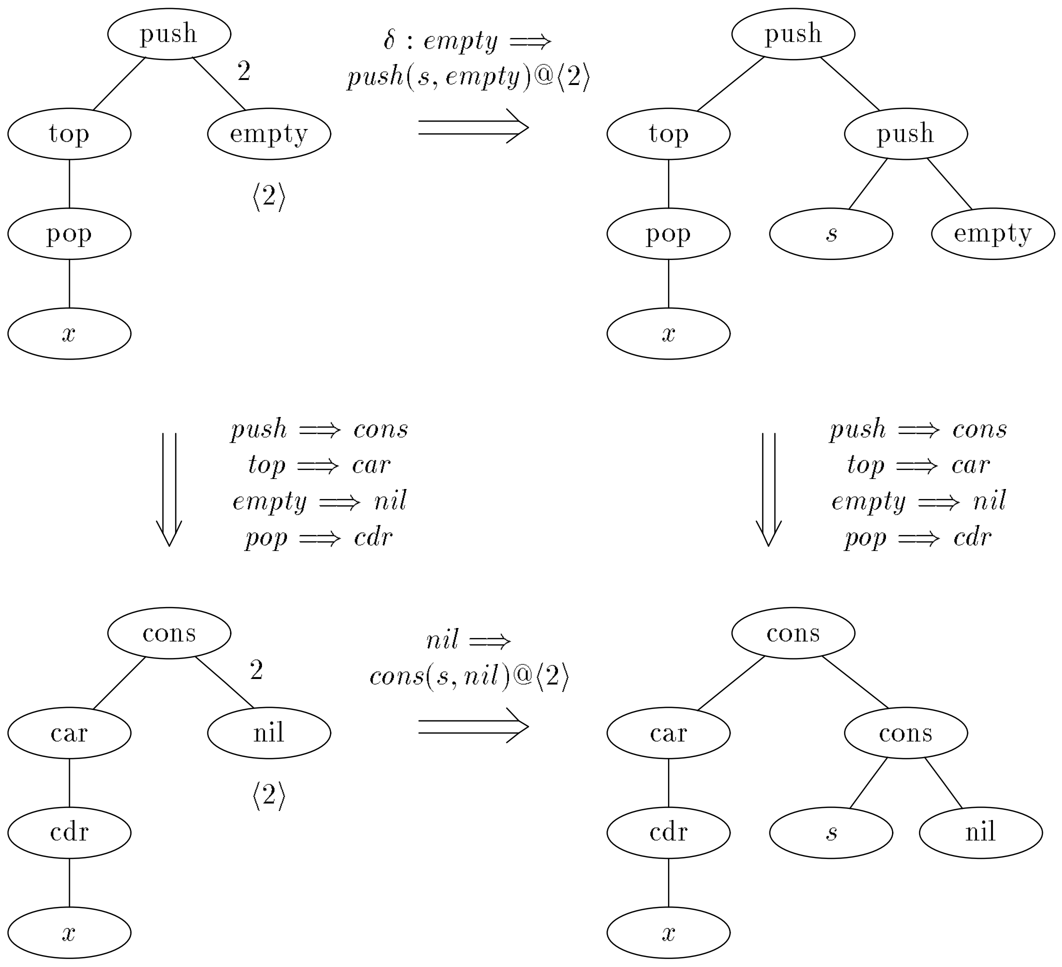


Figure 7.19: Preserving a refinement translates the delta

PRESERVE is not limited to just tree transformations. In Figure 7.19, we show an example of preserving a theory morphism. The delta is simply mapped from the originating domain to the target domain. We remark that formally justifying such a step requires a theory about rewriting rewrites themselves. Our definition of *PRESERVE* sidesteps this requirement.

It may be very difficult to preserve the application of complex transforms such as LR-parser generators in the face of deltas. In the case of a parser generator, it is relatively easy to re-run, so it may actually be reasonable to simply give up and *BANISH* such transformations. One can also consider handling commonly occurring special cases (such as token renaming, addition of terminals to existing rules, etc.).

7.4.2 Procedure for integrating Δ_f

We have seen a conceptual overview of Δ_f integration as ladder construction. We have seen how to form ladder rungs by applying the *PRESERVE* operation. We have hinted that *BANISH* can be used on transformations that cannot be preserved in the face of a delta. Now it is time to assemble the pieces into a Δ_f integration procedure: $\mathit{INTEGRATE}_f$ (Figure 7.20).

This procedure accepts the initial specification f_0, a derivation history H leading to a current implementation and a maintenance delta δ_f applied to f_0; it produces $H_{revised}$, the portion we could save, and an implementation of $\Pi(H_{revised}(f_0))$. $\mathit{INTEGRATE}_f$ operates recursively by *PRESERVE*ing the first transformation in the history and revising the rest of the history according to the resulting delta. If it cannot preserve a transformation, then that transformation is banished, and it revises the resulting history according to the current delta. The final history, $H_{revised}$ is built up from the bottom while unwinding the recursion. The ladder is never built as an entity.

Using $\mathit{INTEGRATE}_f$, it is possible to *insert* a Δ_f in the *middle* of a derivation history. This is accomplished by splitting the history at the point of insertion, revising the suffix of the history according to the desired delta, and combining the unchanged history prefix with the revised suffix. Given a functional specification f_0 with history H, we can insert δ_f between $H[j-1]$ and $H[j]$ by computing:

$$\langle \mathit{implemented}, \mathit{program}, H'_{rest}\rangle = \mathit{INTEGRATE}_f(\Pi(H[1..j-1])(f_0, \delta_f, \mathit{rest}(H, j))$$

and replacing the derivation history with $H[1..j-1] + \delta_f + H'_{rest}$. We name this process $\mathit{INTEGRATEMIDDLE}_f$ to remind us that the revision takes place at some named index point. We will find $\mathit{INTEGRATEMIDDLE}_f$ especially convenient when attempting to repair a design history in Chapter 8 by inserting property-preserving transforms.

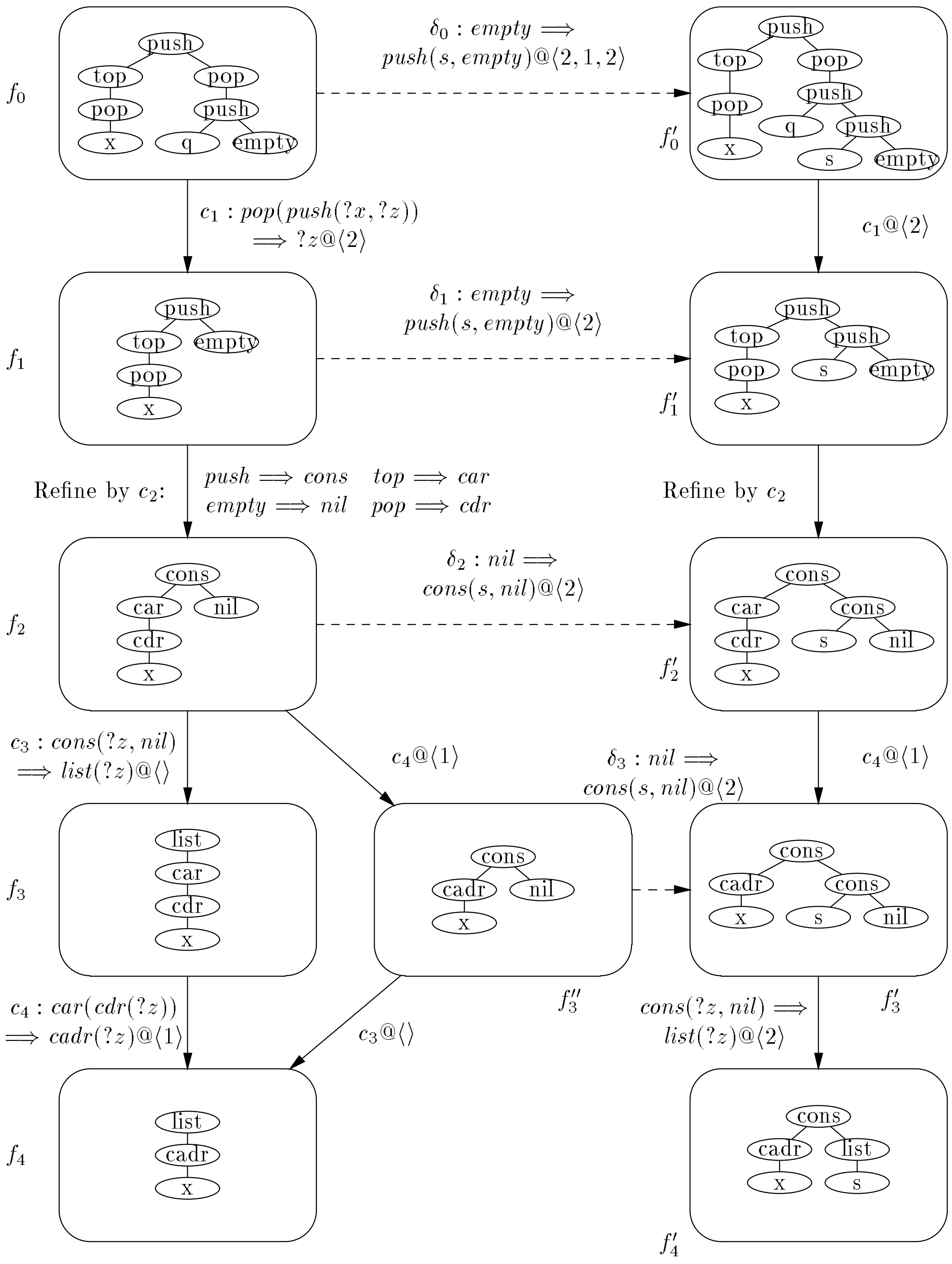
7.4.3 Δ_f integration: An Example

In this section, we provide a concrete example (Figure 7.21) of reusing a derivation history by integrating a functional delta. This is one of the key examples in this thesis. The example follows the conceptual ladder-construction process, rather than the procedural implementation, but the effect is identical.

For the sake of an example, we have chosen a problem domain consisting of stack-computations. An algebraic specification of the problem domain can be found in Appendix C, but the key ideas are stacks-as-values, and operations that push and

```
Function $INTEGRATE_f$(Program: CurProgram, Transformation: Delta,
    DerivationHistory: History)
 Returns ⟨Boolean,Program,DerivationHistory⟩
 % Constructs a new implementation and history for the
 % program defined by ApplyTransformation(Delta,CurProgram) ...
 % by revising the DerivationHistory of CurProgram to integrate Delta
 Declare Program: Implementation, PartialImplementation
 Declare DerivationHistory: RevisedHistory, Boolean: SuccessFlag
 Declare Transformation: PreservedTransformation, RevisedDelta
 If length(History)>0
   And Not ConventionalTransformationalImplementation
 Then
   % Try to Reuse history to derive new implementation
   ⟨SuccessFlag,PreservedTransformation,RevisedDelta⟩:=
       PreserveTransformation(CurProgram,History[1],Delta)
   If SuccessFlag Then
     % We were able to preserve the original transformation
     ⟨SuccessFlag,Implementation,RevisedHistory⟩:=
         $INTEGRATE_f$(ApplyTransformation(History[1],CurProgram),
             RevisedDelta,rest(History,2)) % integrate the rest!
     If SuccessFlag Then
       % Success at revising history and obtaining an implementation
       Return ⟨True,Implementation,
           PreservedTransformation+RevisedHistory⟩
     Else
       % Not able to revise history and obtain an implementation.
       % Perhaps we can get an implementation from CurProgram.
       % If not, it is hopeless from here.
       Return Implement(ApplyTransformation(Delta,CurProgram))
     Fi
   Else
     % Can't preserve History[1] because of some inability to resolve conflict...
     % with the desired Delta so make History[1] stop bothering us.
     ⟨PartialImplementation,RevisedHistory⟩:=
       BANISH(CurProgram,History)
       % ignore PartialImplementation
     Return $INTEGRATE_f$(CurProgram,RevisedHistory,Delta)
     % Won't loop: BANISH chops off offending transformation
   Fi
 Else
   % No more revision possible, nothing left to revise.
   Return Implement(ApplyTransformation(Delta,CurProgram))
 Fi
End $INTEGRATE_f$
```

Figure 7.20: Procedure to Integrate Δ_f into derivation history



f_0
$\delta_0 : empty \Longrightarrow push(s, empty)@\langle 2,1,2\rangle$
f'_0
$c_1 : pop(push(?x, ?z)) \Longrightarrow ?z@\langle 2\rangle$
$c_1@\langle 2\rangle$
f_1
$\delta_1 : empty \Longrightarrow push(s, empty)@\langle 2\rangle$
f'_1
Refine by c_2:
$push \Longrightarrow cons$ $top \Longrightarrow car$
$empty \Longrightarrow nil$ $pop \Longrightarrow cdr$
Refine by c_2
f_2
$\delta_2 : nil \Longrightarrow cons(s, nil)@\langle 2\rangle$
f'_2
$c_3 : cons(?z, nil) \Longrightarrow list(?z)@\langle\rangle$
$c_4@\langle 1\rangle$
$\delta_3 : nil \Longrightarrow cons(s, nil)@\langle 2\rangle$
$c_4@\langle 1\rangle$
f_3
f''_3
f'_3
$c_4 : car(cdr(?z)) \Longrightarrow cadr(?z)@\langle 1\rangle$
$c_3@\langle\rangle$
$cons(?z, nil) \Longrightarrow list(?z)@\langle 2\rangle$
f_4
f'_4

Figure 7.21: Δ_f-integration (replay) using a derivation history

pop scalars onto stacks, producing new stacks. The example is a little contrived in order to make it both small enough to fit on one page as well as a little bit interesting.

A particular expression from the stack domain is provided as the base specification, shown in 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_{rest} implicit, but we assume it includes $p_{language}(f) \succeq \mathit{LISP}$ and some unstated computational efficiency goal.

The leftmost column of the diagram shows an implementation process. 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, of mixed types of transformations. Transformations c_1, c_3 and c_4 are tree transformations with path locaters; c_2 is a theory morphism ("refinement") mapping stack expressions into *LISP*. Both types of transformations are described as examples in Section 3.1.7. 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 are obtained from the algebraic specification of the domains. The implementation process follows that of the Draco system [Nei84a] in its style of repeatedly performing optimize-within-domain then refine-to-new-domain.

The rightmost column is similarly an implementation process, 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 Figure 7.3; 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 (which was presumably difficult for the transformation system to generate because of the control problem)
- the functional delta $\delta_0 \equiv \mathit{empty} \Longrightarrow \mathit{push}(s, \mathit{empty})@\langle 2, 1, 2\rangle$

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. Note the contrast of this situation vis-a-vis conventional maintenance, where all we have is f_4 and some knowledge that it is wrong!

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

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

Step 2. Computing $\mathit{PRESERVE}(f_1, c_2, t_0@\langle 2\rangle)$ produces the result $\langle \mathit{true}, c_2, t_1@\langle 2\rangle\rangle$ with $t_1 = \mathit{nil} \Longrightarrow \mathit{cons}(s, \mathit{nil})$, essentially by applying the refinement to both parts of t_0. We have again avoided use of the transformation system. 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 $\mathit{PRESERVE}(f_2, c_3@\langle\rangle, t_1@\langle 2\rangle)$, which fails (returns *false*) 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 $c_3@\langle\rangle$. We therefore $\mathit{BANISH}(f_2, [c_3@\langle\rangle, c_4@\langle 1\rangle])$, which produces the revised history $[c4@\langle 1\rangle]$. This effectively (*BANISH0*) demotes c_3 below c_4. This demotion is shown in the sub-derivation history which branches from state f_2 and continues down the middle of the page. In practice, *BANISH* also chops off the now-trailing transformation $c_3@\langle\rangle$ because it is already known to interfere with the delta. We show the trailing c_3 so that the reader can see the equivalence of the derivation history pair determined by commuting transformations it contains.

Having *BANISH*ed c_3, and promoted c_4, we compute $\mathit{PRESERVE}(f_2, c4@\langle 1\rangle, t_1@\langle 2\rangle)$, producing the result $\langle \mathit{true}, c4@\langle 1\rangle, t_1@\langle 2\rangle\rangle$, again without resorting to use of the transformation system. Append $t_1@\langle 2\rangle$ to the new derivation history. We compute $f_3'' = c4@\langle 1\rangle(f_2)$, and discard f_2.

Step 4. Either $c_3@\langle\rangle$ was truncated by *BANISH*, or we attempt to $\mathit{PRESERVE}(f_3, c_3@\langle\rangle, t_1@\langle 2\rangle)$ which fails again. In either case, we find that we can make no further progress towards an implementation using the old derivation history information; we consequently throw away any remaining old derivation history at this point. 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 generates 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 3 of the 4 transformations from the original derivation history. All the mechanisms, *DEFER*, *BANISH*, and *PRESERVE* are required to carry this out.

A prototype system that takes a derivation history and a functional delta, using conditional tree transformations and theory morphisms, was constructed in Common Lisp. The system closely matches the structure of the code in Appendix B. It generated this example, as well as a number of similar examples, up to the point of generating a new tail (stops at f'_3) for the revised derivation history. This particular example takes 50 mS. of CPU for 3 retained transformations, or about 17 mS. each. This is clearly a big win over 250 mS. average per generated transformation typical for Draco. While we realize that the small scale of the example prevents any strong conclusion from being drawn, it is nonetheless very encouraging. Larger examples were not run because of the difficulty in obtaining a valid derivation history; no transformation system available to us produces these in a usable form.

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 in any form). 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 to do? It is very 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 does not come as any great surprise that maintenance in conventional software engineering environments is a hard task.

7.5 Intertwining of Implementation with Specification

Our model of transformational maintenance suggests that the transformational implementation process is run as an atomic transaction, and that maintenance deltas are generated between such transactions. London [LF82] discovered situations in which transformational implementation of a **Gist** specification unexpectedly required change to the environmental portion of the specification; this corresponds to feedback from a partial implementation to the specification while running the transformation system. Swartout [Swa82] dismisses conversion of specifications to implementation

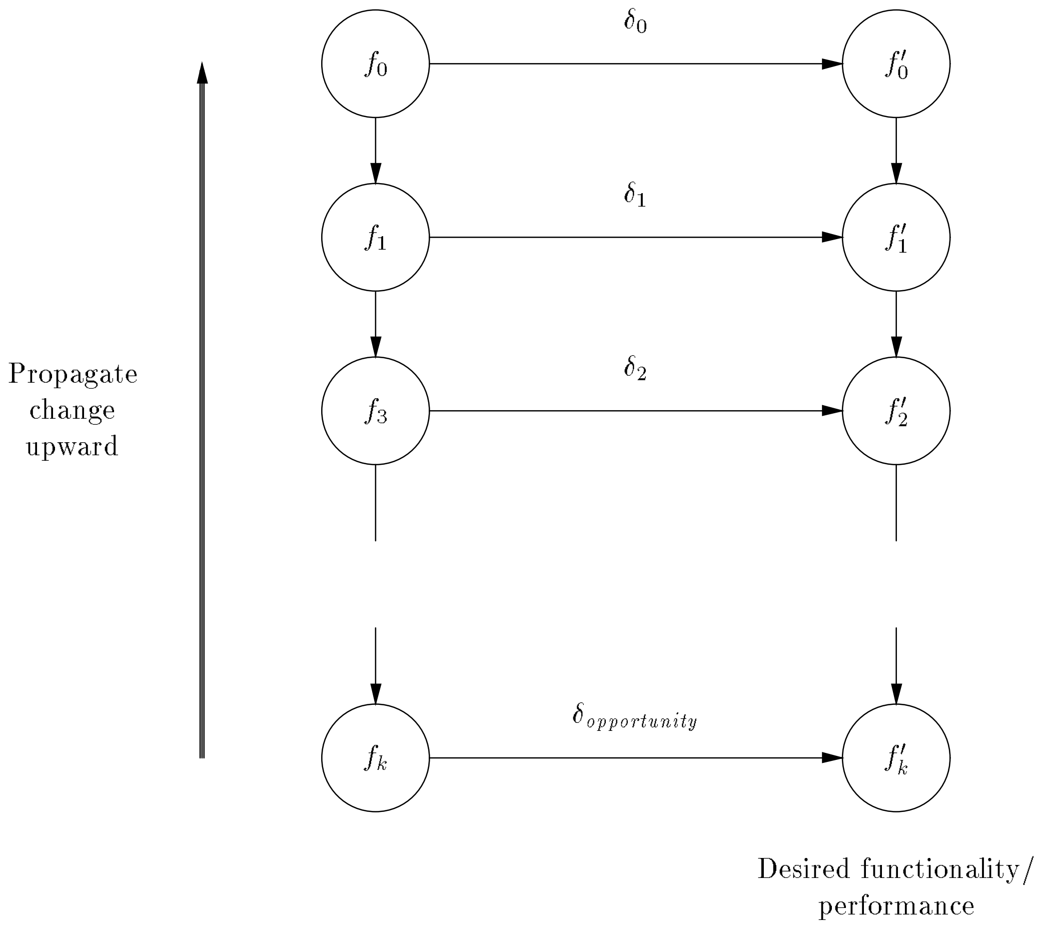


Figure 7.22: Intertwining of Specification and Implementation

with no change of specification as unrealistic; they argue that the realities of implementation will force changes onto the specification.

Given this experience, in a practical Design Maintenance System, we expect that maintenance deltas can arise *during* the transformational implementation process, that we would like to apply to some state in the middle of the design space. This can occur when, part way through an implementation, there is a need to achieve a slightly different functional specification than originally intended, in order to accommodate or take advantage of newly discovered aspects of the environment or implementation technologies. In fact, one might produce a maintenance delta for *any* aspect of the implementation for which it might appear convenient.

One way in which such deltas might arise occurs when an extremely desirable c_i fails to apply at some step s_k; the desirability of the c_i will be due to some performance goal which is difficult to satisfy. If a $\delta_{opportunity}$ can be found such that

$$\mathit{defined}(c_i(\delta_{opportunity}(s_k)))$$

then applying that delta will achieve the desired effect. A common example of this is the decision to implement some previously unrestricted-range integer value using a fixed word size, in order to allow fixed-precision operators (such as machine instructions) to operate on that integer. This obviously changes the meaning of the original program; it no longer operates on unbounded precision integers.

To integrate such a mid-development $\delta_{opportunity}$ (Figure 7.22) one must not only propagate the change forward through the design history, *but also backwards to the original specification.* This is necessary to allow the system analyst to determine, using vocabulary he understands (by virtue of being able to specify with it), the consequences of change, as well as ensuring the presence of an f_0' so that transformational maintenance can be applied later.

Change to use a technology (Δ_f) requires the designer to note the potential utility of a transformation c_i in the library. The ability to inspect the s_j for the appropriate place in which to apply c_i is also necessary; a tool to indicate in which s_i, and where c_i *almost* matches would appear to be helpful. Having settled on a particular s_i, the same program editor outlined earlier, applied to s_i instead of s_0, would be used to capture the specific δ_f necessary to apply c_i.

The procedures we have outlined do forward integration of functionality deltas. We have not explored mechanisms for accomplishing backward integration, but think that most of the necessary ideas are present.

7.6 Evidence for Significant Commutativity in the Design Space

If we hope to take advantage of commuting transformations in the design space, we must be sure it is present often enough for this technique to be useful. It is obviously present, as evidenced by our examples, and is noted by Steier [SA89] after comparing several algorithm syntheses. If it does not occur often enough, the delta integration procedures will still be correct, but so little of the derivation history will be preserved (because *BANISH* chops transformations that fail to *DEFER*) that we might be tempted to simply start fresh each time, contrary to our original purpose.[5]

[5]It may actually be the case that even saving just a few transformations can provide considerable performance gains when re-implementing; [Kam89] shows that reusing even a small plan to solve

We draw our hope for significant amounts of commutativity from three sources:

- Generic: success of conventional software maintenance
- Instance: Experience with transformational porting of software
- Empirical: Speedup measured by Lexical Search algorithm

7.6.1 Commutativity in conventional software construction

Our first indication is the success with maintenance changes are made to conventional (non-transformationally generated) software systems. Virtually all such maintenance leaves a significant portion of the original software unchanged. Our own personal experience of 20 years of building operating systems also convinces us of the stability of existing code. Linton [LQ89] instrumented MAKE and determined that typically only 20% of a system is recompiled after it is changed. Some 6 out of 10 recompilations in a similar environment are caused by poor modularization caused by overly-large source units that are widely visible according to [Bor89], suggesting that only about 10% of a system must actually change. Even this estimate must be too high, as it is measured in terms of compilations of modules, and not the contents of modules, which we suspect stay largely unchanged.

The mostly-unchanged nature of the revised artifacts hints that the original design decisions, however they were made, are preserved, even though we cannot see them directly. Commutativity requires both preservation of the original operators, and the ability to reorder them; preserved design decisions meet part of this condition. The fact that the software looks nearly identical suggests that the order in which the decision to install the delta, before or after the original product, isn't very significant, and lends credence to the idea that reordering should be frequently possible.

7.6.2 Commutativity in the Draco portage project

The initial motivator of the work discussed in this thesis was a project to semi-automatically port the Draco tool [ABFP86] from one LISP dialect to another. The porting process was accomplished by abstracting the source code idioms (used by the Draco source code) in the source dialect, to domain abstractions, stated as functional specification fragments, and then transformationally implementing the functional specification formed by the configuration of domain abstractions that resulted. As we had to manually define the abstractions and the implementations, we naturally

problems decreases problem solving time drastically, and that the savings grow as the problem size grows!

guessed them wrong a number of times, necessitating roughly 10 cycles of correct (the abstractions and implementation transforms), abstract, implement, to obtain a successful port. Each cycle produced an implementation, and we observed that large portions of the successive implementations were identical. The observation of near-constancy of the bulk of the transformationally-implemented code in the face of numerous changes in fact lead to this line of research. We note that the original specification was in fact held constant, but changing the idiom-to-abstraction maps has the effect of functionality changes Δ_f, whereas changing the implementing transforms produced technology changes Δ_C. Here we have evidence of the small-delta implies small-implementation-change in the context of transformational implementation.

In the case of the Draco tool, in fact, all of the refinements from one domain to the next provably commute because they are essentially context-free substitutions. Every transform applied during the porting process were of the refinement type, so in fact a great deal of commutativity in the design space was present.

7.6.3 Commutativity implied by Speedup in Lexical Searching

Our strongest evidence is provided by the empirically determined performance of an algorithm designed to take advantage of commutativity in a search space. *Lexical Searching* [Bax88] is a problem-space search algorithm (see [Pea84] for a thorough discussion of such algorithms). It requires that that all branches through the search space be labeled with elements taken from an arbitrary partially ordered set; this induces a label string for any path through the search space. When commutative operations (note the distinction from operators, which is a special case) in the space are found, Lexical Search explores *only* the path with the lexically smaller label string, thus saving search energy. Lexical search skips only paths which a conventional search would explore fruitlessly.

The laboratory rat for search algorithms is known as the N*-puzzle* problem: an $N \times N$ grid of sequentially numbered, orthogonally sliding tiles, with a single missing tile which provides space into which to slide another tile. The problem is to find a sequence of tile slide movements that organizes the tiles so that the numbers have a particular configuration; a typical requirement is that the numbers increase sequentially from left to right, top to bottom for a solved puzzle, with the blank being the lower rightmost corner. Starting states are scrambled configurations of tiles.

To model the problem space of transformational implementation, a variant of the laboratory rat was bred: instead of a single missing tile, one can have two missing tiles; a typical starting configuration is shown in Figure 7.23. The purpose of this

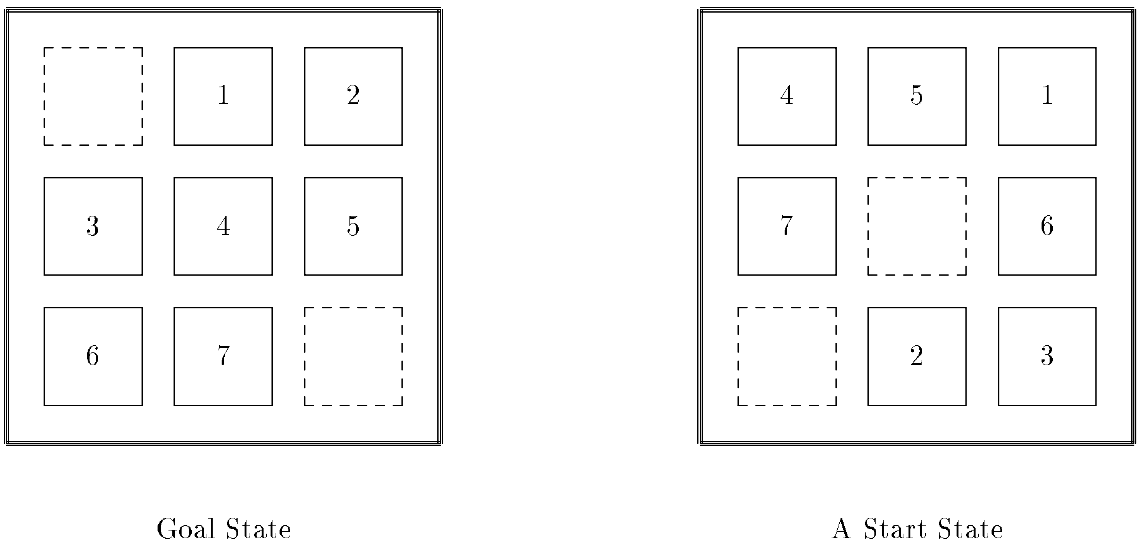


Figure 7.23: 3x3, 2 blank N-puzzle problem

variant is to arrange for operations that sometimes interfere, and sometimes do not. When the blank spaces are far apart, tile moves into one blank space can be performed without regard to moves into the other (consider moves by tiles 1 and 5 in the goal state as examples); such operations commute. Conversely, when the blanks are close together, some moves into one blank disable/enable moves into the other (consider the moves by tiles 5 and 2 in the start state); such operations do not commute. The problem space for this puzzle thus has patches where operations commute (blanks temporarily far apart) and where they do not (blanks come near one another). We argue that this approximates the type of space for software implementation.

A graph of the ratio of effort by a conventional search to a lexical search, to find solutions to some 350 random problems for 3x3 puzzles in this space, versus length of solution[6], is shown in Figure 7.24.

[6]The branching factor in this space is about 5, and solutions of length 22 are being generated by exhaustive search; this would truly be an immense amount of computation if the search were not augmented by a number of algorithmic shortcuts, such as IDA* [Kor85], elimination of inverses, etc. which are inappropriate to discuss here. The fact that both searches produced the same solution was verified in every instance.

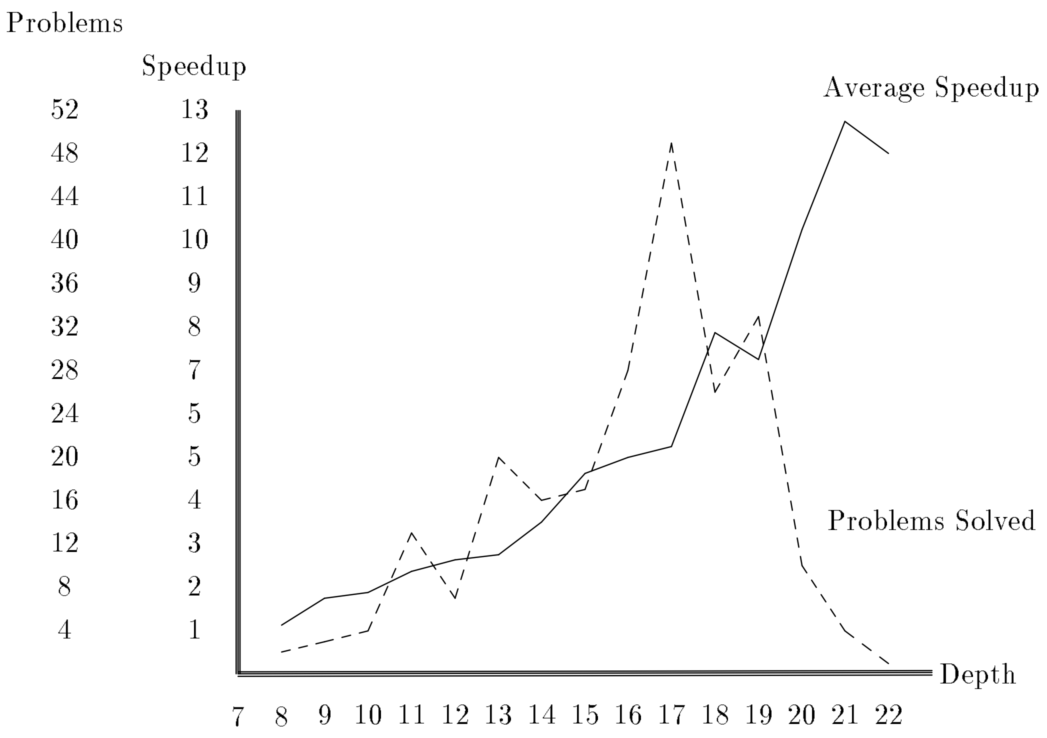


Figure 7.24: Speedup using Lexical Search in artificial design space

The graph shows that lexical search saves a factor of 2 effort for solutions of length 10, and a factor of 10 effort for solutions of length 22, with the savings growing more or less monotonically in between. The interpretation of the savings by lexical search is that there are as many commutative paths in the search space as the magnitude of the savings.

Collecting ratios for individual 3x3 problems averaged about 1 hour of CPU on a 20Mhz Intel 386 under a compiled version of CommonLisp. Attempts to extrapolate this data by scaling up to 4x4 problems were stymied by inordinate execution costs; collecting this ratio for a single 4x4 problem cost 1 week of CPU, but showed a savings of 100 times for a solution of length 27.

This evidence suggests that commutativity of paths in the space goes up monotonically with solution length, and may in fact grow very quickly with the number of steps. The analysis in [Bax88] suggests that the growth is exponential in the solution length; this is not surprising when one considers that permutations are being eliminated, and the number of permutations of a string is $n!$, essentially an exponential function. When we consider that a transformational implementation of a moderate size specification has on order of 10^4 steps (Figure 3.8), the number of different paths to the same point in the design space would appear to be truly immense. This give us great hope that a derivation history can be rearranged for our convenience, leading to the same solution point; this is why we believe the swap procedure in the functional delta integration process is likely to be effective.

Our purpose in defining Lexical Searching was twofold: first, to explore the amount of commutativity in the design space, and secondly, as an enhanced mechanism usable by a transformation system. We have yet to use Lexical Search in the second application.

7.7 Summary

Our purpose was to define and provide procedures for transformational maintenance given just a derivation history.

In this chapter, we have:

- Shown the utility of "commuting" transformations in the design space for managing integration of maintenance deltas into a derivation history
- Provided a theory behind the notion of "commute": *DELAY*
- Identified frequently-occurring special cases and outlined procedures for implementing those cases for tree transformations: *SWAP* and *DEFER*
- Defined procedures for removing unwanted transformations and their dependents from an existing derivation history: *BANISH*, *BANISHBATCH*
- Noted the utility of *BANISH* as a mechanism for use in dependency-directed backtracking
- Provided procedures for revising a derivation history and implementation given technology Δ_C and/or functional deltas Δ_f
- Shown different classes of evidence for the presence of significant commutativity in the design space, justifying the use of mechanisms such as *DEFER*
- Given an empirical demonstration of commutativity in certain spaces with properties similar to those of transformation systems.

These ideas have been tested by a proof-of-concept implementation of the Δ_f integration procedures.

We have demonstrated the theory and practicality of reuse of derivation history for certain types of deltas. Additional information, contained in the design history, is both needed and needs revision in order to handle other types of maintenance deltas. We will consider these in the next chapter.

Chapter 8
Integrating Maintenance Deltas into Design Histories

Chapter summary. Most maintenance deltas affect the design justification. This chapter describes procedures for integrating those maintenance deltas into a design history. Such procedures typically mark places in the design history that are inconsistent with the delta, eventually prune away their dependents, and then repair the remaining history by an agenda-oriented TCL execution scheme. Central to execution is insertion of a transformation into a derivation history, and how the resulting complications in updating the design history are handled. Procedures for each of the various deltas are described.

We have seen how, in Chapter 7, to integrate certain kinds of maintenance deltas ($\Delta_{\mathcal{C}}$, Δ_f) into that portion of a design history called the derivation history. Changes to the derivation history indirectly affect the design justification, and so we also need procedures to adjust the design history when changes are made to the derivation history.

Other maintenance deltas change the means by which the performance specification is achieved (Δ_M), change the specification (Δ_G, Δ_v), or the meaning of the specification ($\Delta_{\mathcal{G}}$, $\Delta_{\mathcal{P}}$, $\Delta_{\succeq}$, $\Delta_{\mathcal{V}}$). Such maintenance deltas also require revisions be made to the design history, as well as inducing changes on the derivation history.

In this chapter, we consider procedures for revising a design history to be consistent with each type of maintenance delta. We will find the procedures defined for revising a derivation history useful. The design history procedures are currently less developed than those for derivation history management, so we only sketch the mechanisms.

8.1 Integration $\equiv$ Revise, Mark, Prune, and Repair

It might be possible to construct specialized design history integration methods for individual delta types that could directly modify the design history. We have chosen a more conservative approach, in which all the design history delta integration methods follow the same general sequence:

1. **Revise**: adjust structures in delta-specific fashion;
2. **Mark**: identify agenda items in the design history inconsistent with the delta;
3. **Prune**: prune away inconsistent agenda items, and all the forced choices dependent on those already pruned; and
4. **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.

A delta may require direct revision of the design history or the support libraries. Revisions to support libraries have been defined in Chapter 6. In particular, a delta may augment the available support technology, allowing the repair process to take advantage of new opportunities. Such revisions must take place before the repair process is started.

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. For many deltas, the design history revision process is limited to simply marking. Direct revision of the design history takes place when handling performance changes, Δ_G.

Undesirable agenda items are then *pruned* away, leaving an incomplete design history (see Section 5.3.1)[1]. 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.

[1]Nothing about our approach prevents us from applying a delta in the presence of an already incomplete design history.

Having finished the pruning process, the incomplete design history must be repaired by processing incomplete agenda items. By designing TCL around a planning approach that uses agenda items, rather than a procedural metaprogramming approach (like that of PADDLE [Wil83] or Goldberg's system [Gol89]), we can accomplish this by simply passing the incomplete design history to the TCL execution engine. 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. In this fashion we avoid the need for a special "replay" mechanism; completion for delta integration and completion for initial implementation are identical.

We consider these activities in the order Prune, Repair, and Mark, because of the commonality of the Prune and Repair processes over all the delta integration procedures, and the diversity of the Mark processes. We will not discuss revision procedures for support technologies. Any other revision procedures will be discussed with the corresponding Mark procedures.

8.2 Pruning 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.

We assume that we have a design history in which some agenda items have been marked *undesirable*. We must prune away:

- all portions of the design history which are directly marked
- every agenda item that depends uniquely on some pruned agenda item
- agenda items generated as descendents of those marked
- agenda items which are indirectly dependent on pruned agenda items

Leaf agenda items are transformations, and pruning them requires that we eventually revise the derivation history portion of the design history.

Our approach is to remove agenda items from the design history known to be bad, or known to depend on some agenda item which will be pruned for which there is no alternative. Those agenda items which are indirectly dependent may not be discovered immediately; we mark the design history in such a way that they will eventually be discovered and pruned. 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 upwards until some parent agenda item that provides an alternative is found, then that item is marked as *incomplete*, and all agenda items below that point are removed from the design history. Agenda items which provide alternatives are *OR*, *ELSE*, and *ACHIEVE*[2] (*APPLY* allows alternatives, but cannot be a parent). The intervening agenda items by definition provide no alternatives, and will have to be removed. A clever pruning process would leave a to-be pruned agenda item directly under a not-to-be pruned parent, annotated as "don't try this particular alternative again." The removal is easily accomplished with a recursive procedure that removes all the descendents of a son, and then deletes the son from the design history. Because the root of every design history is an *ACHIEVE* node, this traversal process can not climb past the root of the entire design history. When pruning a leaf agenda item that *APPLY*s a transformation $H[j]$, we additionally mark $H[j]$ in the derivation history as *undesirable*; during the plan repair process, an eventual banish will remove the marked transformation. Revision of the derivation history by banishment can invalidate downstream transformations; the agenda items which produced such invalid downstream transformations are indirect dependents and must also be pruned as encountered. Notice that we eagerly prune obviously invalid agenda items, but only lazily mark transformations in the derivation history. Each agenda item marked as *undesireable* must be pruned in this fashion before any attempt to repair the design history is made.

In Figure 8.1, we show the pruning process. A sequence of activities is numbered:

1. mark agenda item *undesirable*
2. prune the *undesirable* agenda item and its dependents
3. mark dependent transformations as *undesirable*
4. *BANISH* an indirectly dependent transformation
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. We will see how this takes place in Section 8.3.3.

[2]The variant *ACHIEVEBY* is treated in the obvious way, nearly identically to *ACHIEVE* so we do not discuss it further.

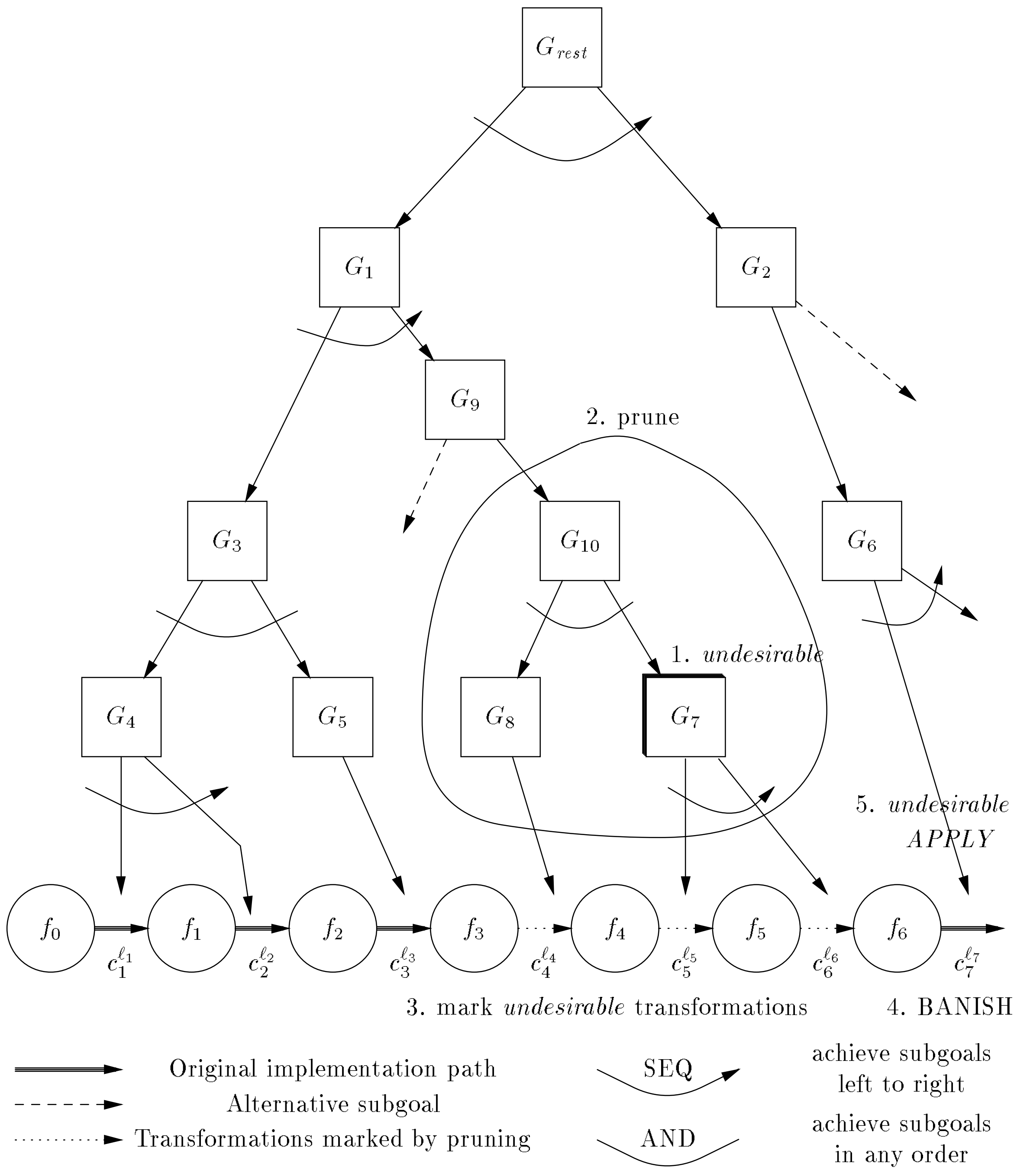


Figure 8.1: Pruning a Design History back to an alternative

Pruning the "subtree" below an agenda item a_m in a design history is slightly complicated by the possibility that a descendent agenda item is actually shared by another. In such a case, the shared agenda item a_s may actually need to be removed, or be simply disconnected from this subtree. We must remove a_s (shown as G_9 in Figure 8.2) if it only serves a_m (G_6); we can retain a_s (G_9 in Figure 8.3) if it serves some other relative of a_m (G_3).

Certain agenda items (*APPLY*, *LOCALE*, *ACHIEVE*, etc.) may use locale values (variables) generated by other agenda items. Should a locale-value generating agenda item be pruned, all of its locale-value using dependents must be adjusted. We mark each locale-using dependent as *incomplete*, and its sons must be removed; this ensures that the plan repair process will later re-execute the locale-using dependents. Since all agenda items in the subtree below the alternative are removed by the pruning process anyway, we need only process locale-using dependents of the alternative's immediate sons in this fashion. Finding the set of locale-using dependents is easily accomplished by taking the transitive closure of the *dependency* slots stored in the *symboltable* of the pruned agenda item.

8.3 Repairing the Design History

Repairing the pruned plan consists of executing incomplete agenda items according to their actions, perhaps generating additional agenda items in the process. Since each agenda item represents the execution of a TCL program fragment, and such incomplete items can be produced by the pruning process in the middle (according to the sequencing constraints in the design history) of the logical transformational implementation process, 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

We purposely glossed over the details of TCL execution in Chapter 4 to avoid any preconceptions about order of execution. The execution order for a metaprogramming language like PADDLE [Wil83] or the tactics language of Goldberg is totally determined, and very difficult to restart at arbitrary points, which 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. Since some agenda items invoke methods or *PLAN*s, processing them produces subagendas. The design history is a static snapshot of the processed agenda nodes and the sequencing constraints.

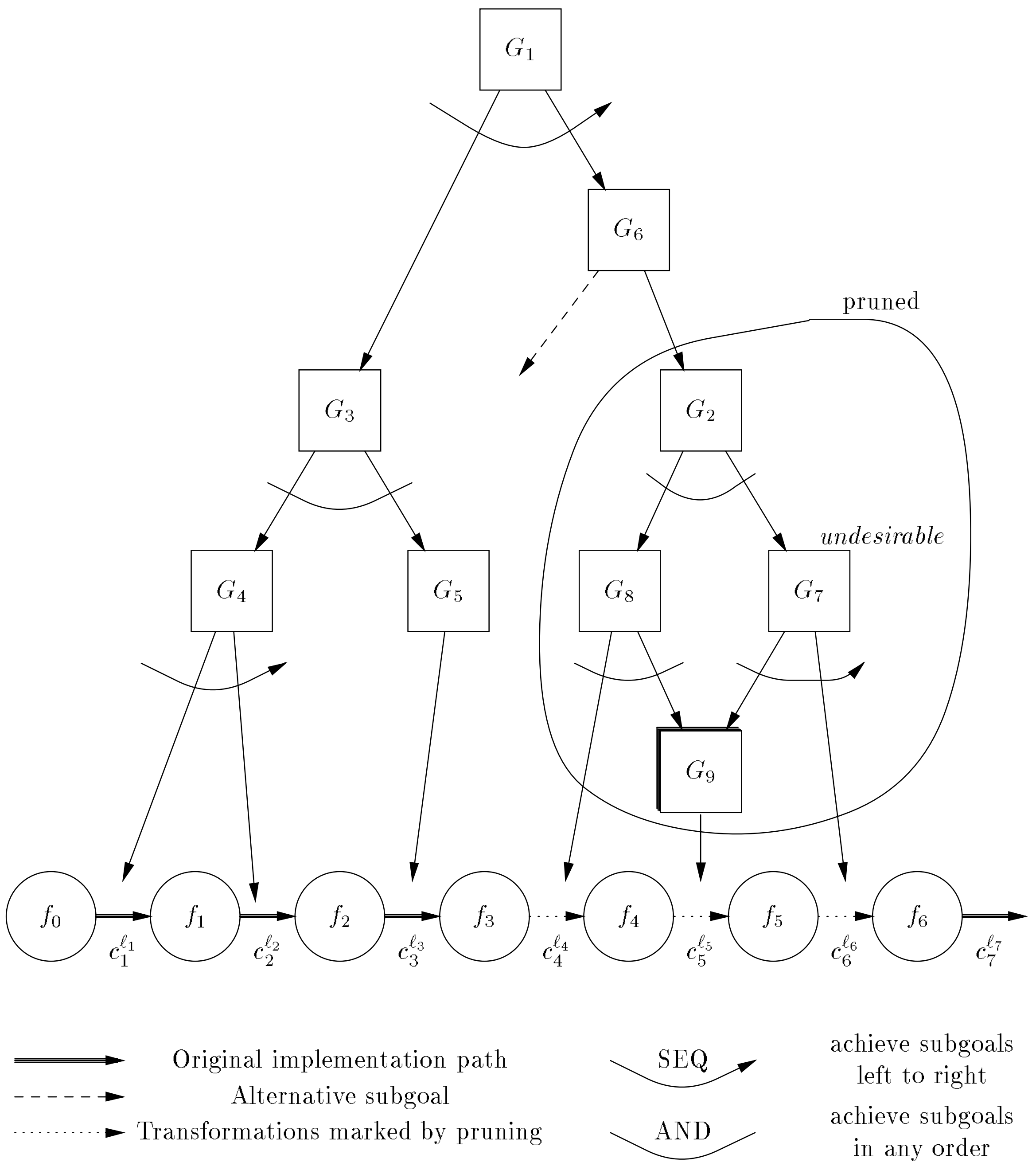


Figure 8.2: Pruning a shared agenda item

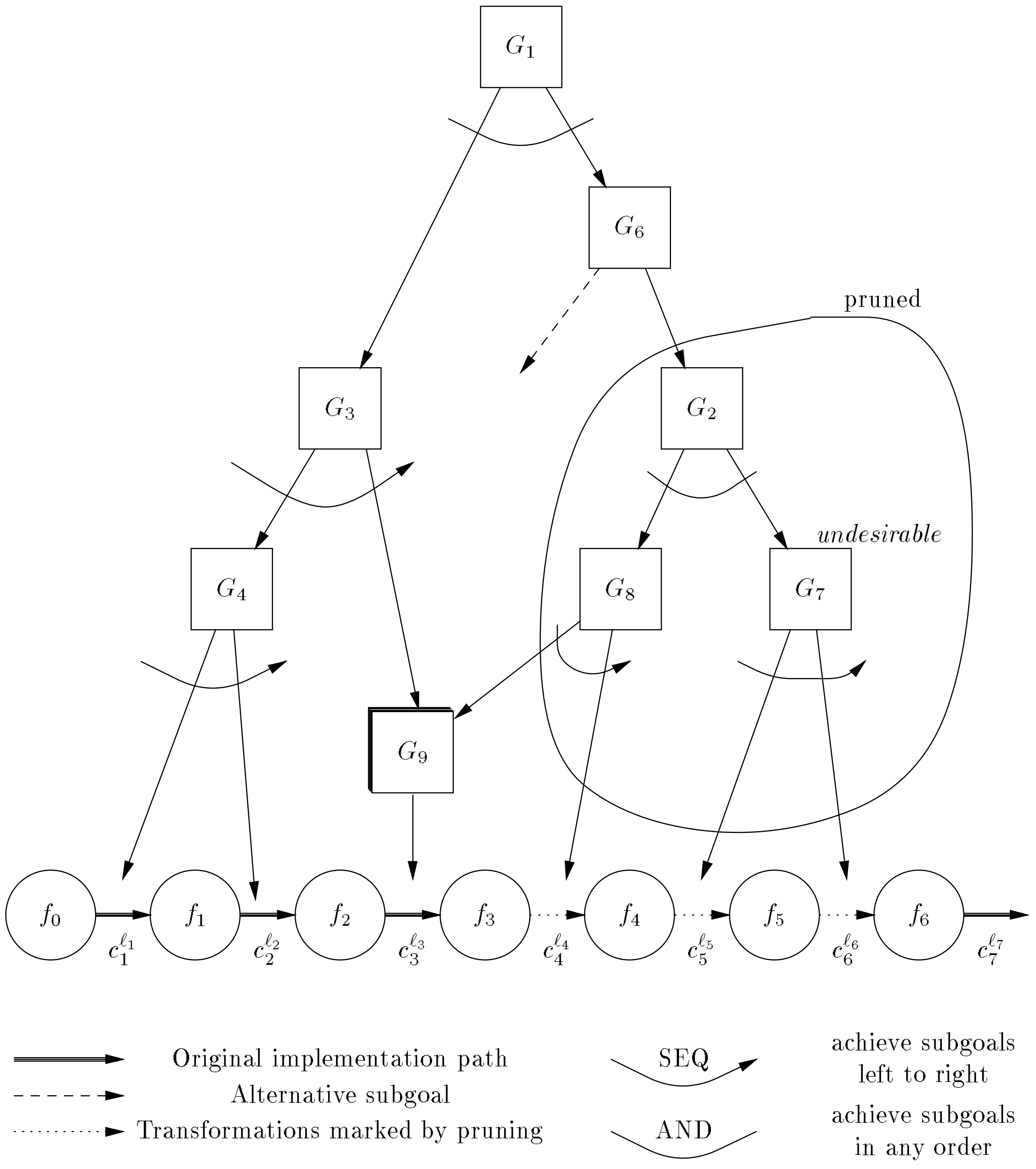


Figure 8.3: A shared agenda item that need not be pruned

8.3.1 Agenda-oriented execution process

Given an incomplete design history, there may be a number of agenda items which are individually incomplete. An agenda-oriented execution method chooses any one of them and executes it, marking that agenda item complete, and possibly adding more agenda items.

Agenda items must be processed in some order. 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 (those agenda items and states which must follow the selected agenda item in the design history) as early as possible. Such ripple affects will cause later parts of the design history to be pruned and/or revised. While we cannot in general avoid handling such ripple, our heuristic minimizes the amount of revision required by doing early actions while the design history is as small as possible. We will see later, when we discuss execution of *APPLY* actions, how an agenda item can affect following design states. Processing the earliest incomplete agenda item first also conveniently honors sequencing dependencies required by locale-use.

Each agenda item specifies a TCL action taken from some TCL method (Section 5.3.2) 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. Alternative generators interact with the pruning process as discussed below. Since other actions are executed in similar ways we will not discuss them.

Agenda items with alternative completions

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* [via multiple possible locaters in a locale], *OR*).

Agenda items which allow alternative completions contain an alternative generator as internal state. Creation of the agenda item initializes the generator. Execution of the agenda item causes the next alternative to be produced, unless there are no more, in which case this agenda item is unsatisfiable. Pruning the design history back to an agenda item advances its alternative generator. An obvious pruning optimization is to continue pruning upwards if advancing an alternative generator exhausts it.

Unsatisfiable agenda items are simply pruned (Section 8.2), exposing an even higher level agenda item that provides an alternative. The remaining design history is then passed back to the repair process (Section 8.3). Should the unsatisfiable agenda item be the root of the design history, then the program specification is unimplementable by the transformation system given the current support technology.

8.3.2 Execution of *PLAN* agenda items

Execution of an agenda item a_p containing a *PLAN* action happens in a way similar to expanding a node in a hierarchical nonlinear planning process [CM85, Kam89]. The design history is augmented by creating new, incomplete agenda items a_i for each element of the *PLAN*, and an ordering relation is placed on these newly created agenda items according to the ordering $>$ given by the *PLAN* action. Each agenda item a_i is annotated with its parent a_p, and the parent is annotated with its list of sons a_i to allow later bidirectional traversal of the design history.

When a new agenda item is proposed for addition at a point in the design history, the TCL execution engine searches the design history to see if there is an existing agenda item which is equivalent in terms of execution order, and has an identical action. Should such an agenda item exist, it is returned instead of creating a new agenda item. This is the mechanism that produces shared agenda items in the design history.

8.3.3 Execution of *APPLY* agenda items

An agenda item containing an *APPLY* action 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 $(\Pi H)(f_0)$ of the derivation history, but with plan repair we can have out of order execution; the transformation might apply to *any* state in the derivation history.

In the case of 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 (largest index i) state $(\Pi H[1..i])(f_0)$ 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. The appropriate derivation history index is the one preceding that used by the earliest, already-complete *APPLY* agenda item which must necessarily follow the new transformation. 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, and so this scheme conveniently subsumes the simple linear execution model (see Figure 8.4, in which we are applying a transformation under G_5, and note the absence of ordering under the root). To be able to apply a transformation at any point along a derivation history, we must have the entire state available to us at every point. This can be accomplished accomplished by computing $(\Pi H[1..i])(f_0)$ where i is the insertion point; one can cache every nth state to minimize this computation. We eventually hope to be able to use partial states in a fashion similar to the way they are handled in nonlinear planners to avoid this cost.

A more interesting case is shown in Figure 8.5, 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. This alternative transformation must be applied just prior to the earliest son of G_7, i.e., it should replace $H[3]$, and should therefore be applied to $f_2 = (\Pi H[1..2])(f_0)$. This is accomplished by treating the new transformation to be applied as a functional delta δ_f and using an extension of the *INTEGRATEMIDDLE*$_f$ procedure (see Section 7.4.2) to construct a new derivation history, shown in dotted outlines, growing horizontally in the figure, with the new transformation inserted in the middle.

Inserting a Transformation in a Design History

Inserting a transformation into the derivation history fundamentally requires us to build a new ladder (Section 7.4) to obtain the revised derivation history. The ladder construction process repeatedly attempts to *PRESERVE* a transformation in the face of the delta, or failing to preserve it, *BANISH*es the offending transformation. Doing this in the context of a design history requires that we extend the procedure *INTEGRATEMIDDLE*$_f$ to respect constraints from, and adjust the design history in parallel with ladder construction:

- for determination of revised locaters for *PRESERVE*
- to re-validate proposed replacements for preservable transformations
- to adjust the design history to record the replacement
- to handle unpreservable transformations

We will discuss these topics in the following sections.

Using Locale information to Compute Revised Locators: In attempting to *PRESERVE* a transformation, what guidance do we have for choosing a locater?

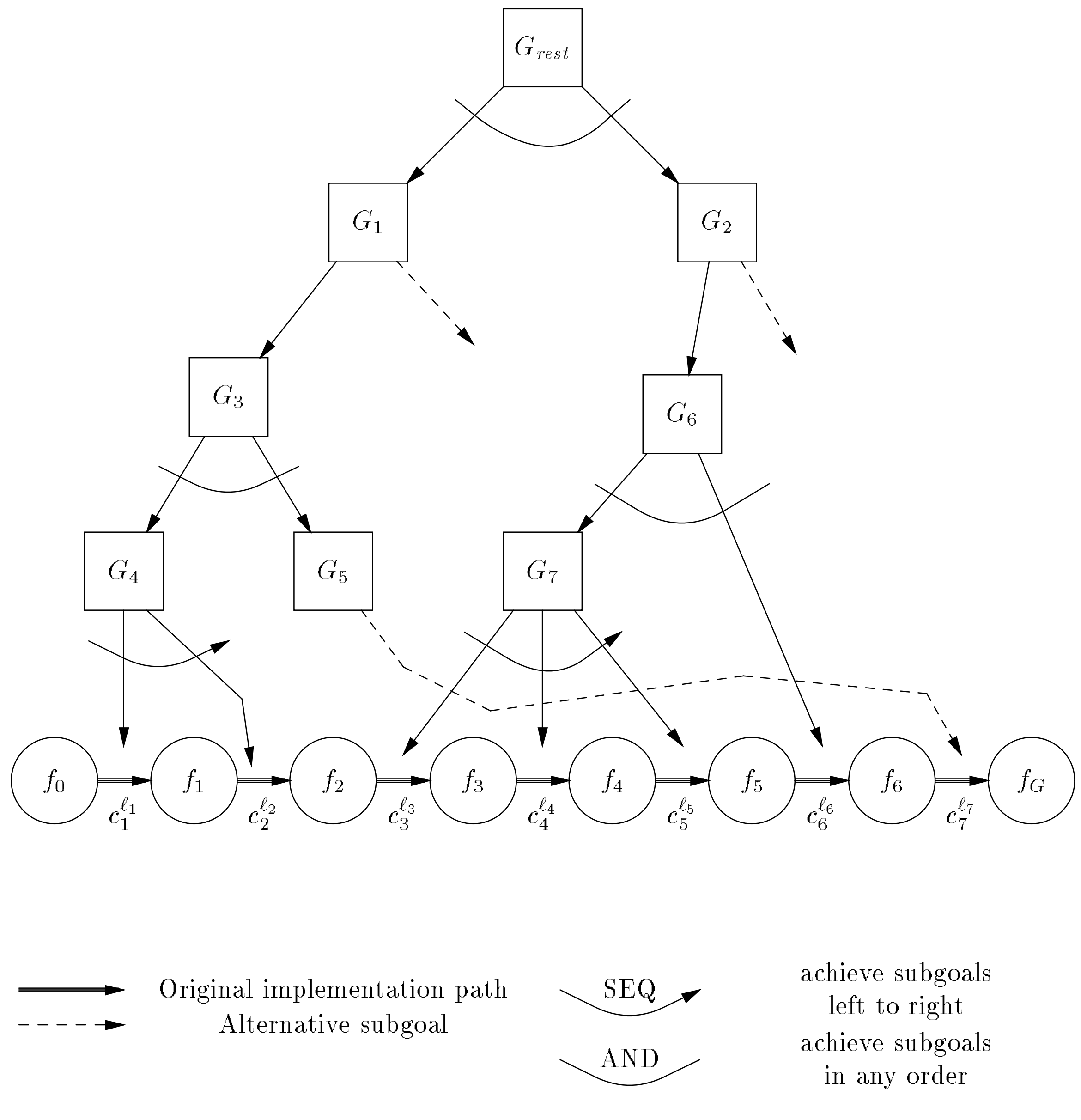


Figure 8.4: *APPLY*ing a transformation as late as possible

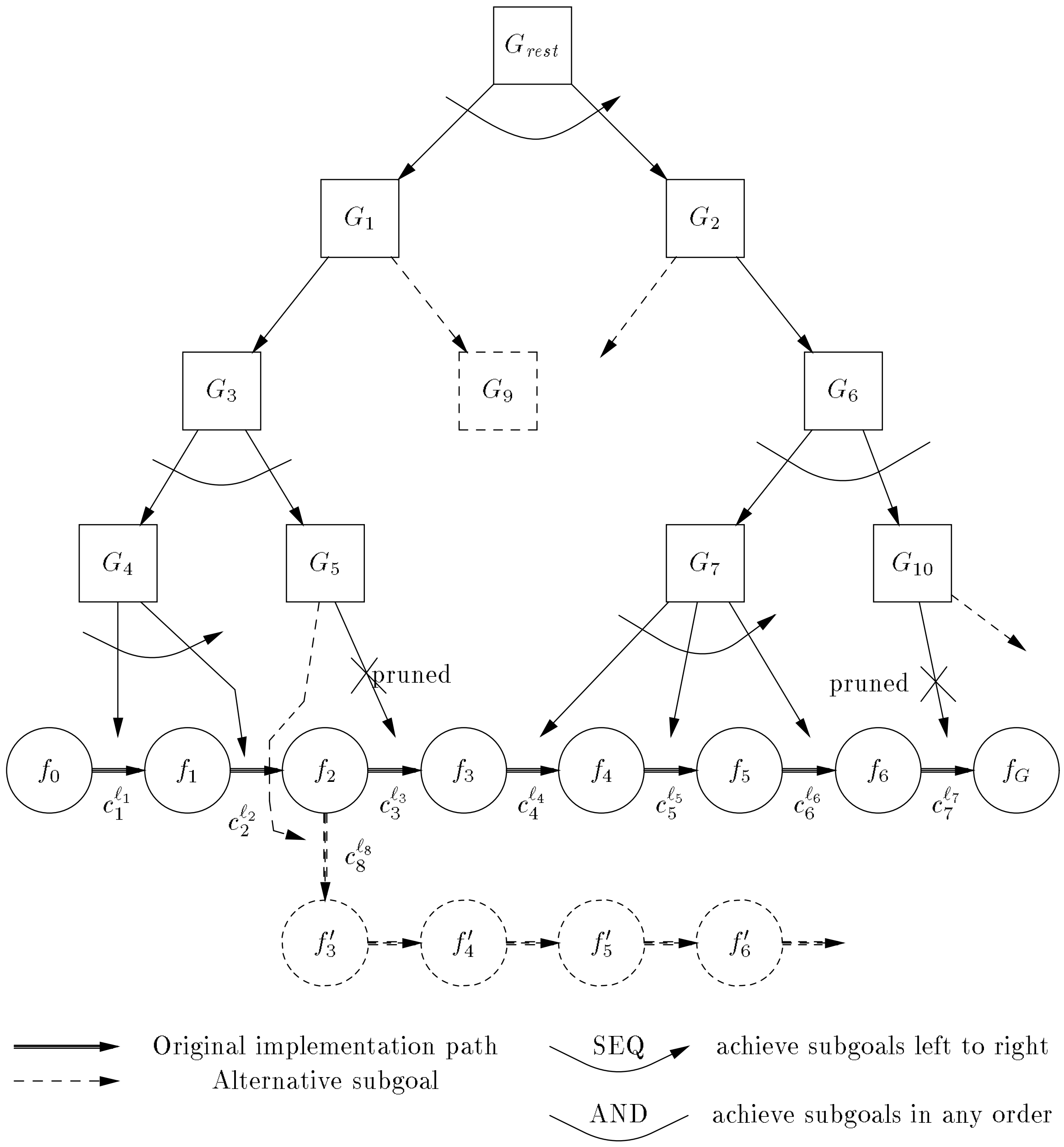


Figure 8.5: Repair by Inserting a Replacement Transformation

Our theory from Sections 7.2.2 and 7.4.1 says that any locater that satisfies the commutative square will do; but there may be more than one. The locale constraint from the *APPLY* agenda item in the design history provides an additional constraint. Even this may not be enough to uniquely determine a choice. Our current solution is to take the first locater satisfying both the commutative square requirements and the locale requirements. Should this turn out to be wrong, eventually some method postcondition will fail during re-validation (see below) and backtracking will occur. In general, it is not possible to prevent this from happening, so we do not feel justified in investing a great deal more energy in improving this solution. What might be helpful would be mechanisms for allowing tighter control of locale constraints, in an attempt to eliminate as many false locaters as possible. Should there be no solution satisfying the commutative square and the locale constraint, then the transformation is unpreservable and is marked *undesirable*.

Re-validating Replacement Transformations: The proposed replacement must be checked to ensure it still satisfies its purpose in the plan which generated it. This can be accomplished by walking up the design history via parent links, and re-executing any dynamic postcondition agenda items (necessarily last sons of a *PLAN* whose action is *REQUIRE*) until a parent is found with other sons that necessarily follow the *APPLY* agenda node, or a parent is found with some incomplete descendents. In Figure 8.6, we see that $c_3^{\ell_3'}$ has been partially banished in the original history, and that $c_4^{\ell_4'}$ and $c_5^{\ell_5'}$ have been preserved, and the ladder constructing process is attempting to preserve $c_6^{\ell_6}$. The preserved transformation $c_6^{\ell_6'}$ requires checking the postconditions of G_7; we need not check the postcondition of G_6 because it has an incomplete son G_{10}. Should any postcondition check fail, the purportedly *PRESERVE*ed transformation is not achieving its purpose, the replacee is marked *undesirable* in the derivation history, and typically *BANISH*ed immediately. Validation of a transformation via a path containing a shared agenda item requires that the validation process be carried through to all the parents; failure to validate along one path requires that the shared agenda item cease being shared along that path.

Our approach to preservation of transformations ensures that any transformation (both the transform and the locater) interactively chosen by the software engineer (Section 4.2.3) to satisfy a particular goal is retained by the repair process if it is still valid. In the complete replay of purely generative design information, such applications are lost.

Adjusting the design history for the replacement: A re-validated replacement transformation must replace its source in the originating *APPLY* agenda item. In addition, the revised locater must be propagated to locale-value dependents found

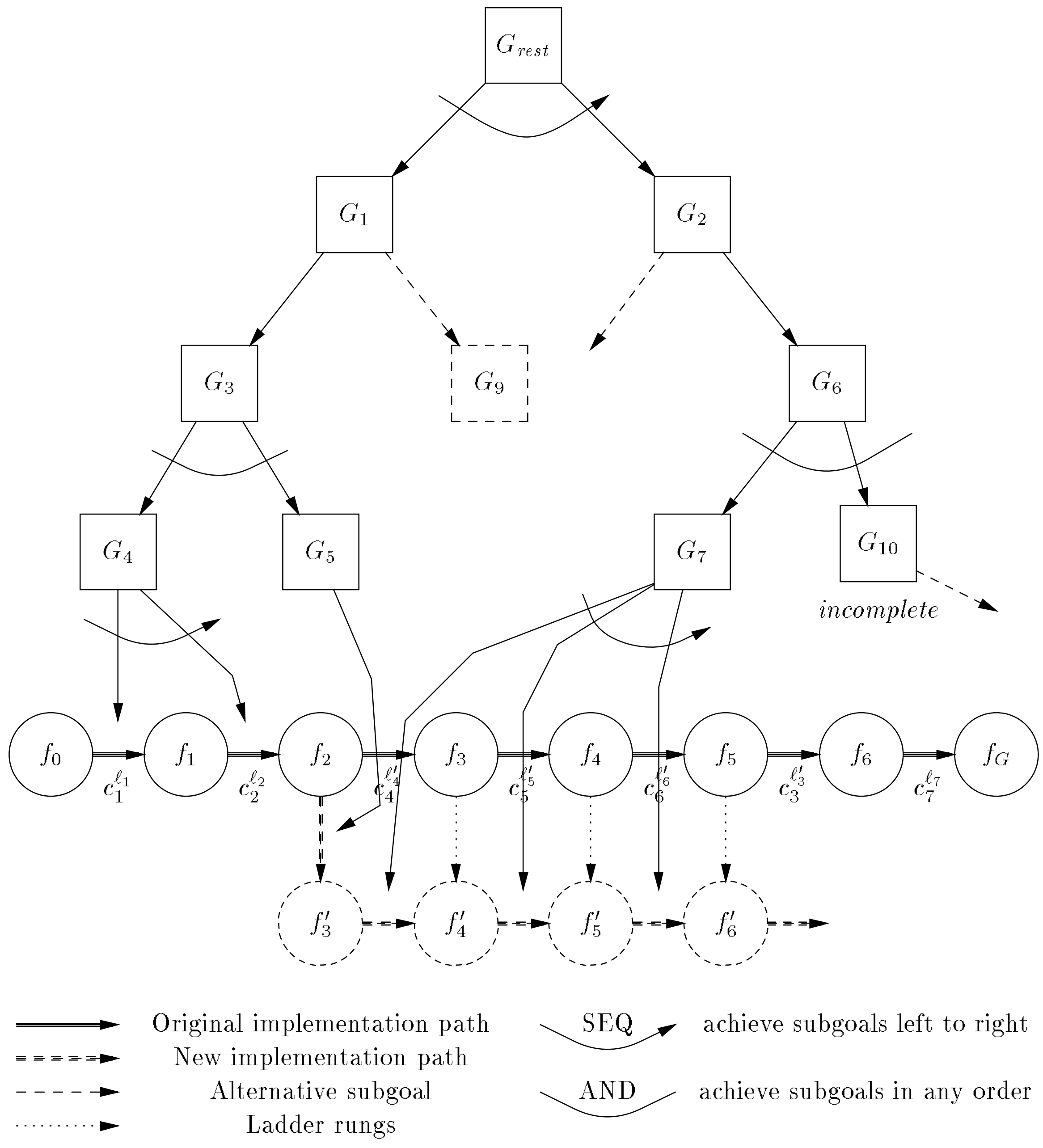


Figure 8.6: Validating a *PRESERVE*d transformation

by following the dependency links in the *symboltable* and recomputing the locale expressions found. The propagation can stop when a recomputed locale value matches the original value. The common case is when the locater does not change; no propagation is needed. Locale-dependent *APPLY*s for which the locater changes must be marked as *incomplete* to force re-application. Since locale-value dependents must necessarily follow the replacement transformations, *APPLY*s that must be marked in this manner are later in the derivation history.

Handling Unpreservable Transformations: When a transformation is *BANISHed* or marked as *undesirable*, the agenda item that generated it is marked as *undesirable*. It is sensible to prune the *undesirable* agenda item immediately, to ensure that dependent transformations in the derivation history are also marked *undesirable*. Use of *BANISHLAZY* delays rearranging the tail of the derivation history as long as possible in an effort amortize the cost of banishing over the largest number of *undesirable* transformations.

A difficulty one can have when marking a transformation as *undesirable* is that it may serve as part of a larger plan; this is why the pruning process climbs the design history until an agenda item providing an alternative is found, and then removes everything down to the leaves of the subplan under the alternative. This ensures that related transforms involved in the plan are also eventually removed. Related steps can be arbitrarily far apart in the design history. In particular, an *undesirable* transformation late in a derivation history may be supported by another transformation arbitrarily early in the derivation history.

Now consider Figure 8.7. A problem can occur while building the ladder. The delta may propagate past a particular transformation $c^{\ell}_{preserved}$, and then conflict with some $c^{\ell_2}_{preserved+k}$ further down the history (in the diagram, $c_{preserved}$ is c_4 and $c_{preserved+k}$ is c_6, and our current delta is δ_5). While $c^{\ell_2}_{preserved+k}$ may be banished, if some plan (G_6) in the design history insists that $c^{\ell}_{preserved}$ together with $c^{\ell_2}_{preserved+k}$ work together atomically, then $c_{preserved}$ must also be removed from the derivation history. This invalidates the part of the ladder built by preserving $c_{preserved}$ ($c_4^{\ell_8}, c_5^{\ell_9}, \delta_4, \delta_5$).

It is easy to remove $c_{preserved}$ from the original derivation history; we simply banish it. The problem is that our delta is already beyond $c_{preserved}$; what is the effect on the propagated delta?

We know of no general way to decide that $c_{preserved}$ must be banished because of an eventual conflict with a supporting transformation $c_{preserved+k}$, without pushing the delta through the intermediate transformations, because the nature of delta is (possibly) changed by the intermediate transformations. There seems to no alternative to backing up, marking $c_{preserved}$ as *undesirable*, and restarting the ladder constructing process from the point where $c_{preserved}$ was applied.

This suggests that, when used in conjunction with a design history, the ladder constructing process must actually construct and retain the entire ladder, including the deltas forming the rungs, because arbitrary backup may be required to handle an unpreservable transformation. This raises the space requirements a factor of three. A more difficult problem is need to retain all the presumed-large states (this was handled incrementally by the derivation history version of $INTEGRATE_f$); this is already a requirement in order to be able to insert transformations at arbitrary points in the design history (Section 8.3.3) and the same solutions apply.

Having discussed design history pruning and repair, we now consider how to mark and adjust the design history according to various types of deltas.

8.4 Integration of Functional Deltas Δ_f

Integration of functionality deltas into a design history is easy because plan repair already does most of the work. There is no need to mark any part of the design history at all. It is only necessary to *APPLY* the functional delta δ_f to f_0. The ladder-revising mechanism described in Section 8.3.3 will propagate the changes into the design history appropriately. The extended design history must be adjusted as shown in Figure 8.8; the new functionality delta is added underneath $G_{invariant}$ to record its purpose.

8.5 Integration of Technology Deltas $\Delta_{\mathcal{C}}$

Integrating technology deltas ($\Delta_{\mathcal{C}}$) requires removal of newly-illegitimate transforms from the design history, and possible use of newly-added transforms. Removal of newly-illegitimate transformations requires marking:

- those illegitimate transformations in the derivation history
- the *APPLY* agenda items which generated them

Pruning and repair can follow immediately. The use of truly new transforms from $\delta_{\mathcal{C}}.\Delta_{\oplus}$ is left to the repair process where they will be discovered indirectly via invocation of modified methods *APPLY*ing the new transforms.

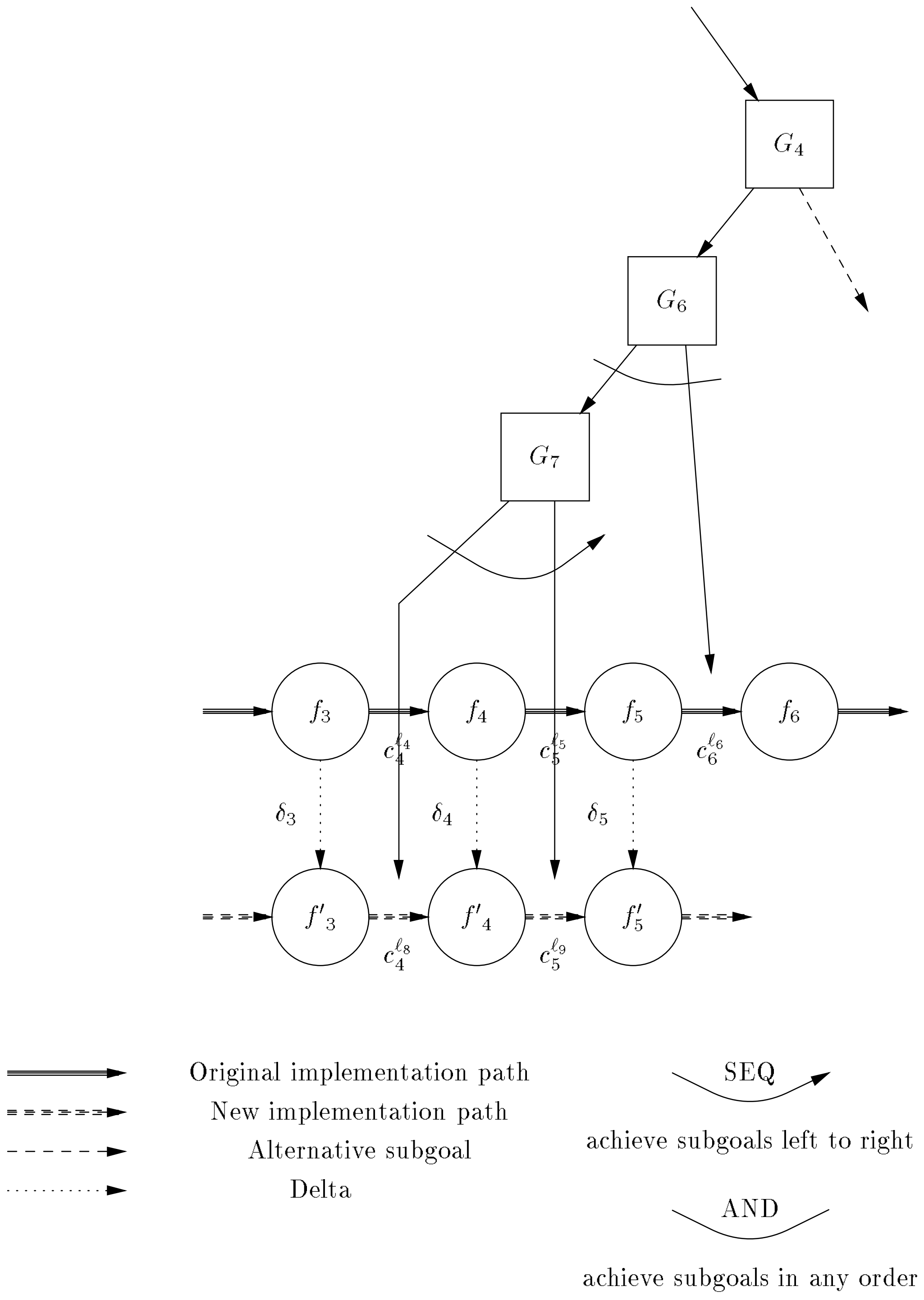


Figure 8.7: Preserved transformation $c_4^{\ell_4}$ supporting unpreservable $c_6^{\ell_6}$

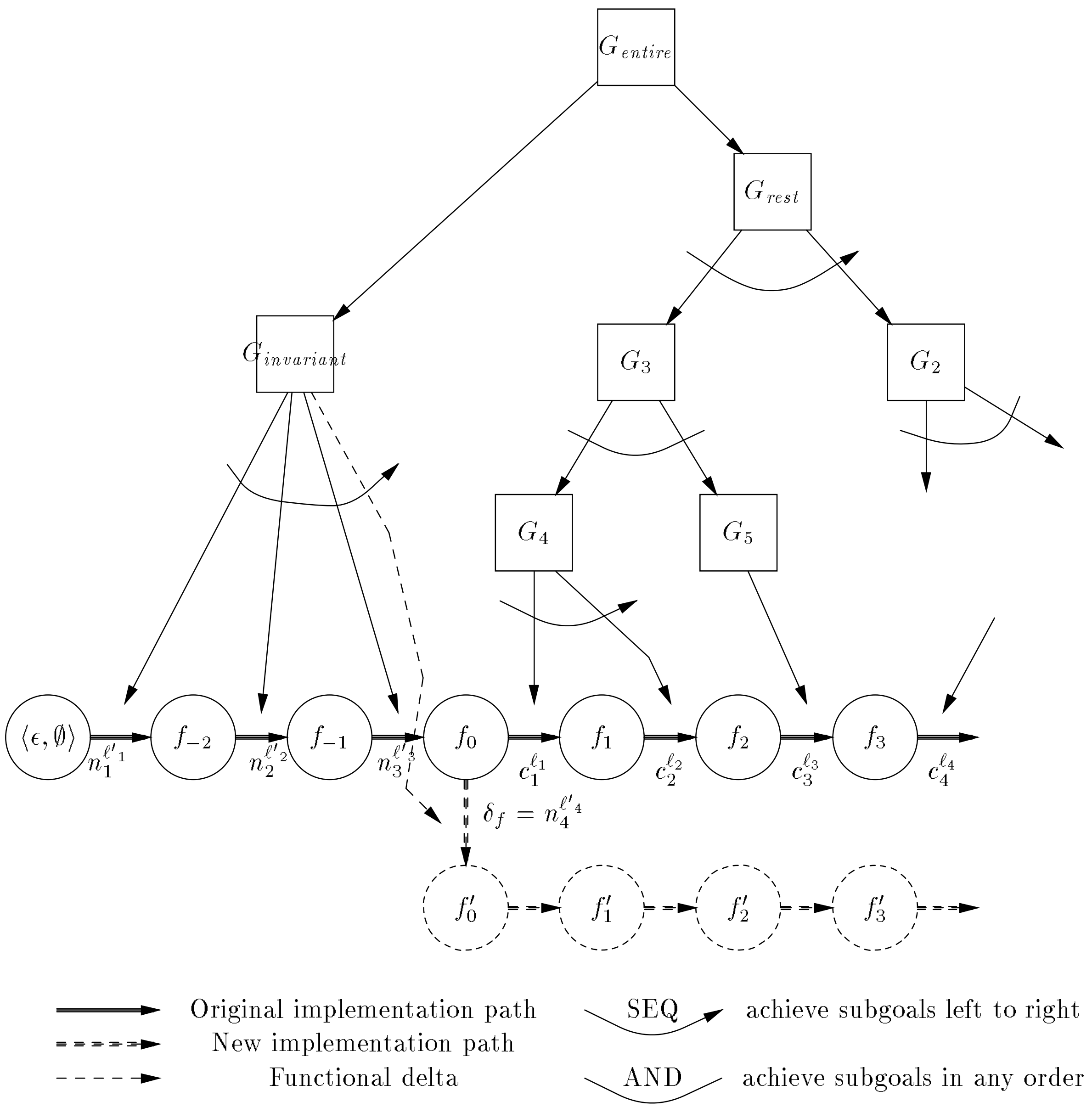


Figure 8.8: Integrating a Functionality Delta Δ_f

The transformations $H[i]$ in the derivation history for which $H^{\mathcal{T}}[i]$ are members of the set(s) of deleted transforms $\delta_c.\Delta_\ominus$ are marked as *undesirable*, exactly as outlined in Section 7.3. This will force eventual removal of these unwanted transformations when some later derivation history scanning process (transformation insertion or *BATCHBANISH*) encounters the marked transformations.

The agenda items which *APPLY*'d the now-illegal transformations could be marked as *undesirable*. Instead, we simply mark them as incomplete. The reason for this is the consistency requirements on deltas. No transform can be applied unless explicitly mentioned by an *APPLY* action in some TCL method. If a transform c in $\delta_c.\Delta_\ominus$ is being permanently deleted from the transform library $C_{library}$, then all methods referencing that transform must be deleted or modified to no longer reference the transform; if the transform is simply being replaced (due to an error in domain engineering), then the applying methods will be untouched. If the applying method remains untouched, we merely desire that the replacing transform be applied instead of the replacement; our policy of marking the agenda item as incomplete will ensure that the replacing transform is eventually installed. In the case of a deleted transform, there will be a corresponding δ_M in the composite delta, that when processed, will prune the offending *APPLY* agenda item (see Section 8.8). The fact that pruning occurs before repair ensures that for any updated transform, the old version is removed before the new version is applied.

8.6 Integration of Performance Deltas Δ_G

Performance deltas Δ_G change the specification of the desired artifact. We have seen that typical specifications are often given as mixed specifications $\langle f_0, G_{rest}\rangle$. Performance deltas are can then limited to changes of G_{rest}. We have seen that changing G_{rest} requires that a different path through the design space be chosen to an alternative implementation in Figure 7.1. The choice of path through the design 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.

Our approach is to revise the design history by propagating the specific δ_G in a top-down fashion, paralleling the design history construction method by goal decomposition.

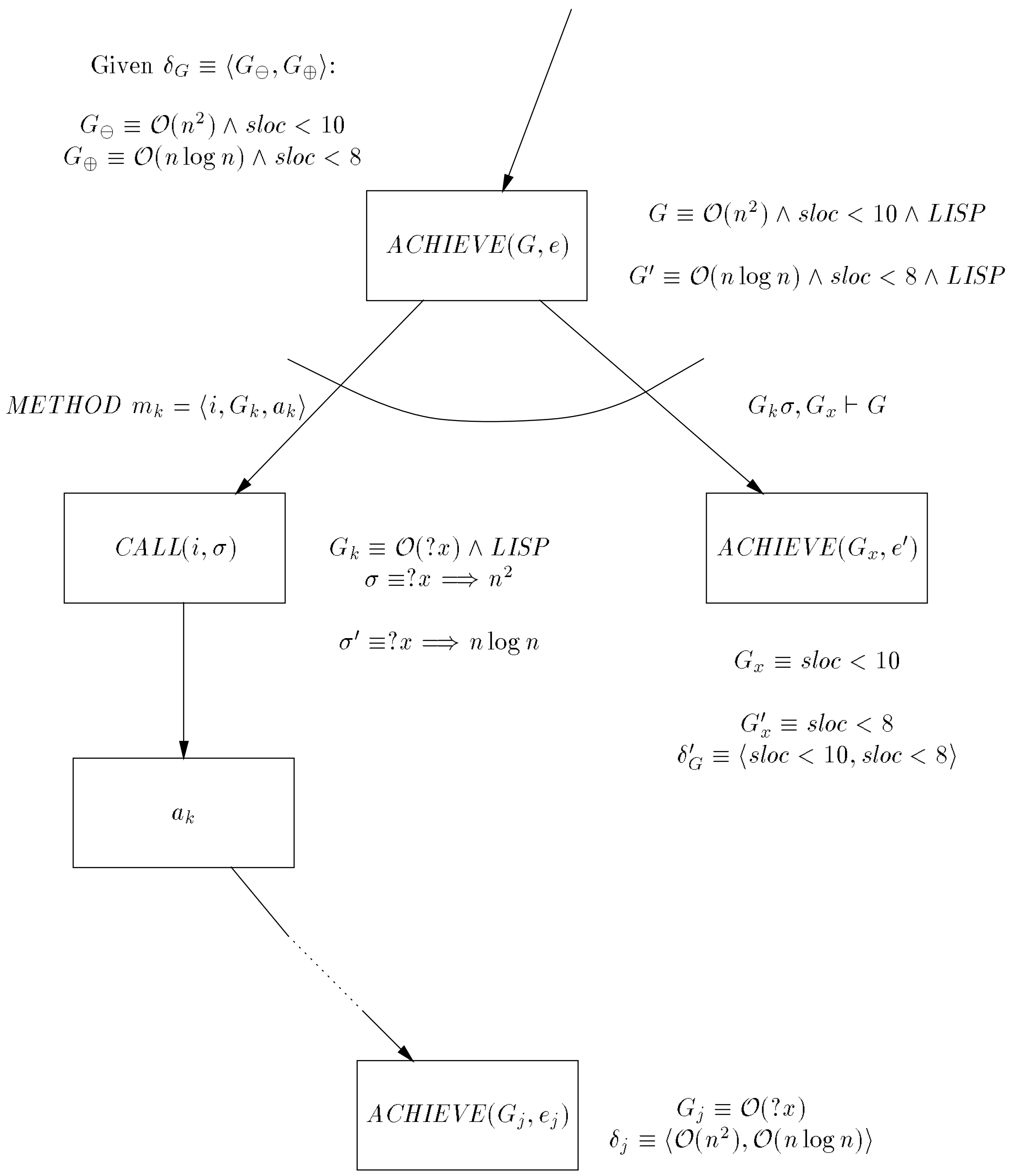

Given $\delta_G \equiv \langle G_\ominus, G_\oplus \rangle$:
$G_\ominus \equiv \mathcal{O}(n^2) \wedge sloc < 10$
$G_\oplus \equiv \mathcal{O}(n \log n) \wedge sloc < 8$
$ACHIEVE(G, e)$
$G \equiv \mathcal{O}(n^2) \wedge sloc < 10 \wedge LISP$
$G' \equiv \mathcal{O}(n \log n) \wedge sloc < 8 \wedge LISP$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

Figure 8.9: Propagating δ_G from root to leaves of design history

As we walk down the design history, at each $ACHIEVE(G, e)$ agenda item (see Figure 8.9), we must do the following:

1. Revise the $ACHIEVE(G, e)$ agenda item to be $ACHIEVE(\delta_G(G), e)$
2. Decide if $CALL$ing method i is still useful as a means of decomposing the revised goal
3. If not, prune the nonprocedurally generated subplan
4. If still useful, determine the changes induced by δ_G and propagate those into both subplans. Propagation should be into earlier subplans first to minimize propagation into subplans that become useless.

Determination of the continued utility of the method $m_k = \langle i, a_k, G_k \rangle$ requires that we re-validate the goal decomposition process. If we cannot re-validate the goal decomposition, then the nonprocedurally generated 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.

A successful re-validation of the decomposition requires verification that $\exists \sigma', G'_x : G_k\sigma', G'_x \vdash \delta_G(G)$. Given such a re-validation, we:

- revise $CALL(i, \sigma)$ to $CALL(i, \sigma')$
- propagate σ' into the subagenda a_k under the $CALL$.
- recursively propagate a new $\delta'_G : G_x \Longrightarrow G'_x$ into the subagenda $ACHIEVE(G_x, e')$, using this same procedure

Propagation of δ'_G can stop if δ'_G can be determined to be an identity. Propagation of σ' can terminate when $\sigma' = \sigma$.

Propagating σ' into the subagenda a_k requires

- propagating σ' into subagendas for all actions a_j such that $a_j \in a_k$ and $action(a_j) = ACHIEVE(G_j, e_j)$:
  1. computing a new performance delta $\delta_j : G_j\sigma' = \delta_j(G_j\sigma)$
  2. recursively propagating δ_j into the design history at a_j
- adjusting other agenda items derived from a_k that are affected by σ', such as $REQUIRE(G_j, e_j)$, etc. This may require re-validating performance predicates.

Such agenda items are easily found because the structure of the design history is a direct reflection of the structure of method body a_k.

In general, verifying the existence of a new σ', G'_x and derivative δ'_G and various δ'_js is very hard; it depends on the semantic relations between the various types of performance goals. We would need a description of this semantic relation and a theorem prover to do this computation. However, our assumption that G_{rest} is conjunctive in nature, and a further assumption that the performance predicates are totally unrelated means that much of this can be treated as a problem in manipulating sets of independent predicates. Much of the computation then consists of shunting subsets of the components $G_\ominus, G_\oplus$ of the performance delta $\delta_G = \langle G_\ominus, G_\oplus \rangle$ to the right place. Figure 8.9 shows how this occurs for a specific case.

8.7 Integration of Performance Bound Deltas Δ_v

Performance bound deltas δ_v are simply a special case of Δ_G, and so essentially the same integration procedures can be used. When revalidating performance goals, additional information about the relation $(\succeq)$ of the old and new performance bounds can make this process easier, by taking advantage of:

$$v_1 \succeq v_2 \supset (p(f) \succeq v_1 \vdash p(f) \succeq v_2)$$

Tighter performance bounds make it less likely that a performance goal will continue to be successful, while looser bounds tend leave its success unchanged.

In Figure 8.9, if $G_\oplus$ was $\mathcal{O}(n \log n) \wedge sloc < 12$, then we can use the fact that $10_{sloc} \succeq_{sloc} 12_{sloc}$ to determine that the original G_x is satisfactory even for the changed performance specification, and so we need not propagate any δ'_G into the subagenda under G_x.

8.8 Integration of Method Deltas $\Delta_{\mathcal{M}}$

Integrating a method delta $\delta_{\mathcal{M}}$ requires handling each of its aspects (Section 6.4.4):

- $\Delta_{\mathcal{M}\ominus}$: removal of agenda items produced directly or indirectly by deleted methods
- $M_\oplus$: revising the method library to make new methods available to the repair process
- $\Delta_{\mathcal{M}postcondition}$: checking that invoked method postconditions are still valid
- $\Delta_{\mathcal{M}action}$: revising bodies of invoked methods

Handling $\Delta_{\mathcal{M}\ominus}$: We mark as *undesirable* all invocations in the design history of the deleted methods; these are agenda items whose *action* field says $CALL(i, \sigma)$ for all $i \in \Delta_{\mathcal{M}\ominus}$. An eventual pruning process will remove the $CALL$, all the agenda items introduced by such called methods, and will also prune upwards until some alternative is found to the invoking plan.

Handling $\Delta_{\mathcal{M}action}$: For each method i with an action revision δ_f ($\langle i, \delta_f \rangle \in \Delta_{\mathcal{M}action}$), we do the following:

- immediately prune the sons of agenda nodes whose *action* is $CALL(i, \sigma)$. If we were to simply mark the sons *undesirable*, a later pruning process would prune away too much: the $CALL$ nodes that invoked the body.
- mark such $CALL$ nodes as *incomplete* (safe and appropriate because its sons have been removed)
- revise the method library element for i

This ensures that the old plan for method i is removed from the design history; the plan repair process will install the revised method body when it eventually repairs the incomplete $CALL$ nodes.

An idea we have not pursued is the possibility of using the transformation component of $\Delta_{\mathcal{M}action}$ that revises the method body to guide the direct revision of the design history at every point where the method was invoked. The similarity of the structure of method bodies in terms of actions, and the corresponding structure of agenda items which act as instances of the method execution are what gives this idea promise; the payoff would occur in the avoidance of re-doing work generated by agenda items representing the leaves of the method.

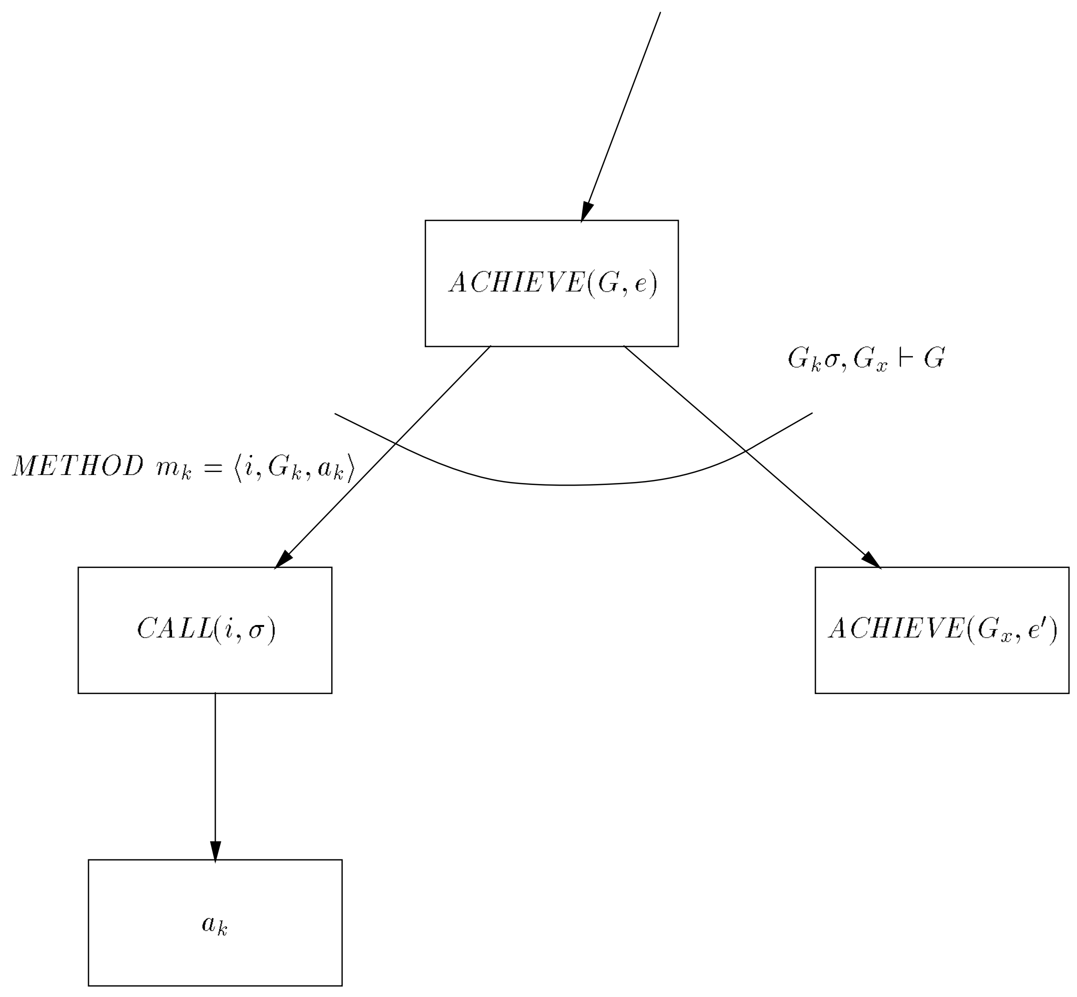

$ACHIEVE(G, e)$
$G_k\sigma, G_x \vdash G$
$METHOD\ m_k = \langle i, G_k, a_k \rangle$
$CALL(i, \sigma)$
$ACHIEVE(G_x, e')$
a_k

Figure 8.10: Nonprocedural invocation in design history

Handling $\Delta_{\mathcal{M}postcondition}$: For each method i with a postcondition change δ_G ($\langle i, \delta_G \rangle \in \Delta_{\mathcal{M}postcondition}$), we check that nonprocedurally invoked instances of method $\langle i, a_k, G_k \rangle$ are still valid and adjust the subagenda according to changes induced by δ_G; those which are no longer valid have the corresponding $CALL(i, \sigma)$ agenda item marked as *undesirable*. In particular, the design history is searched for agenda item complexes of the form shown in Figure 8.10. For each such complex, the goal decomposition process used by $ACHIEVE(G, e)$ is retried to verify that the revised method postcondition $\delta_G(G_k)$ is still effective. If $\exists \sigma', G'_x : \delta_G(G_k)\sigma', G'_x \vdash G$ can be satisfied, then a revised $\delta'_G : G'_x = \delta'_G(G_x)$ can be propagated into the $ACHIEVE(G_x, e)$ subagenda, and σ' can be propagated into the $CALL(i, \sigma)$ subagenda using the procedures outlined for propagation of δ_G into subagendas in Section 8.6. Failing to find a decomposition for the revised postcondition $\delta_G(G_k)$ tells us that the method is no longer applicable; we simply mark as *undesirable* the $CALL(i, \sigma)$ and the $ACHIEVE(G_x, e')$ nodes. An eventual pruning process will prune back to the parent $ACHIEVE$, and plan repair will find a new replacement.

To minimize wasted effort caused by propagating performance specification changes into changed method bodies, $\Delta_{\mathcal{M}action}$ should be processed before $\Delta_{\mathcal{M}postcondition}$.

8.9 Integration of Library Deltas $\Delta_{\mathcal{G}}$ and $\Delta_{\mathcal{P}}$

The support deltas:

- $\Delta_{\mathcal{P}}$: Change of performance measurement functions
- $\Delta_{\mathcal{G}}$: Change of performance predicate library

change definitional aspects of the performance predicates usable in performance goals.

Changes which delete performance measurement functions or performance predicates will affect the design history indirectly, through the consistency requirements on composite deltas. If a performance predicate is no longer available, then the consistency requirements demand that methods which reference that predicate be modified or deleted; the design history will be purged of references to that predicate by some $\Delta_{\mathcal{M}}$ that modifies or deletes methods that referenced the deleted predicate. Similarly, added performance measurement functions or performance predicates can only affect the design history via method deltas.

Changes which replace performance measurement functions or performance predicates necessitate the re-validation of dynamic postcondition checks that use the replaced functions/predicates. Such re-validation occurs automatically when a functional delta is applied, or a dummy *BATCHBANISH* is applied in place of a functional delta, so no additional action is needed to handle such a delta.

8.10 Integration of Range Deltas $\Delta_{\mathcal{V}}$ and Order Deltas $\Delta_{\succeq}$

The support deltas:

- $\Delta_{\mathcal{V}}$: Change of range of performance values
- $\Delta_{\succeq}$: Change of orderings $\succeq_i$

change only the outcome of evaluations of performance goals. Re-validation of dynamic postconditions is sufficient to handle these. As with $\Delta_{\mathcal{G}}$ and $\Delta_{\mathcal{P}}$, such re-validation occurs automatically.

8.11 Processing order of Deltas

A composite delta given to a Design Maintenance System consists of a set of at most one delta of each type, as described in Chapter 6. The individual deltas that make up a composite delta should be processed according to the following partial order to ensure that consistent changes are made:

$\Delta_{\mathcal{V}} > \Delta_{\mathcal{P}}$	Change range before measures
$\Delta_{\mathcal{V}} > \Delta_{\succeq}$	Change range before ordering
$\Delta_{\mathcal{P}} > \Delta_{\mathcal{G}}$	Change measures before goals
$\Delta_{\mathcal{G}} > \Delta_{\mathcal{M}}$	Change goals before methods
$\Delta_{\mathcal{C}} > \Delta_{\mathcal{M}}$	Change transforms before methods
$\Delta_{\mathcal{M}} > \Delta_{G}$	Change methods before performance
$\Delta_{\mathcal{M}} > \Delta_{v}$	Change methods before performance
$\Delta_{\mathcal{M}} > \Delta_{f}$	Change methods before functionality

The general rule is, if concept a is used as a building block for concept b, then changes to a must be processed before changes to b: $\Delta_a > \Delta_b$. A topological sort of this ordering, usable as an order to process the deltas, is:

$$\Delta_{\mathcal{V}} > \Delta_{\mathcal{P}} > \Delta_{\succeq} > \Delta_{\mathcal{G}} > \Delta_{\mathcal{C}} > \Delta_{\mathcal{M}} > \Delta_{G} > \Delta_{v} > \Delta_{f}$$

Since the delta integration procedures consist generally of Mark, Prune, Revise and Repair, we can generally perform just the Mark and Revision portions for each delta according to this ordering. Pruning and Repair are common to all the deltas, and can be generally delayed until Pruning and Revision for all delta types has been completed. This observation motivates the sequence of the design repair process shown in Figure 1.13. The final *BATCHBANISH* is not required if a δ_f is integrated.

8.12 Summary

This chapter has outlined procedures for integrating various types of maintenance deltas into a design history. Such integration procedures typically mark portions of the design history that are inconsistent with the delta, prune away the inconsistent parts and their dependents, and then repair the remainder of the design history by executing incomplete agenda items. Completing agenda items often requires inserting new transformations into the derivation history and further adjustment to the design history. We outlined delta integration procedures for every type of delta defined in Chapter 6, both for changes to the specification, and changes to the support technology used to implement the specification.

The existence of such procedures to revise the design history depends on the presence of an explicit maintenance delta. The maintenance delta guides the process of revising the design history, providing us with a considerable portion of the mechanisms necessary for a Design Maintenance System. Since the procedures are insensitive to whether the design history is complete or not, one can apply deltas using a Design Maintenance System to a partially completed implementation. 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 Design Maintenance System would allow us to perform transformational maintenance relatively cheaply, and leads towards the goal of an Incremental Evolution lifecycle.

Chapter 9
Related Work on Maintenance Systems

Chapter summary. We compare our work to a number of other systems that revise results.

The major focus of this thesis is on design maintenance systems (DMS) for software artifacts. It is built on foundations consisting of a transformation system model and a control language for guiding transformational implementation. We have discussed related work on those topics in their corresponding chapters. In this chapter, we consider work related to systems that repair or maintain various structures, especially software.

Such maintenance systems can roughly be classified as follows:

- Informal software maintenance systems
- Specification recovery
- Reuse of control knowledge for transformation system
- Derivation replay based on transformation system
- Plan reuse and repair
- Truth Maintenance Systems

Informal maintenance systems are those for which no formal software construction model exists, including the widespread ad hoc conventional practice. The rest of the maintenance systems we discuss can be cast as transformation systems according to the model presented in Chapter 3. Maintenance by specification recovery essentially abstracts a concrete program to allow changes to the abstraction rather than the code. Maintenance by reuse of control knowledge simply re-runs the transformation system on the modified specification; the control knowledge used in the previous implementation is available for the new implementation. Derivation replay attempts to avoid direct use of the control knowledge by re-applying decisions resulting from application of control knowledge in the previous implementation. Plan reuse tries to reuse the construction plan of the previous implementation. Truth maintenance systems update derived inferences computed from a set of premises when some premise changes.

We will consider systems of each type in the order given above. We summarize each work, and compare it to ours along the following dimensions:

- Representation of Artifact Goal and Design States
- Capture of control knowledge
- Content and Representation of captured design history
- Notion of change: informal or formal
- Maintenance method

9.1 Informal Software Maintenance Systems

In this section, we examine ad hoc maintenance, the widely used tool Make, some design recording and design recovery systems. The lesson from informal maintenance is just what we expect; the design is needed. Work on design recovery is a natural follow-on. Not losing the design in the first place was our starting assumption.

9.1.1 Ad Hoc Maintenance

Maintenance as practiced by the masses is indeed a sorry state of affairs (perhaps, partly, because the development process is also such a sorry affair). Most maintenance is done by simple editing of source files followed by recompilation and ad hoc testing. In this arena, there is no formal notion of a specification, functional, performance, technological or otherwise. At best, an informal description of the desired artifact exists, usually out of date and too abstract to cover many low level details. Consequently, no explicit design is present, tying specification to implementation. If anything resembling a design remains, it is perhaps some diagrams representing the high-level data-flow architecture of the program. The implementation itself is full of consequences of particular implementation methods for the various program purposes, so that intent is swamped under technological detail. To aggravate the problem, many applied implementation technologies are suboptimal, wrong, or coded clumsily; dead code from previous maintenance exercises swells the volume. The only object in the design history, if it could be said to exist, is the final design state, i.e., the source code.

The representation of a desired changed is usually informal and sometimes only a verbal sketch. Faced with a desired change, maintainers must induce an informal specification for the program by looking through a sea of code (or remembering the informal specification used during development!). A significant part of a maintainer's time is spent trying to understand what the code does, how it is related to the problem at hand, and the impact of a possible change. Since the process is manual and the understanding fuzzy, it is consequently error-prone and the "maintained" program often requires debugging. Knowledge acquired by one maintainer is useless to the next, as all of the knowledge required ends up in each individual maintainer's head.

One can draw a motto from this:

To lose the design, give your programmer a text editor.

All of these defects were fundamental motivations for transformational maintenance. Formal specifications provide unambiguous intent. Validated formal transforms ensure that correct implementations result from the specification. Libraries of methods allow designers to implement by choosing from tested implementation technologies whose performance level is known. A design history capturing the relation between performance goals and methods for achieving them capture the relation of the specification to the code. Explicit specification deltas guide the revision of the design history, producing a revised design history for the next maintenance step. Impact analyses could be performed given the desired delta; semi-automatic installation of the change is managed by a DMS. New implementation technologies can be added to the library by maintainers, and are available for use by the next maintainer.

9.1.2 UNIX Make

The UNIX tool, Make, is used to automate the construction of complex software systems from source files, that require compilations, link-edits, and other sundry result assembly operations. It is especially designed to optimize the construction process after a change has been made to one of the original source components. It is thus an "efficiency hack" in the same sense that a DMS is an efficiency hack: neither is technically necessary. But Make has shown its practicality in everyday software development (read "maintenance") environments, because most changes only affect a small portion of the code [LS80]; consequently, only 20% of a system is typically recompiled by Make after a change [LQ89], which provides considerable savings in computation energy and on-line waiting time. This is precisely the same argument we make for the utility of a DMS: changes will affect only a small part of the implementation, and consequently will require only a small amount of energy to implement.

Make assumes components (and intermediate results, such as relocatable files produced by compiling) of a software system reside in accessible disk files. It requires a system implementer to explicitly state which files depend on which others, and how to regenerate the content of a file when one of those on which it depends has changed. This collection of dependencies and regenerators is in effect a dependency net with very large grain operators. Once a maintainer has modified some set of files, he invokes Make on the dependency net description; Make determines which files need to be rebuilt by examining file date stamps (if $DATE(filea) < DATE(fileb)$, and $DEPENDSON(filea, fileb)$, then $filea$ needs to be rebuilt). The dependency net avoids both the "accidental" ordering that using a linear history would induce, as well as avoiding unnecessary work.

For Make, the functional specification of the desired artifact is captured in the collection of source files known to the dependency net. Such functional specifications are typically low-level program source code as opposed to abstract specifications, so much of the design, i.e., those decisions that went into producing the source code, is already lost. A weak kind of performance specification is defined procedurally by the sequence and parameters of the large-grain operators (i.e., is the compiler invoked with optimization enabled?) in the dependency description. The control knowledge to implement the artifact is also encoded in the dependencies. Design states are approximated by up-to-date sets of intermediate files forming a consistent frontier of the dependency net; the final design state represents the constructed artifact. This is very similar to notions of state used in nonlinear planners. The intermediate files coupled with the dependencies description could be treated as a sort of design history; they capture consequences of design selections. The only notion of change that Make understands is implicit: an out-of-date or missing file. These changes are caused by a programmer using an editing, moving, or deletion tool on a file. When such a change is detected, the dependency net is consulted to determine what operator to apply.

A DMS uses an explicit formal delta to determine what changes initially, and the design history to determine what is indirectly affected, so it does not need date stamps.

Unlike Make, a DMS can repair a failed regeneration step by virtue of having access to the goal structure which generates the individual transformations, and alternative methods for achieving such goals. A DMS can preserve a transformation step that follows a failed step by commuting them; Make simply aborts the regeneration process. Make also only considers data flows, not actual transformations; as a consequence, only objects statically identified by the programmer are traced. A DMS generates design decisions by executing TCL methods, and records the resulting history automatically. Errors in a Make dependency net lead to incorrect re-generation of the product. Errors in TCL methods lead to the same problem, but are presumably less likely because such methods may be used by many projects and are probably better tested; further, once detected, Design Maintenance System can help correct the resulting artifact.

9.1.3 Code understanding as a maintenance prerequisite

Soloway [SJ85] suggests that a significant problem in conventional maintenance is that of understanding how existing programs work. A particular problem is that of *delocalized plans* [LS86]: sections of program text that are widely separated in the linear textual version of the program, that must work together to accomplish a particular effect. Trivial examples of this include "accumulator initialization" before a loop body, and "accumulator summing" buried deep in the loop; together, these two fragments constitute a "sum-values" plan. Human maintainers have trouble when one part of a plan needs modification, and the existence of the other part is not obvious, has been forgotten (due to the complexity of the plan), or is not easily found (due to textual gap between the parts). Soloway suggests that informal text pointers be embedded in comments at each plan fragment that point to the other fragments to make the other parts findable.

Soloway does not address how the maintainer knows *what* a program is supposed to do (no artifact goal) or how it achieves those purposes (no design). There is no design history; only the final code produced as a consequence. Since Soloway's emphasis is on understanding as a prerequisite to change, there is no discussion of change, change representation, or installation of change.

Our DMS requires use of a formal specifications to encode program intent. The design history captured during the implementation process provides traceability from the specification to the implementation, providing a connection from *what* to *how* with intervening plan structures. While we do not specifically use this information for this purpose, it is available for the maintainer to help him understand how the existing program achieves its purposes.

9.1.4 Informal Design Capture

Since understanding is so important for conventional maintenance, Wild [WML$^+$89] suggests that a hypertext [Con87] network be used to capture the decisions leading to source code from the requirements, in a GIBIS-like network [CB88]. Nodes in the network represent decisions. Each decision has considered alternatives recorded, to document dead-ends and to provide hints about later possible improvements. The content of hypertext nodes is text, and relations between nodes are established by the software designers. It would appear to be very difficult to ensure that this design document captures the necessary dependencies, captures the necessary information, and stays synchronized with the actual code.

With a DMS, we capture such decisions in the design history. Alternatives are stored explicitly as agenda items representing *OR* or *ELSE* alternatives, and implicitly as untried methods. Our DMS does not store relative merits of alternatives.

9.1.5 DESIRE: Design Recovery

Design recovery is intended to recapture design information used to generate the source code to be maintained. It does not help with installation of changes; rather, the recovered design information is intended primarily to help the maintainer understand the existing code.

MCC's DESIRE system [Big88, Big89a, Big89b] is intended to help maintainers understand code by hueristically matching it against a database of "design structures", and pointing out which parts of the code match individual structures. The database is organized around conceptual abstractions of software engineering construction technologies, such as "window management", "process dispatching", etc. Each abstraction has associated with it a set of expectations; for window managers, the notion of window, window-update, etc., are expected. DESIRE uses the textual names referenced by the program as an index into a database, either directly to the abstract concepts, or indirectly to an abstract concept via a related expectation. Access to an abstract concept leads to a set of expectations which can be checked against the code. Presence of such expectations simultaneously confirms the concept and clarifies which parts of the code perform related functions; this is similar to the effect Soloway tries to achieve by manual means. Absence of confirming expectation is an indication that term used is either ambiguous or the technology to which it corresponds is not in the database (which provides opportunities to augment the database). It is not clear how much of the DESIRE matching process is automated, although [Big89b] describes a neural-net recognition mechanism. The final result of the DESIRE mechanisms are a Prolog-style database capturing relationships between objects in the design and code, and hypertext-style graphic aids using that database to inspect existing code.

DESIRE is intended to help understanding, and indirectly populate reuse databases, but does not address the problems of installing changes as does a DMS.

9.2 Specification Recovery

The simplest model of maintenance in a transformational context is to modify the specification and re-implement [Bal85a]. Of course, to do this, one must actually *have* a specification. One can treat the implementation source code as an extremely low-level specification, but then no leverage is gained over conventional maintenance. The DESIRE system abstracts the source code somewhat. Carrying this abstraction process to the extreme can allow one to recover a specification at a truly abstract level, where it is easier to make changes, or at least understand the intent of code implementing the abstraction. We call this *specification recovery*. With a full transformational maintenance model, one simply never loses the specification, and so specification recovery is unnecessary.

By simply recording the sequence of abstracting steps, a reverse-chronological derivation history could be trivially obtained and reused by a tool like our DMS, but often specification recovery tools do not do this.

Our DMS differs from these tools in that rather than re-discovering a specification and/or a derivation history, we simply don't lose them in the first place. As much of the original design is preserved as possible as opposed to reimplementing the abstract program from scratch. None of these plan recovery systems recapture any performance goal information, or aid in the installation of changes.

9.2.1 Transformational Model of Maintenance

Recovery of abstract functional specifications from source code is described by a *Transformational Model of Maintenance*, or TMM [ABFP86]. A domain-oriented, transformational software construction model is assumed, even for programs not constructed transformationally.

The result of the process is a more abstract specification of the existing code and the set of transformations used to implement that abstract specification. The process consists of an analyst manually guessing the abstractions and their implementing transformations, and applying a tool that converts matched implementation fragments in the code back to their abstractions. This work also discussed forward engineering from the abstract specification to accomplish change in support technology, functionality, or performance, providing the overviews given in Chapter 7, and leading to this thesis. The work describes a successful program porting project based on these ideas; this amounts to a change of support technology by revising the set of available transforms.

The TMM work characterizes maintenance in terms of a design space, and discusses choosing different paths through that space, but does not take the logical step of actually recovering the derivation history of an implemented artifact and then revising it to obtain another. Neither explicit performance goals or change representations are provided. There are early hints of the ladder-construction process for integrating functional deltas that was presented in Chapter 7, but in this thesis we have produced theories and procedures for actually accomplishing it.

9.2.2 Program Recognition

There are a number of similar systems which try to recognize program structures; this is effectively the extraction of a more abstract functional (usually even procedural) specification. Such systems match code against libraries of program fragments in an attempt to recognize abstraction function specifications. These systems differ from DESIRE by being more formal in their approach.

PROUST [SJ85], [Joh86] matches small Pascal programs against their intentions coded procedurally; close matches are tolerated and the difference is diagnosed as a program bug. PAT (Program Analysis Tool) [Nin89] uses a forward-chaining inference engine to heuristically combine recognition of program fragments into recognition of more abstract specifications, such as determining that a loop containing an exchange of two data elements based on the results of their comparison is a *BUBBLESORT*.

Dudu [All90] re-validates canned proofs to prove that several program fragments work together to accomplish an abstract effect. Our definition of TCL assumes that proofs are associated implicitly with methods; DuDu shows the value of associating an explicit proof with a method.

The *Programmer's Apprentice* (PA) [RW88] is representative of these systems. PA is a system of tools to help a programmer construct and modify programs in conventional programming languages such as Ada or Lisp. PA maintains a database of *cliches*, which are transformations mapping abstract programming tasks (functional specifications) into more concrete code. Most such cliches are related to programming domain knowledge, as opposed to problem domain knowledge. The implementation part of such cliches are matched against programmer-selected portions of existing code to partition it into understandable chunks represented by the functional specifications; Wills [Wil87] describes a graph-matching mechanism to actually perform this. It is not clear what happens when cliche implementations conflict or share parts. It is also not clear if PA records the reverse derivation history of code to abstract code. Tools are provided to allow abstract specifications to be inserted in a program, and then transformed into actual code. Changes are made not to program implementations, but rather to abstracted programs themselves. The change to be made is not stated explicitly, but an installed abstract change can be semiautomatically implemented by transformation.

9.3 Control Knowledge Reuse for Reimplementation

The utility of reusing transformational control knowledge to reimplement a modified specification is a simple consequence of the value of control knowledge in driving the transformation system in the first place. The need for capturing control knowledge was the initial motivation for TCL. We can divide control knowledge into two types: that which is generally applicable, and that specific to implementing the specification at hand. It is clear that generic control knowledge should always (be available to) be reused in reimplementing any specification; this is why our model of a transformation system has a method library. Control knowledge useful for the specification at hand appears valuable for reimplementing a slightly changed version of the specification on the grounds that we don't expect the implementation to change much, and therefore the generation process must be similar.

Several problem stem from using just the control knowledge without a design history for maintenance purposes:

- the assured cost of complete replay
- unnecessary expense of resolving choices left by metaprogram
- inability to use a maintenance delta to revise implementation

The first problem is the requirement for complete re-execution of the metaprogram every time a change is made. We described the high cost of transformational implementation in Section 3.3. In the face of the small-change/small-effect assumption, this seems an unneeded expense. In the face of scale, even practical software development processes give up this assumption, partitioning systems into modules partly to keep the compilation costs reasonable, providing dependency-net management tools such as Make for assembling modules.

Retention and repair of the design history has the potential of allowing most of the metaprogram re-execution to be avoided. This would be fruitless if the entire design history must also necessarily be scanned (as in fact our design history procedures pretty much do during *BANISH* and/or ladder building), but with the aid of a nonlinear dependency net on transformations there is some hope of avoiding this.

The second problem has to with the availability of certain types of choices in the design space: how are subproblems decomposed, and where should transformations be applied? With a goal-oriented metaprogramming language such as TCL, a performance goal may be decomposed in many possible ways, of which only a few at best are actually useful for the specification at hand. Similarly, applying a transformation must choose an appropriate locater from within the allowable locale. The control knowledge does not necessarily contain this information, especially if it generic. The actual choices made are precisely what is captured by the design history. When replaying just control knowledge, such choices must be resolved again. Our methods for integrating a delta into a derivation history in the common case go through effectively zero effort to "pick" a locater; it is already recorded.

One can specialize the control knowledge to the particular specification being implemented, to constrain decompositions and locales until this control knowledge approximates a design history; as an extreme, one can consider the design history we have defined as a fully specialized metaprogram. If we assume that human agents generate such control knowledge (metaprograms), it very expensive to generate such specific knowledge; further, we run the risk of specializing the control knowledge so much that it no longer applies when we change the specification.

Lastly, given just a changed specification and control knowledge, what can we do to take advantage of any knowledge of the change? Our DMS procedures for delta integration into a design history provide us with concrete methods for taking advantage of such knowledge. We consequently think that *one should have both the design history and its generator*: the design history to cache low level details about what precisely was done, and the generator to use when repairing the design history after parts inconsistent with the delta are stripped away.

9.3.1 PADDLE: A Metaprogramming System

The TI system for constructing software transformationally [Bal85a] represented design states as functional specifications written in the GIST specification language [BGW82]; no performance specifications exist. PADDLE [Wil83] is a procedural metaprogramming language to guide the transformational implementation process. The operation of the PADDLE was described in Section 4.3.3. It was suggested PADDLE be used for replay purposes after specification was changed, as well as for initial development. In practice, the PADDLE program would be changed in parallel with the specification, and then replayed [Bal85a]. No explicit notion of change is used.

The model of replay assumed by the PADDLE design is complete re-execution of the metaprogram. There are several problems specific to PADDLE:

- Early abort prevents maximal replay
- Mis-applied transformations during replay
- Unnecessarily limited ability to repair failed plans

Early abort prevents maximal replay

The purely sequential execution model used by PADDLE aborts execution of the metaprogram at the first sign of trouble, usually on encountering a transform which cannot be applied. It may consequently stop replaying when there remains a significant number of still-reusable design steps. The plan repair performed by DMS retains all steps not obviously invalidated by the specification delta, by *BANISH*ing transformations that no longer apply and rearranging the design history accordingly. This is only possible because TCL explicitly allows nonlinear sequencing of plans, and because the DMS understands how to reorder transformations (*DELAY*).

Misapplied transformations during replay

A PADDLE metaprogram that successfully implements an initial specification need only choose correct locaters for applied transforms for the specific implementation being attempted. Each locale constraint need only choose a unique locater for that particular specification. In a changed specification, the same locale constraint may be ambiguous, and so replayed transforms may be applied in inappropriate places. Wile [Wil83] notes this shortcoming, and suggests that a richer language is needed in which the locales can be more accurately expressed. TCL does not directly provide a richer constraint language (although we have identified useful operators for such in Section 4.2.1), but does compute replacement locaters from deltas and the design history that cause at least equivalent local effect. Further, since our delta integration mechanisms re-validate replayed transforms, if one should get misapplied, it will be detected and *BANISHed*. Such re-validation is only possible when performance goals are available to be checked.

Limited ability to repair broken plans

PADDLE metaprograms do not have performance goals attached to plan steps. Consequently a failing metaprogram plan cannot be replaced by another plan that achieves the same effect, precisely because no other plans are labeled with information describing their effect. TCL metaprograms, in contrast, have explicit post-conditions describing performance goals, and failure by one method to achieve an effect can be alleviated by backtracking to another method with a post-condition achieving a similar or stronger effect. Thus TCL contributes to the repairability of a failed plan.

9.3.2 Glitter: A Goal directed Metaprogramming System

For the TI system, the difficulties caused by the absence of goals in PADDLE were recognized:

> "... a major impediment is the operational rather than specificational nature of PADDLE..."

[Bal85a]

The Glitter system [Fic80, Fic82, Fic85] is also a metaprogramming system operating on GIST specifications. Its operation is described in Section 4.3.3. Glitter provided "transformational" goals, which can be considered process and/or performance goals, and has a TCL-like style. This is an improvement over PADDLE, but it does not appear that Glitter was ever considered for use in a maintenance context. If it had, it would have suffered the problems of replay of a pure metaprogramming system as outlined earlier.

9.3.3 Hueristic Plan Repair

LP [Sil86] is system for learning control knowledge for solving algebraic equations. It assumes an underlying algebra solving system, PRESS, implemented as a transformation system with control mechanisms as described in Section 4.3.3. PRESS control knowledge consists of TCL-like methods containing plans with postconditions (performance goals such as "number of occurrences of variable X"). LP is presented with a problem-solution consisting of a sequence of algebraic formulas. From this sequence of formula-states a derivation history can be trivially extracted by comparing successive pairs of states. LP explains the derivation history by partitioning it into sections that achieve interesting effects such as leaving an equation factorable (a performance goal). Essentially a sectioned sequential plan with goals for each section is learned. This corresponds roughly to the recursive TCL structure

$$ACHIEVEBY(G_{solved}, e, SEQ(ACHIEVEBY(G_{sectiongoal}, \ldots), \ldots))$$

LP is doing more just design recovery, because it augments the support technology.

One can treat LP as a maintenance system by presenting it first with an algebra problem and its solution, and then presenting it with a slightly changed algebra problem (an implicit functional delta). While learning new methods is certainly an interesting way to improve a system's problem solving ability on slightly different problems, it is not our concern here. Rather, we are interested in how LP handles failures in a learned plan when applying it to the modified problem.

A learned plan is executed sequentially. At each plan step, the goal for the section is tested, and the balance of the section is skipped if successful. This allows serendipitously accomplished steps to be ignored. If the section goal is still false, the plan step itself is executed, and if successful, the next step is tried. If the plan step fails, other methods with identical postconditions of the failing plan step are tried until one is successful, at which point the next step is tried, or all fail. Should all methods fail, then LP attempts to find a method whose postcondition satisfies the preconditions of the failing plan step, and whose preconditions match the current state; this has the effect of dynamically inserting repair steps. Should this fail, LP gives up. All of this logic is wired into LP's control mechanism.

Our DMS operates differently. Steps carried out by favorite plans with fallback plans are captured neatly by the TCL *ACHIEVEBY* action, and only a small amount of hardwired control mechanism is associated explicitly with it. Unlike LP, TCL will not leave out a plan step, because we assumed that TCL methods capture precisely what is needed to accomplish its postcondition in a provable way; optional steps can be easily described by *ACHIEVE* goals that are sometimes achieved serendipitously. Similarly, TCL execution does not dynamically insert steps; need for this is an indication of a missing *ACHIEVE* goal.

9.4 Maintenance via Derivation History Replay

The idea of reuse of design decisions is not new. To do so with automated help requires some sort of formalization of the software process, which is invariably chosen to be transformation systems. Considering the utility of the state space model, and the broadness of the transformational model we formalized in Chapter 3, this is also not a surprise. A number of systems using these ideas have already been built, and we cover them in this section.

What distinguishes our work from that presented in this section is:

- A transformation system model including explicit performance specifications
- A nonprocedural transformation control language in which all the control knowledge is explicitly stated
- A design history that captures relations between plans and performance goals
- A formal notion of delta
- Procedures for revising the design history controlled directly by an explicit delta
- Methods for revising derivation histories based on commutativity
- Re-ordering of history, and modification of replaced transformations (cf. *DEFER* and *PRESERVE*) dependent on the actual delta.

We make these remarks here to avoid repeating them in each subsection.

In particular, we want to emphasize that *naive replay* (simply re-applying still-legal decisions, skipping now-illegal decisions from a history) is relatively easy but suspect without revalidating that those replayed decisions serve a useful purpose according to the (possibly-revised) performance specification.

9.4.1 Replay in SINAPSE

The SINAPSE transformation system [DKMW89] replays *critical* design selections. A "specification" for SINAPSE consists of a functional specification for solving differential equations, and a set of named design choice, named design selection pairs which are used to guide the refinement of the functional specification; such pairs act as a kind of partial performance specification. A built-in control mechanism makes most of the design selections either by default or by selection of a domination selection according to hardwired criteria which effectively act as a procedural encoding of the rest of the performance specification. When a design choice is proposed whose name matches that of some member of the "specification", then SINAPSE chooses the corresponding named selection from the "specification". By changing the decision choice/selection pairs in the "specification", different implementations of the functional specification are produced. Design choices stated in the "specification" which are not encountered during the implementation process are simply ignored. In practice, the SINAPSE system takes only a few minutes to go through the entire refinement process, and so it is practical to change the functional specification and rerun the transformation process; this appears to be caused by limited design selections. This aspect of SINAPSE treats the specified performance specifications as a kind of constrained metaprogram.

There is discussion of actually storing a tree of choice/selection pairs, with states containing functional specifications attached to each instantiated branch. Change of performance is accomplished by pointing to a choice in the tree, forcing another selection, and restarting the transformation system on the design state corresponding to the branch leading to the selected choice, dynamically generating new choices and selections to reimplement the balance of the specification. The history replay mechanism has been used to effectively produce a decision tree on a small number of design decisions, leading to alternative implementations. SINAPSE has successfully synthesized different practical finite differencing codes for a real application using such histories to guide the process.

9.4.2 Cheatham's Program Development System

The Program Development System (PDS) [CHT81, Che84] was used to transformationally implement and maintain sizable programs, including targeting a network communications system for 4 different machine architectures and porting the EL1 compiler. PDS maintained a derivation history of the modules of system as described in Section 5.4.4, similar to that maintained by Make. Maintenance consisted of changing an abstract module (corresponding to computing some $\delta_G(f_0)$), changing a transformation set (corresponding to applying some δ_C), or changing the transformation sequencing rules (corresponding to applying some $\delta_{\mathcal{M}}$), and then regenerating modules affected by these changes as determined by the dependency net. This solution has the same disadvantage as that of Make: large grain dependencies.

9.4.3 The Zap transformation system

While the focus of Feather's Zap system [Fea79] was to transform moderate size programs, it is one of the earliest for which the notion of maintenance is prominently discussed. The fundamental problem for Zap was to build interesting control procedures to enable practical transformation of moderate size programs. The fundamental control concept is that of a *CONTEXT*, which provides *pattern-directed transformation*. The operation of *CONTEXT*s were described in Section 4.3.3, and can be summarized as nonprocedurally determining a sequence of transformations to achieve a state in which selected portions have a form specified by the current *CONTEXT*. The individual *CONTEXT*s used seem to be very specific to the program being transformed. Sequences of *CONTEXT*s form a metaprogram for generating an entire implementation. Such sequences are established by constructing a script file containing a series of *CONTEXT* descriptions. An individual *CONTEXT* can be considered analogous to a TCL method with postcondition requiring a particular functional specification fragment; the script file can be considered a high-level derivation history.

Maintenance in the Zap environment consists of modifying the original functional specification, and naively replaying the script file. It is not clear what happens when a *CONTEXT* fails to achieve its goal; we would expect the script is aborted as successive *CONTEXT*s appear to build on structures introduced by preceding ones, but one could simply abort the particular *CONTEXT* and go on to the next. Zap successfully reapplied such a script to a changed telegram word-counting program. Most of the *CONTEXT*s successfully completed due to their nonprocedural nature. This suggests that the nonprocedural nature of TCL method invocation will be useful during dynamic replay.

9.4.4 Bogart: Replay of a tree-structured derivation history

The BOGART system [MB87] stores a tree-structured derivation history representing the recursive refinement of components of the original functional specification. We described the history structure in Section 5.4.3. Control knowledge is represented as transforms which refine components into connected sets of subcomponents. The original functional specification can be changed directly, and BOGART will attempt to replay the derivation history to reimplement the specification. Deleting a component obviously deletes the refinement history of that component. Adding a new component effectively adds an empty refinement history for that new component. Changing a component constraint does not directly affect the history, but may affect whether a component is still refinable by its history. The replay process is top-down for the refinement history of each individual component. Each derivation history entry which refined a component is retried; if successful, the subhistory is replayed. Failure of a component to refine according to the history aborts replay of that subtree of the history; however, other subtrees can still be replayed. The BOGART system required one minute per replayed step, due to an expensive constraint propagation system; only small designs were tried.

If our estimate of $10,000$ transformations per average implementation is correct, the BOGART system cannot produce a timely result. Ignoring performance goals, our experimental DMS replayed unconditional transformations at the rate of tens of milliseconds each; admittedly, conditional transformations can take significantly longer but often even conditional transformations trivially commute because of the the non-overlap of locaters.

Our DMS has two other significant advantages. First, failure of a design history element simply causes DMS to remove that element, and following, but not semantically dependent elements are exposed for continued replay. Secondly, BOGART's refinement replay, while always producing a functionally correct result, may no longer achieve the unstatable performance goals that were achieved by the original derivation history on the original specification. Our DMS re-validates transformations when they appear reusable to ensure that performance goals continue to be met.

9.4.5 Replay by use of heuristic correspondence

Goldberg [Gol89] discusses a preliminary derivation replay system for the KIDS (Kestrel Interactive Development System) enhancement to the REFINE transformation system. We go into considerable detail because it is one of the few working history replay systems for software construction.

We have already examined, in Section 4.3.3, his tactics (procedural metaprogramming) language in which primitive tactics actually change the program and "abstract" tactics provide control; tactics are parameterized (see the example tactic in that section). Goldberg records a derivation history as a time-ordered trace of the calls (Section 5.4.2) to each tactic (abstract or primitive) along with the parameter values used at each call, especially the values of program-parts (equivalent to our locales).

The need for history replay is triggered by performing arbitrary edits on the original program. Changes to "specifications" are limited effectively to functional deltas. This is due to REFINE's lack of performance measures or performance goals.

Replay consists of attempting to re-execute each tactic, in order, from the derivation history with suitably revised parameter values. The revised values are generated roughly by looking up original parameter values, and substituting their equivalents, from a *hueristic correspondence* relation between program-parts of the original program, and program-parts of a revised program. Consequently the emphasis is on computing such a correspondence relation.

According to Goldberg [Gol89, page 7]:

> This is called the correspondence problem. Our method for establishing a correspondence is heuristic. It relies on three mechanisms:
>
> - Name Correspondence: The definition of the same identifier name in the program establishes a correspondence[1].
> - Structure Correspondence: Code appearing in the same (relative) position within the abstract syntax tree corresponds.
> - Parameter Correspondence: The execution of a tactic may cause a tactic variable to be bound to some code. Code bound to the same variable corresponds.

An initial correspondence relation based solely on name correspondence between the original and revised programs is constructed before starting replay. The correspondence relation is updated as the original tactics are replayed by identifying program-parts bound to the same tactic variable. If a tactic about to be replayed introduces a gensym'd variable name into the original program, then replay of that tactic will introduce yet another gensym'd variable name into the revised program; a correspondence between the two gensyms is added to the correspondence relation.

When replay of a tactic is attempted, its parameter variables must be bound. Certain values (constants like resource bounds, etc.) are copied intact from the original history. Otherwise, values for tactic parameters which are program-parts are obtained by lookup of the original value in the correspondence relation, and replacement by the corresponding value.

If the original program-part cannot be found in the correspondence relation, an abstract-syntax-tree tracing heuristic is used to locate the corresponding part. This part is found by climbing up the abstract syntax tree of the original program (both the original program and the modified program must apparently be generated in parallel, somewhat like the states in our abstract ladder) starting from the place identified by the original program-part, until some point n in the tree is found that is present in the correspondence relation. This defines a relative tree-path p from n back down to place defined by the original parameter. The place corresponding to the original parameter is then found by starting at corresponding point in the revised abstract syntax tree, and following the path p. This implements the notion of structure correspondence. If p does not apply in the revised state, then the replay step is broken and manual intervention is required. There is no discussion of what can be manually done to alleviate failure of that tactic, nor whether intervention

[1]We assume Goldberg means a variable declaration or assignment in the program, and not in the metaprogram.

is required immediately on tactic replay failure, or is delayed while attempting to replay other tactics. It does appear, since replay is done in the context of KIDS, that the software engineer could simply take the partially implemented specification and continue a conventional transformational implementation.

Since his high-level tactics mechanism is not implemented, he cannot play (let alone attempt replay) those tactics steps, and so he has not tried the parameter correspondence ideas. However, Goldberg has replayed derivations of up to 40 primitive-tactic steps in length for a high-performance topological sorting program. Even limited to replaying primitive tactics, he sees these results as "rather impressive."

Comparison to DMS The correspondence problem [MF89b] is the identification of objects in the new problem which corresponds to objects in the problem for which the design history was generated. If one can solve it, then replaying the design history is straightforward. Goldberg goes to considerable lengths to solve it, with only partial success as we will see below. This problem comes about when one has an old problem and design history, and is presented with an entirely new problem. With DMS, we finesse this problem by insisting that a formal delta to the old problem be given to define the new problem; inherent in the delta is the correspondence. It is interesting that systems using correspondences do not compute a delta simply to define the correspondence.

Our notion of locater as a constraint on bindings subsumes that of program-parts as tactics parameters. Our DMS unsurprisingly records a derivation history in almost an identical fashion to Goldberg, but additionally has a design history containing goal information and indexes back into the metaprogram, connected to the transformations in the derivation history. This allows the DMS to re-validate a replayable transformation, and to find substitute transformations or methods for those that cannot be replayed.

Goldberg's system only handles the effect of functional deltas, but never sees an actual delta. For DMS, we found the functional deltas to be extremely helpful in guiding the rearrangement of the derivation history and computation of revised locaters, by providing both a focus for the region of change, and also providing means for generating potential new locaters (by taking advantage of the structure of the deltas, implemented as subtree-tracing). As a consequence of not having the delta to guide the manufacture of replacement locaters, Goldberg must have the tree-walking scheme to find correspondences.

Goldberg's hueristic correspondence scheme appears to fails under some fairly simple circumstances. The name correspondence hueristic fails if the applied delta simply renames a variable. The structure correspondence hueristic must fail when subtrees of a design state are rearranged, either when changing the specification or by insertion of a transformation like a commutative law into the derivation history. Neither of these changes will confuse our DMS. We do see value in using the hueristic correspondence for generating potentially useful locaters; however, the DMS also validates their legitimacy (cf. discussion on *DELAY* in Section 7.2.1). If one has little knowledge of what the transforms actually do, i.e., they are opaque, Goldberg's hueristic correspondence is probably a reasonable approach, so we see our techniques as those of first choice, with Goldberg's being backup.

Failure of the correspondence heuristic can cause a tactic to be indetectably misapplied; our DMS would verify its correct role in the plan generating it. Furthermore, there appears to be no way to undo a misapplied transformation; the DMS can use *BANISH* to effect dependency-directed backtracking.

Goldberg provides no theory as to why his method should work at all other than implicitly leaning on the requirement that all primitive tactics should be correctness-preserving. Failure to apply a tactic leaves further useful execution of Goldberg's replay process in doubt, as the correspondence structure on which he depends is not updated. Assuming that Goldberg continues the next tactic in the face of failure of the current one, the maintained correspondence structure must diverge further from the true correspondence as failed tactics accumulate, either leading to more failures, or worse, misapplied but undetected tactics. In contrast, the DMS derivation history rearrangements have been shown to be legal. The DMS drops undesirable transforms and their dependencies. Retained transformations are validated against their place, and therefore purpose in the design plan. A failed DMS transformation has an index back into the metaprogram to provide the opportunity to manufacture alternatives explicitly allowed by the metaprogramming language.

A contrast: PADDLE [Wil83] represents a derivation history generator, and is re-executed in entirety in replay. Goldberg's tactics language is also just such a generator; however, having one replayed a tactic trace, it is difficult to relate the reused tactics back to their generator. We try to walk a middle ground with TCL and our replay scheme, in which reused transformations retain their place in the design history and thus continue to justify their purpose, as well as their index back into the generating metaprogram via agenda item *action* slots; this provides a way to locate TCL methods when plan repair is needed.

9.4.6 XANA: Replay for DIOGENES

XANA [MF89b] is the mechanism used to replay transformational derivations of search algorithms in DIOGENES, specifically for the purpose of re-implementing a changed functional specification. Unlike BOGART, DIOGENES applies arbitrary tree-transforms rather than just component refinements.

We have simplified the explanation of XANA somewhat, causing a considerable change of terminology in an attempt to be as consistent as possible with that of this thesis.

For DIOGENES, a specification is simply functional; no other performance specification is used. Each state s is a tree representing a functional specification. Each tree node has an identity, consisting of the label i for the transformation $H[i] = \langle t_j, \ell_i \rangle$, that generated it, and the relative path for that node from the point where the root of the tree transform t_j was applied. States can share tree nodes from earlier states.

An XANA derivation history is a digraph forming essentially a dependency net of tree nodes on applied transformations. The history consists of set H of transformations, each $H[i]$ consisting of an pair $i : \langle t_j, \ell_i \rangle$ implicitly labeled by the transformation index, where $\ell_i = \langle k, \mathit{path} \rangle$, with k being a label on some other transformation in the history, and *path* being a relative tree locater, a path (as described in Section 3.1.7) from a root to some subtree. The effective locater used when applying the transform t_j is found by concatenating all the relative paths found in the chain of transformations selected by following the backward pointing indices k. This computation can be avoided by naming the tree nodes as described below, and interpreting each transformation as meaning "apply t_j at to the tree node labeled $\langle k, \mathit{path} \rangle$". Our explanation requires each history H to have a distinguished transformation $1 : \langle n_1, \langle 1, \emptyset \rangle \rangle$; this introduces the functional specification by rewriting the empty specification to the desired specification, $n_1 \equiv \epsilon \Longrightarrow f_0$, obviously being a non-property-preserving transform. This requirement allows us to generalize both the global and relative paths used in [MF89b] into the single notion of relative path.

The transformations are recorded in the order applied by DIOGENES during the original implementation. Replay is in the recorded order. This order is one of the many legal equivalent-effect topological sorts of the history according to the dependencies; technically, any such sort for application order would be legitimate, but only the recorded order is used. The reason is because transforms are actually conditional, and can inspect tree nodes above or below the the point of transform application; the inspected tree nodes ("weak dependencies") appear not to be recorded in the history. The very fact of their inspection adds additional implicit sequencing constraints on the history that the recorded order honors, but other equivalent topological sorts may not. In contrast, our DMS will actually rearrange the order, substituting different transforms and/or locaters as required by the delta; the necessary sequencing is maintained by the requirement that reordered transformations provably commute.

For XANA replay, a new specification $\epsilon \Longrightarrow f_0'$ is created, defining $H'[1]$; this is an implicit delta. A correspondence between the tree nodes in the design states of the old and new histories is computed by inspecting the relative paths stored with each transformation. Transformations from H are sequentially copied to H' if their effective locater (computed using the new history) is still valid and the transform still applies. Should the transformation fail to be reusable, there is no recovery, because unlike DMS, there are no performance goals or alternatives recorded in the history; that transformation is simply skipped. Before replay, certain transformations in H can be manually marked as unreusable; the purpose of this is presumably to allow different performance goals to be achieved. There is no discussion of control knowledge or how it might be used in the replay process.

The relative path idea is similar to the technique actually used in our experimental system to trace subtrees during rewriting. The subtree tracing process produced the revised locaters without going through a British Museum algorithm.

Mostow tested the replay mechanism by perturbing a specification and running the replay process. For specification changes which were merely parameter substitutions, the derivation history was completely replayed. Specification changes substituting dissimilar constraints allow all steps but those involving the constraint to be replayed. Deleted constraints caused the replay process to terminate early because some subsequent steps depended syntactically on their presence. Marking such superfluous steps as useless allowed all the remaining steps to be replayed; Mostow suggests a goal structure would allow such steps to be automatically detected and deleted. For DMS, simple parameter substitution will always replay completely because such substitution will never prevent transformations from being swapped. Spurious syntactic dependencies are also handled by the *DEFER* mechanism. Finally, DMS does retain the performance goal history, although we have made no effort to remove transformations that achieve superfluous effects.

Like our DMS, XANA replays old transformations as long as they continue to apply (modulo lack of validation), so a manually applied transformation is retained over multiple changes unless invalidated. However, XANA should fail to replay parts of the history that apply to a part of the functional specification which is moved en masse by the functional delta, because the correspondence between the moved part of the specification and the original is lost when the delta is applied. The DMS use of a functional delta prevents loss of this correspondence.

A more serious objection is that XANA's replay scheme depends on the representation of the transformations; in particular, on the notion of tree path with respect to a root. XANA would not work if graph transformations were used, mostly because the notion of relative graph locater is not well understood. Our methods for design history integration are not sensitive to the representation of states or transformations.

9.4.7 ARIES: Specification Evolution

The ARIES project [JF90] is intended to provide assistance in the construction of functional specifications, prior to transformational implementation. It does not provide for the construction of performance specifications. In one sense, this work is complementary to ours, because it is focused on construction of the specification, rather than maintenance of the implementation. However, it is founded on the notion of using and modifying derivation histories, and so it is related to our work in Chapter 7.

We have discussed how the functional specification f_0 transformed by a transformation system is constructed by applying non-property-preserving transforms to the empty specification ϵ (Section 5.2). Work by Feather [Fea89a], leading up to the ARIES project, defined an "elaboration" effectively as a history H_n composed largely of non-property-preserving transforms n_i applied to an approximate functional specification $f_{-length(H_n)}$ to produce the exact functional specification $f_0 = (\Pi H_n)(f_{-length(H_n)})$. Such non-property-preserving transforms were termed "evolution transforms". The purpose of an elaboration is to allow the approximate specification $f_{-length(H_n)}$ to serve as a "white lie", or abstraction of the real specification f_0 [Bal85a], for expository purposes. An unstated assumption is that the essential meaning of the white-lie version of the specification is retained in the real specification.

For this idea to be useful, one must manufacture such elaborations. Feather suggested that new transformations can be added to the tail of an elaboration, or that transformations on the tail can be undone in reverse order of addition. Our DMS derivation revising mechanisms can be used to extend this to modifying elaborations. One might wish to apply a functional delta δ_f to $f_{-length(H_n)}$ if the "white lie" is inconvenient, producing $H'_n = \delta_f^{-1} + H_n$ as a revised elaboration producing the same f_0. Using (Section 7.4.2) $INTEGRATEMIDDLE_f(f_{-length(H_n)}, \delta_f, j, H_n)$ to insert a delta in the middle of H_n, or (Section 7.2.3) $BANISHATPOINT(H_n, j)$ to delete a now-unwanted delta would be useful for adjusting the exact specification.

Feather goes on to describe a scheme to merge "parallel" elaborations (derivation histories H_{n1} and H_{n2} divergent from f_{-k}) into a single elaboration H_{merged}. The purpose of this is to allow separate aspects of desired functionality to be independently developed from a common approximate functional specification, and then to combine these aspects to generate an exact specification containing both. This process operates roughly by merging transformations that do not interfere into the resulting history, much like our *PRESERVE* step (Section 7.4.1). A related idea for merging software enhancements by combining program "slices" [HPR87, HPR88] works by merging non-interfering portions of design states: f_0 is obtained by merging the slices of $H_{n1}(f_{-k})$ and $H_{n2}(f_{-k})$.

DMS classifies its deltas according to what input of the transformation system is affected. All ARIES evolution transforms are functional deltas δ_f by DMS standards. ARIES transforms are classified further, according to their effect on a GIST specification, as follows [JF90, p. 241]:

- Behavior changing
- Structure-adding (adds Type declarations, etc.)
- Replacement (Rename Concept, etc.)
- Terminology elaboration ("adding or changing an existing declaration")
- Abstracting (makes spec more abstract by discarding detail)
- Approximate unfolding (replaces use of construct with *nearly* equivalent construct)
- Unfolding (definition substitution or refinement) such as interposing a buffer between agents; also implementation decision)
- Reorganizing (restructuring without changing meaning)
- Data flow modifying (change flow without affecting meaning)

The Reorganizing, Data-flow modifying, Unfolding, and Approximate-unfolding transforms appear to serve as architectural implementation decisions rather than specification constructing operations (compare to our discussion on architecture in Chapter 10). The types of evolution transforms seem to derive from the particular structure of GIST as a specification language; we speculate that each functional specification formalism will induce a set of evolution transforms unique to that formalism, although the set may be similar in style to those listed here.

The ARIES evolution transforms affect a semantic net possibly containing virtual semantic links that represent the specification. This is an unusual specification representation, but fits within our model of states; virtual relations can be modeled as cached inferences. Some semantic relationships used in ARIES evolution transforms are:

- *component*: relations between modules and their components
- *entity-relationship*: specialization-of, parameter-of, type-of, instance-of
- *data flow relations*: between producers and consumers of values
- *control flow links*: control-substep and control-successor
- *fact flow links*: accesses-fact, modifies-fact between processes and declarations of facts used/modified
- *state description links*: associating statements and events with their pre- and post- conditions

Fact flows exist because the specification formalism explicitly allows statement of information flows; they are eventually turned into low-level dataflows. Operations on the semantic network are used to define the evolution transforms. The authors observe that since the set of semantic links is incomplete, an evolution transform library based on existing links is likely to be incomplete.

Merging elaborations requires detecting when two evolution transforms do not interfere. We have discussed in Section 7.2.2 how this is done for the ARIES semantic net representation.

Effort is spent to make evolution transforms retrievable by effect, described in terms of such network operations. Queries are made to an evolution transform library in terms of semantic network manipulation operations desired. This has the effect of defining methods with performance predicates for choosing evolution transforms. The authors expect that preconditions on evolution transforms will allow ARIES to plan more complex evolution transforms than directly requested by the specifier.

We assume that all transformations generated in a transformational implementation from the functional specification participate in plans that achieve a statable purpose. What is remarkable about the ARIES work is the idea that constructing specifications by aspects also has such purposes. We do not know how to encode such purposes with performance predicates because of the non-property-preserving nature of the transformations involved; this is clearly an area for further research.

9.5 Truth Maintenance Systems

A DMS bears many similarities to Truth Maintenance Systems (TMS) [Doy78, CRM79, Doy79, MD80, Doy83, McD82, McD83, Pet87]. In this section we outline the analogy, and then consider how effective one would be for design maintenance. We consider TMSs because they initially attracted our attention as having the right kind of revision properties for maintenance.

9.5.1 TMS Essentials

TMSs can be thought of primarily as rule based inference systems, with a repair mechanism used to fix inference chains whose facts/conclusions are found to be incorrect when tested against an external model. TMSs are generally used as a component of a larger, domain-specific problem-solving system (PSS), and are used to reason about a problem, and to identify potential points of interest in the problem description.

A TMS is initialized, by the PSS, with a set of *inference rules*, and a set of atomic *facts*. Facts each have a *truth status* of *true* or *false*, some acquired by direct *assertions* from the PSS. The inference rules are used by the TMS in two ways: actively, to produce new facts with truth status, and passively, as a set of relations between the facts.

In the active mode the rules are used to infer new facts and their corresponding truth status; the actual control regime (forward, backward or mixed) tends to vary according the ambitions of the TMS designer [Pet87]. Thus the *extant* facts in the system at any instant are the assertions and/or consequences of the set of inference rules that have run up to that instant. It is important to note that there are *potential* facts (and statuses) that could be inferred by rules, but have not, up to the instant in question. The TMS does not concern itself with potential facts. Unlike a pure production system, however, the fired rules and their consequences are retained along with their relationships to the extant facts. A TMS can provide explanations of its conclusions by tracing the fired rules.

A set of facts and fired rules are *consistent* if the fact statuses match the conclusions that the fired rules would draw if individually re-run on a database of facts whose truth status was that of their assertions. Such a state can be achieved by a batch-executed *consistent labeling procedure* [Rus85]. A set of rules connected by particular facts may not have a consistent labeling. Because the batch procedure can be slow, it rarely done; usually a consistent state is incrementally constructed by addition of single, consistent new facts [Rus85].

A newly asserted or inferred fact (status), however, may be *inconsistent* with the extant facts. This causes the TMS to attempt to resolve the inconsistency, by changing the statuses of the extant facts in such a way that that inconsistency evaporates, while treating the fired rules as constraints to be honored among the extant facts[2]. No new rules are fired. If a new consistent labeling can be found, then the result of the resolution process is a list of facts whose status has changed in order to make the extant fact base consistent with the fired rules. The PSS then processes this resulting list to either validate it against the world, or to choose some new sub-problem to consider.

Many possible sets of status changes may achieve consistency; since some facts are "more believable" than others in most problem domains, special *resolution* rules are sometimes specified (possibly by the PSS) to control which facts the TMS will consider for revisions first [Pet87].

[2]A trivial strategy for achieving this effect is to change the status of the newly asserted/inferred fact; since the new assertion/fact status is presumed to be more recently validated by external means than other facts from the PSS viewpoint, this trivial method is not used.

Justification-based TMSs (JTMS) [MS86] are willing to propose changes to the status of any extant fact, either asserted or derived by rule application. *Assumption-based* TMSs (ATMS) [dDR+78, dDSS77, deK84, deK86a, deK86b] treat PSS assertions as the key facts to consider for truth status alteration; other facts produced as consequences of inference rules will get revised as a consequence of consistency adjustment, but the presumption is that the assumptions are the ones in which the PSS has the most interest. ATMSs are consequently quite useful for diagnosis; assumptions about the correctness of the artifact are postulated by the PSS, the consequences drawn by the ATMS inferences, and those consequences validated by the PSS against the actual artifact. Contradictions of consequences found by the PSS are asserted, causing the ATMS to propose certain correctness assumptions need revision, and thus potential fault sources are exposed for the PSS to test.

The distinction between extant and potential facts leads to a peculiar effect: the consistency algorithm operates on a closed-world assumption with respect to the extant facts; no account of conflict with potential facts is attempted. This is obviously a concession to the cost of inference.

As a general rule, the process of adding new facts and revising consistency of the fact base are interleaved.

9.5.2 Relation to a DMS

In our context, there is an analogy to a Software Development System (SDS) [Fre87] with an DMS to a Problem Solving System with a TMS. The Software Development System consists of an organization, with goals to develop software, and the DMS corresponds to a software constructor/maintainer[3].

Like a TMS, the DMS provides the low level construction/maintenance/focusing mechanism for the SDS. The SDS defines the initial assumptions (software functionality and performance requirements), and the inference rules (domains, domain semantics (rules of transformational exchange), refinements, and control *resolution* heuristics). The DMS draws "conclusions" (implementations by applying transformations) from the "assumptions" (specs) given by the SDS. The controls and transformations fired are stored for later reuse. Like a JTMS, the DMS can explain parts of an implementation by tracing the fired transformations and control heuristics, and like an ATMS, can explain what parts of the specification control what part of the implementation.

[3]Unlike the PSS, the SDS is for the near future most likely to be an informal system.

In spite of the similarity, however, a TMS cannot serve as an DMS, for a number of reasons:

1. A TMS manipulates atomic facts. The DMS manipulates structural relations between design entities.
2. Atomic facts have truth statuses which are independent of other facts. Structural relations between entities are not "true" or "false" (although one could treat the *existence* of that relation as a boolean) but may exist in a number of different design states. This related to the frame problem [Pyl87]; each structural element in an DMS has positive support from the transforms which generated it, and negative support from transforms which delete it.
3. Denying a fact simply changes the status of the fact in a TMS; in an DMS, denying a structure is tantamount to say "that spec/implementation simply won't do."
4. A proposed fact's inconsistency is easily detected in a TMS (the fact is present among the extant facts with the opposite truth status); in an DMS, denial of a structure's existence does not appear to contradict anything.
5. TMSs seem to be natural in problems in which the set of facts is relatively fixed, so recording them all explicitly is reasonable. DMS operates in the domain of software construction, where the set of currently non-existing structural relationships is unbounded, and storing them simply isn't practical.
6. Repairing the inconsistency of a TMS fact-base requires running a repair procedure which depends only on the structure of the TMS; no new rules are fired, and no rules are "unfired". Finding a new implementation requires the DMS to run a repair procedure which depends on the control heuristics, requires previously applied transformations to be dropped (as they are no longer relevant), and new rules are likely to be fired to produce new structures.
7. With a TMS the validity of an inference is never denied[4], but the validity of transformations may be denied to the DMS and it must find a new implementation.
8. A TMS can have cyclic dependencies. An DMS cannot; no valid implementation can simply assume portions of itself are correct.

9.6 Nonlinear Plan Repair and Reuse

Nonlinear planners are often used by robots to produce possible plans of actions given some desired goal state [CM85]. The notion of nonlinear plan is often used

[4]While Proteus [Pet87] *rules* do have a truth status, that status is not used; i.e., it is never denied.

as a representation for sets of possibly unordered actions that achieve some overall result [Sac77, Geo87]. In this section, we compare our work to the SIPE [Wil88], IPEM [AI87, AIS88], and PRIAR [Kam89] nonlinear planners, each of which does some form of plan repair and reuse.

We can compare our work to that of nonlinear planners in general by drawing parallels between:

- design states and planner world states
- transforms and operators
- functional specifications and initial planner world states
- performance goals and goal world states
- histories and nonlinear plans (see Section 5.4.4)

Planner world states are often conceptually represented by set of predicates describing primitive relations between world objects, along with derived relations. Such predicate sets correspond to our notion of a design states containing cached consequences.

Operators change the set of predicates which form planner states, while transforms map design states to design states. A fundamental difference in representation exists in that nonlinear planners almost never realize complete representations of state as we have for transformation systems. Instead partial states are dynamically computed relative to some set of nodes in the nonlinear plan; determining if some relation is true in such a partial state is called the *modal truth criterion* [Cha87]. While computing such truth values is expensive, the absence of nonessential sequencing makes it well worth the trouble. We used complete states for DMS to avoid the problem of computing performance predicates over partial program schemes, because we did not know how to characterize how program schemes could be partitioned; this problem is related to that of defining appropriate notions of locale. Such notions of partial state are needed to make constructing ladders in the context of a design history practical (Section 8.3.3).

A planner is given an initial world state and must find a sequence of operators to apply to change to a goal world state; a transformation system is given a functional specification f_0 and must apply property-preserving transforms to locate a state in which the remaining performance predicate G_{rest} is true. The parallel between performance goals and goal world states is uneven, because performance goals are often stated in terms of complex derived properties of states, whereas planning goals are usually stated in the exact same terminology as used for initial world states.

Similarities between transformational implementation and planning suggest that similar problems in the planning domain have solutions of interest for transformational maintenance. Several problems exist for such robot planners:

- Plans must be constructed
- Planning is expensive
- The robot's knowledge of the world may be faulty
- An action may fail to execute perfectly
- The robot's goals may suddenly change

Plan construction is essentially a search problem. We discussed control mechanisms for nonlinear planning in Section 4.3.4.

Because of the expense of planning, it is interesting to find and reuse plans in new environments. We can turn this into a maintenance problem by computing what amounts to a functional delta δ_f between the present world state and the world state of the recycled plan; Kambhampati's PRIAR system [Kam89] effectively does this. Errors discovered in knowledge about the current state during execution of a plan can similarly be treated as a sudden requirement to insert a functional delta into a derivation history; SIPE [Wil88] does this with "Mother-Nature" actions.

While executing a plan, an applied operator may fail to act properly; the state predicted by its action may not be achieved by its action. In this case, an unexpected world state is suddenly encountered. This case directly matches the transformational maintenance situation in which the functional specification (the expected state after operator application) is changed by a δ_f into the unexpected world state, and the plan must be repaired accordingly. Both the SIPE [Wil88] and the IPEM [AI87, AIS88] systems handles such plan repair. In a transformation system, transformations do not fail, but methods can; however, failed methods cause backtracking rather than Δ_f integration because the functional specification does not change.

During execution of an existing nonlinear plan, the robot may decide that the original goals which motivated the plan are no longer appropriate. Such a change of goals requires that the existing plan be modified in some fashion to take into account the new goals, and drop plan components related to now obsolete goals. The IPEM planner [AI87, AIS88] does this. Such a change corresponds to a performance goal change δ_G in our framework.

Our notion of shared agenda item is a useful addition to the notion of phantom goal used by typical nonlinear planners. Phantom goals are recorded when a plan step s is to achieve a desired effect g, and g is found serendipitously to be true in the partial world present when g is to be accomplished. This is certainly valuable when s must definitely follow other steps $S = \{s_1, s_2, \dots s_n\}$ that made g true, but places an inappropriate asymmetry in the plan when s can be executed in parallel with S. The asymmetry shows when an explanation of the plan is requested; the explanation for a phantom goal node s is "Nodes S necessarily before s have already accomplished this.", whereas the explanation for a shared agenda item would be "Doing S accomplishes this, and also serves $parents(S) - s$." The asymmetry of a phantom goal node also shows when some step $s_j \in S$ is suddenly no longer needed to accomplish the purpose of S; one must expend effort to determine of s_j serves a phantom, and if so, replace the phantom by s_j. No plan repair system with which we are familiar does this; rather, they delete s_j and attempt to re-achieve g at a later time, wasting the knowledge that s_j already has the desired effect, and the already generated subplan under s_j that actually achieves it. By using a shared agenda item, we achieve this effect easily.

Nonlinear planners need not "reorder" most independent operators at all; this is the entire point of the nonlinear plan representation. For DMS, this corresponds to *SWAP* with unchanged locaters. However, for DMS, we have seen the value of *SWAP* in which the locaters do change, and the corresponding value of *DEFER*. Nonlinear plan repair mechanisms have nothing remotely similar.

9.6.1 SIPE and replanning

SIPE [Wil88] is one of the few domain-independent, nonlinear hierarchical planner that allows for replanning in the face of unexpected events.

Control for SIPE was described in Section 4.3.4. SIPE plan critics diagnose and fix plan bugs produced during the planning process. The design selections made by such critics can be captured in nonlinear histories but the design choice causing them is not. The absence of such explicit records we think makes design repair harder because certain alternatives are left implicit.

Planning, plan execution, and execution monitoring for surprises in SIPE are interleaved to allow recovery from unexpected events; replanning only occurs when the environment changes, corresponding to DMS recovering after application of a functional delta δ_f part way through the implementation process.

The following replanning actions can occur [Wil88, page 153]:

- Insert: inserts a new subplan after an existing subplan. This is not used directly by plan repair, but rather acts as a "subroutine" for most of the following repair actions.
- Insert-Conditional: inserts a test for value of unknown state variable
- Retry: converts a phantom goal node into an incomplete goal node
- Redo: Adds a new goal to be achieved to the plan
- Insert-parallel: Adds a parallel set of goals to the plan
- Reinstantiate: Change binding of a variable to an object to reachieve a predicate
- Pop-redo: Removing a subplan and replace by an incomplete goal node
- Pop-remove: Removing a subplan whose effect is already achieved.

TCL plan repair and the delta integration procedures collectively provide very similar actions:

- Insert: Executing an agenda item, and in particular, inserting a transformation into the derivation history
- Insert-Conditional: unnecessary in a transformation system; there is never any doubt about the accuracy of state information.
- Retry: accomplished by pruning back to an *ACHIEVE* node.
- Redo: Adjustment of *ACHIEVE* conditions in the face of Δ_G (Section 8.6)
- Insert-parallel: Like Redo. Implicit in a single *ACHIEVE*, so it isn't really necessary.
- Reinstantiate: Changing a locater to achieve the same effect
- Pop-redo: Pruning a subplan back to an alternative (Section 8.2)
- Pop-remove: Pruning a subplan back to an *ACHIEVE*; when tried, the agenda execution mechanism will discover that the *ACHIEVE* condition is true.

The SIPE notion of deleting a "wedge", the subplan below an agenda item, is equivalent to TCL subplan removal. This kind of mechanism must be present in any kind of hierarchical planner precisely because of the notion of hierarchical plan; deletion of the parent of such a plan must naturally delete all of its components. However, we see little value in limiting the mechanism to mere wedge removal; invariably after removing a wedge, one must prune back to a choice point. There appears to be no need for the Design Maintenance System notion of agenda-item marking, because SIPE does not handle changes to methods or transform libraries.

For SIPE, replanning occurs when the current world state is suddenly changed; SIPE is told precisely which facts changed by inserting a "Mother Nature" operator (in effect, the explicit δ_f) into the plan network that expresses the status of the revised facts. By computing the necessarily following nodes of the "Mother Nature" node, those parts of the plan that use the changed facts can be found and re-tested. The changed facts can cause any of the following problems, which are cured by the corresponding actions [Wil88, page 153]:

PROBLEM	REPLANNER RESPONSE
purpose not achieved	Redo
previous phantom untrue	Reinstantiate, then Retry
unknown variable	Insert-Conditional
future phantom untrue	Retry
precondition untrue	Reinstantiate, then Pop-redo
parallel postcondition untrue	Insert-parallel

Most of the problems detected by SIPE are accomplished by Design Maintenance System via replacement transformation revalidation (Section 8.3.3).

Design Maintenance System obviously handles many more kinds of changes than SIPE.

9.6.2 IPEM: Plan repair

The Integrated Planning and Execution Monitoring (IPEM) nonlinear planning system [AI87, AIS88] takes a kind of production-system approach to both planning and plan repair in the face of unexpected events. In particular, planning and plan repair are indistinguishable, simplifying the overall architecture of the system. We followed this philosophy for the TCL execution engine. An aspect of IPEM which we do not consider is its ability to interleave both planning and execution, as such ability is not really meaningful for transformational implementation.

IPEM uses a notion of *range* to tie effects produced by one action to preconditions of following actions; this is a special case of the validations used by PRIAR Section 5.4.6. Ranges have the effect of providing sequencing constraints between nodes, as the action producing a range must necessarily be executed before an action that consumes it.

The IPEM system elaborates plans by execution of "metaplanning" operators. Each metaplanning operator has a precondition under which it fires and an procedure which modifies the existing partial plan. Plan *flaws* are defects in the plan; incomplete agenda items, improper ordering among actions, etc. For each flaw, there are a set of plan *fixes* methods for resolving the flaws. A set of flaw/fix pairs constitutes the metaplanning operators. Such a framework is similar in spirit to the DMS notion of delta-specific integration procedures.

The most interesting aspect of the fix/flaw framework is that plan repair is completely incremental; flaws can be fixed in any order (dependencies will of course necessitate backtracking). Fixing a flaw can, of course, introduce yet another flaw. Since IPEM is implemented in PROLOG, backtracking occurs automatically if an applied fix eventually leads to a dead end, and alternative fixes are then tried.

We list each IPEM-defined flaw, and the corresponding fixes:

- Unsupported Action Precondition:
  - Attach Range to action known to be earlier
  - Attach Range to parallel action, ensuring it is earlier
  - Attach Range to newly-created action
- Unresolved Parallel Conflict: Order conflicting actions
- Execution Flaws:
  - Incomplete Action: Expand Action
  - Unexecuted Action: Execute Action
  - Timed Out Action: Remove Action and Dependent Ranges
  - Unextended Range: Attach Range to Plan Head
- Replanning Flaws:
  - Redundant Action: Remove Action
  - Unsupported Range (false fact): Remove Range

An unsupported action precondition flaw is roughly equivalent to a TCL *ACHEIVE* agenda item. The corresponding multiple fixes are essentially different ways of satisfying the goal. The fixes that attach ranges to existing actions constitutes making a phantom of the goal; creating a new action constitutes decomposing into subgoals. IPEM apparantly has no way of constructing a shared action.

Expanding an incomplete action roughly corresponds to the TCL agenda-oriented execution model, with TCL placing priority on agenda items which are "early" in the plan to maximize downstream damage early while the plan is still small.

The execution flaw "Timed Out Action" handles a problem which occurs in robots: actions may not work or complete. Under such a circumstance, replanning to achieve the originally desired effect is necessary. This is reminiscent of a "temporary" technology change, i.e., designation that a particular transformation in a DMS derivation history is invalid without changing the transform library.

The replanning flaw of Redundant Action corresponds to handling $\Delta_G.G_\ominus$, i.e., removing additional performance goals, thereby removing the need for actions to achieve them. Adding new goals causes unsupported action preconditions. The unsupported range flaw detects changes in the current world state and its fix removes actions which depend on newly deleted facts; this is similar to handling Δ_f.

9.6.3 PRIAR: Nonlinear Plan Reuse

The PRIAR nonlinear plan reuse system [Kam89] modified a supplied plan for use in a new problem situation.

Reusable nonlinear plans are augmented by *validations* and *annotations*, providing fine detail about which actions generate and which actions consume which generated facts, as described in Section 5.4.6. These fine-grain dependencies are the key to efficient modification of the plan. In particular, the validations provide for fact dependencies in a way which is not dependent on the applied operator sequence, as is XANA.

A mapping α specifies a partial map from the objects in the supplied plan to the objects in the new problem situation. From the mapping, deltas similar to those of our DMS could be generated and processed. Changes to sets of facts in the initial world correspond to δ_f. Changes to sets of facts in the goal world correspond to δ_G. Since PRIAR handles both sets of changes at once, it acts as though it processes a composite delta $\langle \delta_f, \delta_G \rangle$. PRIAR uses methods similar to those for DMS for adjusting the design plan.

Applying the map α to the entire recycled plan produces a plan for the new problem, which must usually be repaired. Facts that are no longer true, new facts in the new starting situation, extra goals and unnecessary goals are marked in the recycled plan. Next, each validation dependent on a marked fact is checked. For each failing validation, a repair task is proposed.

Each new goal causes a new $ACHIEVE(goal)$ node to be added to the plan, to be later solved by the planner under the implicit assumption that separate goals are usually independently achievable (in contrast, DMS walks down the design history tree with a δ_G changing $ACHIEVE$ nodes as it goes; this difference seems caused by the nonlinear planner representation of facts and therefore goals as separate entities, in contrast to the DMS treatment of states and therefore performance predicates as monoliths.) Deleted facts that formerly satisfied phantom goals cause the phantoms to be "de-phantomized", also to be solved later by the planner. Deleted facts that formerly satisfied action "filter" preconditions (i.e., analogous to enabling a *REQUIRE* in a method) cause the subplan containing that action to be pruned in a fashion virtually identical to that of TCL pruning. However, the action at the root of the pruned plan is replaced by a goal to *ACHIEVE* all the E-conditions of the subplan, recording all the necessary effects of the now missing subplan; this prevents an immediate cascade of failed validations for portions of the plan depending on the pruned section. New facts serendipitously satisfying preconditions cause subplans to be pruned and replaced by phantoms. A special case leaves changes a validation without changing the plan structure: if originally $E \vdash C$, and E is replaced by E' by α, then $E' \vdash C$ is checked, and if provable, only the validation is adjusted; this is similar to a performance bound delta subsuming an existing performance bound when DMS is installing a performance delta.

After the plan has been repaired, the partial plan is shipped to the planner for completion; unlike DMS, the planner in repair mode (as opposed to fresh-problem solving mode) attempts to instantiate pruned subplans first before newly added goals in an attempt to satisfy validations already present in the partial plan. This is probably one of the best ideas in PRIAR, as it tends to prevent the spread of damage to the plan. Because a pruned subplan is converted into a goal to achieve the pruned subplan E-conditions as subgoals, it appears that a repaired plan may be unexplainable in terms of the problem solving primitives available to the planner; problem decomposition is not likely to produce such idiosyncratic sets of of subgoals. It is not clear whether the planner keeps the annotations up to date, or why the planner doesn't actually use them during planning; if the annotations existed during planning, the planner repair mode would simply be a clever backtracking method.

The PRIAR work shows that plan reuse cuts the search space exponentially, and shows an empirical validation of plan reuse saving 95%+ over fresh planning in selected blocks world examples. It is suggested that PRIAR ideas could be used in design replay; we agree that they should be investigated.

The difficult problem of handling an unpreservable transformation supporting a larger plan (Section 8.3.3) is not handled directly by PRIAR; it is somehow hidden in the planning mechanism backtracking logic. It becomes an explicit problem in our DMS framework because of our retention of a delta during the ladder building process.

Our shared agenda item deletion (Section 8.2) process is reminiscent of Kambhampati's task node deletion when the task no longer has any "external" effects.

9.7 Summary

We have compared our notion of a DMS with a number of related works. The DMS concepts and methods subsume most of the work not related to planning. Nonlinear planners have the advantage of nonlinear primitive operations over our DMS, making the notion of *SWAP* for trivial exchanges trivial to compute. We summarize this relation in Figure 9.1.

We see that DMS is the only system that supports:

- Performance specifications
- Explicit Deltas of a variety of types

Our analysis in this chapter demonstrates that our design history representation and delta integration procedures are more broadly based in terms of the range of deltas handled, and more robust than those of the other systems examined, by virtue of being theoretically motivated.

In brief, for each of the following schemes, DMS has the listed advantage:

- Dynamic Metaprogram Replay: No need to rediscover actual history elements
- Correspondence Discovery (Goldberg): Functional delta unerringly guides
- Node dependencies (XANA): Not confused by movement of specification parts
- Derivation Histories only: Revalidates reused transformations, finds new methods to replace failed transformations
- Syntactic Dependencies: DMS can reorder if not semantically dependent

System Name	Explicit Perf. Spec.	Maintenance of Specification	Explicit Derivation History	Explicit Delta	Replay Ability	Explicit Justification
Ad Hoc	n	n	n	n	n	n
Make	n	n	y	n	y	n
DESIRE	n	n	y	n	n	n
TMM	n	Y	n	n	n	n
Paddle	y	Y	n	n	Y	n
Glitter	n	Y	n	n	Y	y
LP	n	Y	Y	n	Y	y
PDS	n	Y	y	n	Y	n
ZAP	n	Y	Y	n	Y	y
SINAPSE	n	Y	Y	n	y	n
Goldberg	y	Y	Y	n	Y	n
Bogart	n	Y	Y	n	Y	n
XANA	n	Y	Y	n	Y	n
ARIES	n	Y	Y	Y	Y	n
TMS	–	–	Y	Y	y	Y
IPEM	n	y	Y	n	n	Y
SIPE	n	y	Y	Y	n	Y
PRIAR	n	y	Y	Y	Y	Y
DMS	Y	Y	Y	Y	Y	Y

Key:

n	definitely not
–	irrelevant.
y	possible to argue that it supports, usually not by design.
Y	obvious support

Figure 9.1: Comparison of systems supporting maintenance

Chapter 10
Conclusions and Future work

Chapter summary. We summarize the thesis results. We consider some insights derived from the work. Future research directions are discussed. The impact of this work is considered.

10.1 The main result

Our fundamental interest is in making the notion of Incremental Evolution of software possible: integrating a stream of deltas generated by comparing implementations to expectations, to obtain successively better implementations. Having determined that design information is necessary in order to accomplish practical modifications to existing implementations, we chose a formal model of software implementation, transformational implementation, in order to force such design information to be formally representable and therefore capturable. We determined that much of the necessary design information could be captured by recording design history of the decisions made by the transformation system, and that maintaining this design was the key to revising implementations. With this background, we limited our purpose to demonstrating that:

> **We can efficiently maintain software generated transformationally by integrating formal deltas into design histories.**

Our approach was to produce theory and procedures necessary for a Design Maintenance System, which would then realize an efficient kind of support for Incremental Evolution.

To achieve the goal of constructing such theories and mechanisms, we have:

- Provided an architecture for a Design Maintenance System based on a transformational implementation model;
- Defined a transformation control language, TCL, as a means for generating implementations and producing design histories as byproducts;
- Formalized a complete set of maintenance deltas based on our model of a transformation system;
- Provided procedures for integrating various types of deltas into an existing design history by taking advantage of commutativity in the design space;
- Provided an empirical validation of the existence of significant commutativity in a model of a design space;
- Demonstrated the utility of the derivation history revision procedures by means of examples;
- Validated those procedures for revising the derivation history component of a design history by an experimental implementation

10.2 Analysis and Insights

In this section, we consider some global aspects of a Design Maintenance System. We discuss controlling change management costs, types of of modularity and their utility, and a new perspective on what constitutes an architecture.

10.2.1 Completeness of a Design Maintenance System

We have tried to ensure that our model of a Design Maintenance System is complete by modeling the entire software construction process formally, and providing delta integration procedures for changes to each type of input. If a transformation system can develop software fully automatically from its description, then our approach is complete. We see two possible failings. The first is that our model of a transformation system is wrong or missing some input. This kind of problem should be relatively easy to repair in our framework; simply postulate a different/new input and produce integration procedures for it. The second failing, which is more likely, is that the transformation system does not have enough design knowledge of its own to carry off an implementation by itself, and so certain transformations are chosen by a software engineer for which the motivations are not recorded. We feel this is really a problem in knowledge acquisition and not a problem with our framework.

10.2.2 Change Integration Costs

The purpose of Incremental Evolution is to make implementation and maintenance more effective. We have provided procedures for integrating deltas into design histories. Can such procedures be accomplished efficiently? We have shown that certain operations on a derivation history are $O(n^2)$, and that $n = 40000$ is not unreasonable. It is clear that if the cost of such procedures exceeds the cost of re-implementing a specification from scratch, it would be better to re-implement. In the worst case, we can bound the delta integration costs by running an implementation process in paralell, but we expect to much better on average. Our integration procedures depend on commutativity in the design space. We determined empirically that for small spaces with properties like that of design spaces, there was a considerable amount of commutativity, even for $n = 28$; analytical analyses from [Bax88] suggested the commutativity grew exponentially with the size of the space, so there is considerable hope. This hope is complemented by the fact that real maintainers perform the maintenance task successfully every day, without changing much of the maintained artifact.

10.2.3 Artificial Modularity vs. Essential Modularity

One of the fundamental methods for conquering complexity is problem partitioning. Such partitioning has become an important part of software engineering in the form of the slogan "information hiding" [Par72]. An organization divides a software system into "modules", defines fixed interfaces for the modules, and can then parcel out work to smaller organizations. We call this scheme *artificial modularity*, as the structure of the modules is imposed by the organization on the designer. Many software methodologies attempt to make such boundaries fit natural boundaries of the problem itself (OOP, JSD) in an attempt to minimize future maintenance troubles.

We contrast artificial modularity with the idea of *essential modularity*: the real separability of concerns in a software system. Essential modularity is partitioning that respects only the true semantic dependencies derived from the nature of the problem and its solution, rather than simply being imposed. Artificial modularity is the often imperfect, very conservative abstraction of essential module boundaries.

The purported value of artificial modularity in freezing module boundaries is to limit communication between using and implementing teams to agreement on the module interface. The difficulty with this idea is that such artificial partitions often do not match the problem. When difficulties unresolvable by a module team arise, no solution is possible precisely because the module interface is frozen; this is a failure of artificial modularity to separate the concerns. Changing the module interface is an admission that artificial modularity has failed to achieve its goal of minimizing communication. The real difficulty is usually that a semantic dependency crosses the module boundary. It is often the case that the problem is easily resolved when both teams are willing to make changes; this tells us that the essential module crosses the artificial module boundary.

Consider a module A that manipulates a data structure logically via access procedures in module D. Optimization requires spreading information across module boundaries; for A to be "high" performance, we can expect that some representational properties of the data structure have been encoded into A. Changes to the data structure itself are likely to affect module D, and therefore to affect module A. Artificial modularity would insist that A and D are implemented separately, preventing the optimization we desired. Essential modularity would keep track of how A used the procedures of D, allowing aspects, and therefore changes, to D to be traced to their effect on A.

Others have noticed similar problems with artificial modularity. In an analysis of typical modifications made to real software systems, [Bor89] discusses problems caused by non-essential change propagated across artificial module boundaries:

> ... Not all effects of modularity are beneficial. (Our work) suggests that most of the recompilations performed after a change to an interface are redundant and that this redundancy is a direct consequence of how we modularize software systems. ...
>
> ... we would expect between 6 and 9 out of every 10 compilations to be unnecessary (as a consequence of this fact) ...
>
> ... (Evidence) validates the approach ... to use an underlying flat (i.e. non-modular) representation of program objects, and to the extent that recompilation costs reflect general program complexity, leads us to question some basic assumptions about modularization.

The ability to detect real impact, rather than artificial impact, can help alleviate this.

Essential modularity will allow teams to divide problems along natural boundaries. Interactions between teams are necessarily required when problem affecting other teams work arises. In the conventional SE environment, where communication is manual, slow, unreliable, and the problem is not well understood, essential modularity is perhaps a liability. In an environment where consequences of effects can be traced quickly, we argue that essential modularity is not a disadvantage; besides, it is not possible to get rid of such interactions anyway (as the existence of changes to module interfaces suggest).

To make using essential modularity possible, we must have tools that can trace the effects of decisions across module boundaries. The design history supplies key information needed to trace the effects of decisions. We see a Design Maintenance System as type of tool necessary for managing such essential modularity.

We do recognize the utility of artificial modularity for the purposes of controlling reasoning costs, both computational and conceptual. We simply want to point out that artificial modularity for controlling communication, as required by software engineering methods of past and present, may be of hindrance in methodologies of the future in which communication is not such a large problem.

10.2.4 On what constitutes an "Architecture"

Transformational maintenance gives us a new perspective on the meaning of an architecture. [Gov71, p. 113] defines *architecture* as

> a method or style of building characterized by certain peculiar style of structure or ornamentation.

In engineering, the term usually refers to fundamental organizational properties of an artifact. For software systems, an architecture is usually some high level choice of problem solution coupled with a partition of the solution into major components which cooperate to achieve the solution. A widely available software engineering text [Fai85, p. 40] doesn't really define architecture; it simply states "Architectural design involves identifying the software components, decoupling and decomposing them into processing modules and conceptual data structures, and specifying the interconnections among components." It seems clear that the architecture of such artifacts is the consequence of some decision-making process. What is it that makes the notion of architecture useful?

Our answer deemphasizes the actual structures, and instead emphasizes the *cost* of acquiring, understanding, and/or undoing the decisions that lead to the particular artifact at hand. The architecture comes about by careful consideration of the problem solution, and is usually tampered with at the peril of the tamperers. *From the transformational perspective, we suggest that architecture is precisely those structures induced by the design selections which support a large portion of a design history*[1]. Such architecture is recognizable because it repeatedly appears in similar artifacts,

[1]Remember that certain costs may be caused by factors external to the implementation: software engineer understanding and user education. User re-education costs explain why apparently trivial design decisions tend to get preserved.

and attempts to change it are usually expensive, typically in learning how to live with the new structure.

The repeated appearance of a structure is a consequence of its reuse, either because it is a fundamental technique for solving a problem in the domain, or because discovery of the technique was difficult, the problem is common, and consequently the solution was deemed valuable enough to save and reuse. In the case of a design history, some portion of the history will survive repeated installation of changes. We define the long-term surviving portion to be the architecture of the artifact. This makes architecture a consequence rather than a cause.

10.2.5 On Commutativity in the Design Space

If one had to choose a single lesson to be gained from this thesis, it would be
Commutativity in the design space aids design activities.

The commutative nature of the design space provides us with considerable opportunity for design repair. It was this insight that lead to this entire approach to transformational maintenance by rearranging a derivation history. Dependency nets are based on a weaker form of this idea; commutativity induced by the large diameter of the design state versus the relatively small scope of effect of individual transformations. Dependency-directed backtracking also necessarily involves commutativity.

A related lesson appears in Lexical Searching [Bax88]: the notion that that a problem space can be *nearly decomposable*; while we cannot expect to have problems neatly decompose into entirely separate problems, we can hope that subproblems are not so hopelessly entangled in their brethren that subproblem solutions are useless. Thus we see commutativity as actually being an aid to the problem of design. In particular, commutativity is a major source of essential modularity.

We find it rather remarkable that there is often independence between design decisions, and are pleased that it exists, allowing us to revise our designs without necessarily throwing all of our other decisions away. Otherwise design might truly be an impossible task.

10.3 Impact

We consider the utility of this work in a broader context than simple transformational maintenance.

10.3.1 Incremental Engineering

We set out initially to realize the dream of truly incremental engineering: the ability to dynamically change our mind about desirable properties of an artifact and acquire one quickly. Our Design Maintenance System seems to provide a start in this direction. What effect would this have on conventional software engineering practices?

We see the following effects:

- Continuous feedback model of design and implementation
- True melding of rapid prototyping with design and implementation
- Different costing schemes will be required
- Possibility of better cost predictions
- Better documentation for would-be maintainers
- Focus of debugging on requirements rather than implementation
- Lessening of costs of errors or changes in requirements

From the point of view of the customer, software lifecycles based on the waterfall model usually require an intense interaction with the developers during the requirements analysis process, a long quiet period during implementation, and then a major surprise when the implementation finally appears, and consequences of early requirements decisions are finally seen. An Incremental Evolution model suggests that construction consists of continuous comparision of a partly completed artifact with customer desires. The customer is involved with the process continuously. This is similar to an extreme version of Boehm's Spiral model; rapid prototyping and implementation are no longer distinguishable.

The waterfall lifecycle model in its purist form is a one way street. Management likes it because it appears to provide definite milestones in a software construction process, and such milestones aid planning. The difficulty is that the pure model does not reflect reality; there is continuous feedback between all the various stages, and none is every really quite complete until the project is declared done. The very milestones on which management is basing schedules simply don't exist; it is not surprising that many projects arrive at a "Test" stage and stay there long after the original estimated completion date.

Prediction of costs must be made on a basis other than major milestones. With an Incremental Evolution model, construction consists of integrating large numbers of small deltas. It is possible that these deltas have useful statistical properties; one property might be the average number of deltas for a typical problem domain implementation. Such statistical properties would provide management with alternative prediction schemes. The possibility of using dependency nets to gauge the difficulty of proposed changes would also help cost estimation.

Understanding what a system does and how the effect is achieved is a prerequisite to changing it. A Design Maintenance System provides, via the design history, useful implementation information for the would-be maintainer. Unlike conventional maintenance, this information is completely current and accurate. Tools to navigate the design history can allow more focused browsing than conventional designs in which no justification is recorded. These effects should lower the cost of understanding artifacts, decreasing costs of generating change proposals, as well actually enhancing change management.

Use of a Design Maintenance System base on transformation systems would change the emphasis of debugging from implementation repair to requirements repair. Use of formal specifications and a base of tested methods and transformations ensures that the implementation generated truly meets the specification; the problem then becomes one of acquiring the right specification rather than finding errors in the implementation process.

Since a Design Maintenance System is intended to automate much of the process of installing changes according to specification changes, the cost of installing such changes should be significantly less than corresponding costs in conventional software engineering processes.

Overall, Incremental Evolution implemented via a Design Maintenance System should have positive beneficial effects on software lifecycle activities and costs.

10.3.2 What do we do about "Dusty decks"?

If a transformation system is required to do maintenance, what can one do about existing programs that are not derived transformationally?

The rather obvious answer is to construct a design history for the existing program along with its specification, and then apply the methods outlined in this thesis. This can be a painful exercise when we realize just how much information is missing.

We need a formal specification and a complete design history. Simply acquiring a formal specification is likely to be hard for many reasons:

- We may not have a clear idea of the problem domain in which the program operates. This requires that we do a domain analysis [Nei80, Nei84a, Ara88] before we even start, just to identify the proper vocabulary. Even if we have a library of existing domain analyses, we will need to validate the choice of any particular domain as the problem representation basis.
- There is no accurate, written, let alone formal, specification of any of the performance aspects of the problem to be solved. Absence of even informal written specifications is a long-standing tradition with most code, let alone informal specifications which accurately reflect the intent.
- Assuming it is even expressible, a formal specification is likely to have a large number of ugly warts due to useless, buggy, arbitrary, idiosyncratic, or environment-specific code that is present in the existing code. Such warts will be difficult to understand or validate. We strongly believe that commitment of intentional abstraction error [ABFP86] will be necessary to minimize the difficulty caused by such warts; this is sort of the converse of revising the specification due to the implementation [Swa82]; instead, we revise the implementation as dictated to simplify construction of the specification.

Constructing a legitimate design history has its own pitfalls:

- The set of transformation rules and methods are similarly likely to be unclear, necessitating a domain engineering step [Ara88] or at least domain engineering validation.
- A valid design history must be generated that converts the proposed specification to the implementation. If human agents propose the specification, it is highly likely to be wrong, and no correct implementation of a wrong specification can lead to the existing program. Specification repair will be necessary, but knowing when to repair the specification and precisely how to do so are likely to be difficult.

The idea of reconstructing an idealized explanation of programs is propounded by [PC86], who suggest faking a rational design process during program construction. Such a characterization is at best informal, and one needs considerably more detail, but an informal design characterization is probably a necessary intermediate step. [ROL90] gives some methods for identifying informally various design decisions present in existing code.

Systems like GIBIS [CB89, WML$^+$89] use hypertext to annotate documents such as source code with decision points, possible choices, and arguments pro- and con- for those choices, and might be useful tools during the design recovery process.

Design recovery systems ([Big89a, Big89b, BCC89, CC90, CS89, HN90, Nin89, Our89, PGLS88, RD88, Wil87, RW90] are first steps towards automating the recovery process in that they attempt to recover some of the design of existing code by matching code fragments to various program plans that accomplish known computational goals; the result is parameterized plans for the code, but not true performance specifications. More ambitious systems actually attempt to recover a formal specification from code instances into semi-formal (JSD) notations [SJ88] and formal (denotational semantics) specification styles [WCM89].

A characterization of transformational maintenance from a partial recovery point of view is given in [ABFP86], and was the initial impetus for this work.

We conclude that there is possibility of use of our paradigm on conventional software, but many obstacles are present. Considering the amount of presently existing software, the effort to solve these problems might be justified. It is certainly much easier to justify applying these techniques to new software systems, where one can start transformationally from scratch.

10.3.3 Reuse of Components by Design Modification

Software component reuse is often touted as a possible source of major efficiency gains in the software construction process. The popular approach to implementing a component reuse scheme is to provide a library of components, let a potential reuser locate candidate components using some browsing mechanism, and then have the reuser modify the best candidate to fit his application somehow [Dia85]. Few concrete proposals have been made for how this modification process is to take place.

A Design Maintenance System could be of great value for this purpose. Having located a component that has a formal specification and design history, a delta between the desired specification and that of the component could be formed and applied to produce a component with the desired properties. Candidate components could be ranked by the size of the delta, or by an initial estimate of the impact of the delta by carrying through with part of the marking and pruning processes. Kambhampati [Kam89] makes a related observation for reuse of plans. James Neighbors[2] has remarked on the possibility of building large, general components, such as databases and graphics subsystems, and reusing them by stripping away unnecessary generality; a Design Maintenance System would be effective for this purpose. Since removal of generality is usually easier than addition of missing capability, this might be a very effective way to store components.

[2]Personal communication.

Such a reuse scheme could be a valuable addition to a synthesis system, which recursively decomposes a specification into a code fragment with slots containing yet more detailed specifications; each specification would be checked against the reusable component library for an easily modified solution before trying further decompositions.

10.3.4 Relevance to Digital Hardware Design

We are rather dismayed by the apparent separation of software and hardware design systems. It seems rather obvious to us that the distinction between the design of digital hardware and and the design of software is merely that of low-level geometric constraints[3]. The problems of specification, representation and application of implementation choices seem virtually identical. The fact that hardware systems have considerable fine-grain implementation-level parallelism, and most software systems design systems currently handle that poorly, merely reflects on the state of software design and implementation technology; eventually, this problem will need to be solved also for software. There is little in this thesis which is specific just to software. Consequently we believe that the ideas presented in this thesis will serve equally well in the digital design domain.

Remarks about maintaining dusty decks apply equally well to “dusty circuits”. Million-transistor VLSI designs (such as the Intel 486 and Motorola 68040 CPUs) have enough longevity, and certainly enough financial effect if modified incorrectly to make a Design Maintenance System-like tool attractive.

10.4 Future Work

This thesis has presented some solutions to the problems of implementing a Design Maintenance System. Quite a number of future directions for research suggested themselves during the course of our work.

10.4.1 Implementation and Empirical Validation

By far the most obvious need is to implement the ideas and validate them on a transformation system used for practical work. Existing transformation systems

[3]Physical placement of components, wiring layout, sizes of drivers dependent on line length and number of loads, etc.

by and large do not have performance specifications at all, and do not conveniently produce design histories, so one must either build a fresh transformation system or find some way to augment existing systems. Because of the absence of suitable bases for our work, all of our demonstrations have been performed on extremely rough engineering prototypes with no concessions for scale of programs or library sizes, longevity of libraries (i.e., database storage), or operator amenities of any kind. It is difficult to judge even the ability to use such a system, let alone its real utility, in such an environment. A valuable byproduct of testing on real examples is a measurement of the payoff of design maintenance as the problem sizes scale up.

10.4.2 Specification

Considering that we have so many data types (programs, performance measures, transforms, locators, maintenance deltas) and operations (transformationally implement, delta-type-specific revision procedures, etc.) it would probably be worthwhile to construct an algebraic specification of a transformation system with maintenance deltas to provide a secure formal basis for these ideas. Such a specification can serve as a basis for a new implementation of a Design Maintenance System. This exercise is practical, as demonstrated by the algebraic specification of a transformation system done by the CIP-S project [BEH+87].

10.4.3 Self Application

Having a specification, a larger scale validation could be attempted by applying a Design Maintenance System to its own construction. This would have the added benefit of obtaining synergy during the tool construction process[4]. The bootstrap construction of CIP-S from CIP-L shows the value of this approach [BEH+87].

10.4.4 Application to Dusty Decks

The amount of existing software that needs maintaining is simply too enormous to ignore. Given the partial successes of reverse engineering transformationally, augmenting a Design Maintenance System with tools to aid in such a process could be a useful way to extend the utility of a Design Maintenance System while simultaneously testing its limits. Existing plan recognition tools are needed, as well new tools

[4]We refer to leveraged self-application of a tool as an *avalanche* technology, on the grounds that little effects can by self-magnified by the tool.

to recognize the performance goals acheived by plans. We expect that the design recovery process will be significantly aided by the ability to store and revise design histories.

10.4.5 Improvements to Transformational Model

We find our transformational model weak in several respects. First, we have no notion of construction-process oriented metrics or goals, and yet these are part of all practical software construction methods. Addition of process goals also leads to process goal deltas and integration methods.

Considering the value of the notion of *locale* as a program part to TCL for navigation, we would like a better definition, perhaps derived from a topological description of the program representation. Such a definition should allow us to reason directly about whether locales overlap, and therefore determine common cases of noninterference of transformations. Work is also needed on determining useful types of locale-combining operations.

There is the unsatisfying problem of fitting synthesis systems into our model. The CYPRESS synthesis system [Smi85] recursively decomposes a pure performance specification into a functional specification and a set of more detailed performance specifications. While this decomposition step could be treated as a transformation on a state *containing* a specification, it does not match our model because there appears to be no performance specification for the transformation system to maintain as an invariant.

10.4.6 Performance algebras

We believe that explicitly defining what we would call *performance algebras* as systems of computations for performance measures, using algebraic specification techniques, will eventually be needed to allow deep reasoning about the effect of changes on measures, and therefore on goal achievement. The subsumption relation would be a natural part of the algebra, as would any definable performance predicate. Such performance algebras would probably fit very nicely into transformation systems designed around algebraic frameworks such as that of CIP [BEH$^+$87] or PROSPECTRA [KB88].

10.4.7 Dependency networks for transformations

One of the severest problems with our current approach is the assumption of a monolithic state, and the requirement by the ladder building routines of Section 8.3.3 to keep all the intermediate states. Some initial investigation on our part suggests that one could use dependency networks for the derivation history rather than a simple linear chain; XANA [MF89b] in effect does this. One would no longer depend on the idea of a monolithic state; rather, each transformation produces a partial state, like those used in nonlinear planning systems. Partial states could be represented by sets of predicates describing relations between state components. The problem of non-local constraints (results of multiple transformations) interacting to violate preconditions of dependent transformations must be solved [Cha87], [Kam89, page 150]; this is the primary reason we chose not to pursue this approach. As we pointed out earlier, Kambhampati's work on plan reuse [Kam89] looks like a very good starting point.

Such a nonlinear transformation dependency network would make many trivial *SWAP*s actually unnecessary; the large size of a practical derivation history indicates that this should be a very effective optimization. Naturally, the notion of *DEFER* must be retained because dependency nets are conservative; "a depends on b" may only be syntactic and not semantic. The formal characterization of *DEFER* must change because of the change in state representation.

We do not believe that a Design Maintenance System will make new software production virtually instantaneous; rearranging a design history can be expensive in its own right, and design history repair by transformational implementation can also be expensive, perhaps measured in days or months depending on the scale of the change. Dependency networks might make change-cost impact analyses possible, by assuming that the number of transformations dependent, according to the dependency network, on a particular transformation is an estimate of the required work. Such a count can obviously be made without actually changing the network. "What-if" estimates could then be made from proposed deltas.

An additional benefit of dependency networks might be the ability to maintain multiple implementation versions, each sharing much their individual design histories. Different versions would be represented by different consistent frontiers of the dependency network.

10.4.8 Finding commutable transformations

Our methods for functional delta integration depend fundamentally on finding a composably equivalent pair of transformations to replace an existing pair, often by revising just the locaters. Although this thesis theoretically characterizes the generation of such revised locaters as a British Museum Algorithm, one cannot afford to do this in practice. Efficient methods for determining such pairs depends on a deeper understanding of the relation between rewriting, the structure of the objects being rewritten, and the notion of locaters; our prototype system uses knowledge about tree-rewrites with pattern matching by unification, over trees, using paths as locaters, to achieve this kind of efficiency. We intuitively trust, but do not know how to formalize, the *DEFER*ral of an optimization through a refinement, as shown in Figures 7.9 and 7.19. A categorical exploration of rewriting motivated initially by this need has been started [Sri91]. With such understanding, one might be able to generate the portion of procedures such as *DEFER* that handle geometrically overlapping but non-interfering transformations, automatically from descriptions of the topology of the states.

10.4.9 Increasing the Grain Size of commutable elements

While our model of transformations does not include them explicitly, TCL methods are technically transforms; they are definitely partial maps from states to states. We did not consider the idea of applying *DEFER* or *PRESERVE* at the level of method application during functional delta integration. Successful deferral at the method level can avoid investing much larger amounts of energy attempting to defer transformations at lower levels. Considering that methods have postconditions describing the desired effect of the method, we are overlooking a rich source of information that can tell us about possible impacts.

10.4.10 Representation of Program Schemes and Functional Deltas

For our experimental system, we chose a tree representation for programs and conditional tree rewrites as the standard form of transform. Choosing simple tree rewrites implicitly defined our functional deltas to also be tree rewrites. Two independent changes separated by great distance in a tree program unfortunately require a very big tree delta. We briefly considered representing deltas as bags of tree rewrites. But the additional fact that tree representations do not easily lend themselves to commonly-occurring transformations that use information from "far away", such as variable declarations, suggested instead choosing a graph representation for programs and using graph transforms as deltas. Specific techniques to handle commuting graph transformations would be needed. Research outlined in the Section 10.4.8 would be helpful here.

10.4.11 Delta Acquisition

We have assumed that deltas simply appear. It would be convenient if such deltas could be produced by a regular process given a partial implementation and feedback from a customer. Techniques such as Shapiro's *critical experiments* for automatic debugging [Sha83] could perhaps be used to focus attention on erroneous parts of the specification or incorrect transformations. We have already remarked on the possibility of producing functional deltas given an almost applicable transformation in Section 7.5. The KATE system is intended to acquire and check specifications[Fic87]; specification errors could be cast as deltas.

10.4.12 Asynchronous Evolution

Regardless of the power of our tools for constructing software, there will always be ambitious projects requiring more than a single software engineer. Our characterization of a Design Maintenance System assumes a fully synchronous cycle of

$$\textbf{repeat}\ \mathit{CollectDelta}; \mathit{ProcessDelta}\ \textbf{end}$$

With a large number of engineers, this is probably not practical. Methods for coordinating the entry, integration of deltas, and plan repair (TCL execution) all in parallel are likely to be needed. We think there is promise in the database notions of serializable transactions, and in particular in nested transactions [Mos85a], because of the similarity between the notion of atomic transaction and the all-or-useless transaction-like nature of TCL methods.

10.5 Summary

This thesis has explored the problem of maintenance from a transformational perspective. Results of this exploration include:

- Improved transformation system models and mechanisms:
  - A formal model of a transformation system, including performance specifications. Few such models or systems exist.
  - TCL, a metaprogramming language, in which performance goals are stated explicitly and drive the transformational derivation. Other existing metaprogramming languages do not encode performance goals, effectively having procedural semantics for performance specifications.
  - Design history capture for potential explanation of implementation
  - Dependency-directed backtracking (*BANISH*)
  - Partial design repair by agenda item execution
- An architecture for a Design Maintenance System based on:
  - A formal model of transformational maintenance. This is a significant improvement over the ad hoc characterization of maintenance presented by standard software engineering texts.
  - A new classification of maintenance types based on transformation system inputs; this classification tells one precisely what methods are needed to install change. Conventional classification of maintenance types provide no clues as to how to handle the change installation.
  - Theoretical procedures, based on commutativity in the design space, for preserving a significant portion of the design history in the face of a change, and the understanding that what part can be preserved can be determined by explicit use of the change.
  - An empirical demonstration that search spaces, and therefore design spaces, are likely to be highly commutative.
- Insights:
  - That initial implementation and maintenance, which appear to be completely separate lifecycle phases in conventional SE models, are in fact not truly distinguishable.
  - The notions of essential versus artificial modularity.
  - Architecture as decisions which are expensive to remove

This investigation leads us to the conclusion that a Design Maintenance System based on these ideas might well be practical, and has the potential for revolutionizing the way in which software is built and maintained. Coupled with the notion of Design Recovery this work might ultimately lead to practical systems for maintaining software not constructed transformationally.

Appendix A
Notation

We summarize here the notation used throughout the thesis.

Generally, calligraphic letters represent universes, capital letters represent specific sets, and lower case letters represent set elements.

$\supset$ means logical implication.

$\subset$ means "subset of".

$\mathcal{F}$ = set of all possible program schemes.

$f_i \in \mathcal{F}$ is a particular program scheme.

$?z$ refers to scheme variable z.

f_G is a program satisfying predicate G.

$\epsilon \in \mathcal{F}$ is the "empty" program, **skip**.

$\mathcal{Q}$ are possible facts inferred about programs.

q is a fact.

$q_{i,j}$ are *consequences*, or deductions, drawn about a particular f_i

$Q = \{q\}$ is a set of facts.

$\mathcal{S}$ = set of states in the design space.

$s_i \in \mathcal{S}$ is a state, consisting of a pair $\langle f, Q \rangle$, where f is a program and Q is a set of cached inferences about f.

$\vdash^* \mathit{facts}(f)$ is the theory of f, the transitive closure of the deducibility relation $\vdash$.

$\mathcal{V}_i$ is the set of performance values computable by performance value function p_i.

$\mathcal{P}$ = set of *performance measuring functions* $p_i : \mathcal{S} \to \mathcal{V}_i$.

$\mathcal{G}$ = set of *performance goals*.

$G_i : \mathcal{S} \to \mathit{boolean}$ is a performance goal.

$g_i \in \mathcal{G} : \mathcal{S} \to \mathit{boolean}$ is a performance predicate.

(Notation, continued)

$\mathcal{I}$ is an arbitrary set of possible identifiers

$\mathcal{B}$ = set of *bindings*, or indicators of specific places in the state where it is legitimate to apply a transform.

$b_i \in \mathcal{B}$ are particular bindings.

$\mathcal{L}$ = set of *locaters* or *locales*: binding constraints.

$\ell \in \mathcal{L}$ is a particular locater

$\mathcal{X}$ is the set $\mathcal{T} \times \mathcal{L}$ of transformations.

$x \in \mathcal{X}$ is a transformation.

lv is the name of a method variable capable of holding a locale value.

l is a particular locale.

$\mathcal{T}$ = set of *transforms* possible, with members denoted t_i.

$t_i \in \mathcal{T}$ is transform i, a function $t_i : \mathcal{S} \times \mathcal{B} \to \mathcal{S}$ (partial functions). For simple transformational models in which the state consists solely of a program (as in most extant transformation systems) then $t_i : \mathcal{F} \times \mathcal{B} \to \mathcal{F}$.

$match(t, s, \ell)$ is the subset of arrows leaving s selected by ℓ.

$apply(t, s, \ell)$ follows one arrow from S to some S'.

t_i^ℓ is a transformation, i.e., a transform i with *locater* ℓ, a function $t_i^\ell : \mathcal{S} \to \mathcal{S}$

$defined(t_i^\ell(s))$ is a predicate which is true if $t_i^\ell(s)$ is well defined, and false otherwise.

$f_i \Longrightarrow f_j$ means that program f_i is transformed to program f_j

$\mathcal{C}_i \subseteq \mathcal{T}$ is the set of G_i-preserving, or p_i-preserving transforms. Individual members are denoted $c_j \in \mathcal{C}_i$.

$\mathcal{N}_i = \mathcal{T} - \mathcal{C}_i$ is the set of non-property-preserving transforms with respect to goal G_i. Individual members are denoted $n_j \in \mathcal{N}_i$.

$\mathcal{M}$ is the set of all possible hueristic *methods* used to guide the design process.

$M \subseteq \mathcal{M}$ is a specific set of methods. A specific set of methods used to implement a particular specification is called a *metaprogram*.

$m_i = \langle i, a, G \rangle \in \mathcal{M}$ is a specific method consisting of a identifier i, action a and a postcondition G.

(Notation, continued)

$[t_i^j \cdots]$ represents a *sequence* of transformations, i.e., a *derivation history.*

$\mathcal{H}$ is the set of possible derivation histories.

H is a *derivation history.* This is a triple $\langle k, H^{\mathcal{T}} : 1..k \rightarrow \mathcal{T}, H^{\mathcal{L}} : 1..k \rightarrow \mathcal{L}\rangle$, representing a sequence of transformations $[x_1, x_2, \cdots x_k]$, where $x_i = t_{H^{\mathcal{T}}(i)}^{H^{\mathcal{L}}(i)}$.

$length(H) = k$ is the length of a derivation history $H = \langle k, H^{\mathcal{T}}, H^{\mathcal{L}}\rangle$

$H[i]$ is the transformation $t_{H^{\mathcal{T}}(i)}^{H^{\mathcal{L}}(i)}$

$H[i..j]$ is a subsequence $[t_{H^{\mathcal{T}}(i)}^{H^{\mathcal{L}}(i)}, \ldots, t_{H^{\mathcal{T}}(j)}^{H^{\mathcal{L}}(j)}]$

$rest(H, i) \equiv H[i..length(H)]$ is the tail of H.

$H_1 \subset H_2 \equiv \exists i, j \mid H_1 = H_2[i..j]$

$H_1 + H_2 \equiv [H_1[1], \ldots, H_1[length(H_1)], H_2[1], \ldots, H_2[length(H_2)]]$

$\Pi(H)(f_0) = f_{length(H)}$ is the program achieved by computing $f_i = t_{H^{\mathcal{T}}(i)}^{H^{\mathcal{B}}(i)}(f_{i-1})$ for $i = 1 \cdots length(H)$.

$\mathcal{D}$ is a *design history*, = H plus unfolded goal plan.

$\mathcal{R} \subseteq \mathcal{H}$ is the set of *refinement histories*, consisting of those transformations which add detail, i.e., enlarge the theory of the specification.

$R \in \mathcal{R}$

(Notation, continued)

Δ_{type} represents the set of values forming δ_{type} associated with a function $REVISE_{type}$.

$REVISE_{type} : \textbf{object}_{type} \times \Delta_{type} \rightarrow \textbf{object}_{type}$ is a function which revises an instance of $\textbf{object}_{type}$ according to a delta of that type.

$\delta \in \Delta_{type}$ is a particular change. We write $\delta(d)$ to mean $REVISE_{type(\delta)}(d, \delta)$.

Δ_G is the type (set) of changes to a performance specification

Δ_v is the type (set) of changes to performance bounds

Δ_f is the type (set) of changes to a functional specification

$\Delta_{\mathcal{G}}$ is the type (set) of changes to performance goal library

$\Delta_{\mathcal{C}}$ is the type (set) of changes to property-preserving sets of transformations

$\Delta_{\mathcal{P}}$ is the type (set) of changes to the performance measure library

$\Delta_{\mathcal{V}}$ is the type (set) of changes to the sets of performance values

$\Delta_{\succeq}$ is the type (set) of changes to the subsumption relations between performance values

$\Delta_{\mathcal{M}}$ is the type (set) of changes to the method library

$\mathcal{A}$ is the set of possible actions of a method.

$a \in \mathcal{A}$ is a specific action.

$\langle i_1 : e_1,\ i_2 : e_2,\ \ldots,\ i_k : e_k \rangle$ defines a tuple with slots named $i_1, i_2, \ldots$

$e.i$ refers to the value of the tuple slot named i of the tuple e.

Appendix B
Procedure for Integrating Δ_f into a Derivation History

The code in this section gives an abstract procedural description of view of Δ_f integration into derivation histories. It models the generation of deltas by a customer, and the integration of those deltas into (revised) derivation histories, preserving as many of the transformations as possible. It is not intended to be efficient; its purpose is to convey the intent. Performance goals are not handled (see section ?).

The program design language is intended to be a relatively conventional block-structured procedural language, that can manipulate records and sequences as entities. Most of the constructs should be self-explanatory, but, here are a few notes about the more esoteric aspects:

- Keywords are boldface: **Declare If Then Else Fi Procedure Function Returns Return Guard**
- Comments begin with % and their italicized content continues to the end of the line:
  % Comment
- One dimensional arrays/sequences are 1-origin indexable. A subsequence can be selected by writing $sequence[m..n]$ with $sequence[m]$ being shorthand for $sequence[m, m]$. The function *length* returns the length of a sequence. $rest(sequence, n)$ is the same as $sequence[n..length(sequence)]$. Sequences can be concatenated via the "+" operator.
- Records are formed by the expression

  $$\langle slot1 : slot1exp, slot2 : slot2exp, ..., slotn : slotnexp\rangle$$

  where fields are separated by commas, the name of record field appears to the left of a colon, and the value to fill that field is to the right of a comma. The slot names act as record access functions in the notation *exp.slotname*. A record can act as a sequence of length 1.

- $\langle name, name, ...\rangle := exp$ means multiple assignment from a multiple- or record-valued expression. Think of this as record disassembly.
- A **Guard** block consists of a sequence of *predicate: action* clauses; it nondeterministically executes just one of the actions for which the predicate in the clause is true (Dijkstra's guarded conditional). The **Guard** block in *SoftwareLifecycle* below is intended to model nondeterminism on the part of the system analyst.

A derivation history is represented as a sequence of transformations.

A quick summary of the procedures:

SoftwareLifeCycle captures the process of building and maintaining a particular program. It is shown only to provide a sense of the kinds of actions a software engineer might request of a Design Maintenance System.

ImplementProgram takes a program and returns either an implementation and its generating derivation history or a failure signal.

Integrate integrates a delta into a history, returning a new implementation and history, or a failure signal.

BANISH gets rid of the transformation at the head of a history by rearranging the history so that the offending transformation is delayed as long as possible; then the offending transformation and all transformations which depend on it are chopped off. Banishment always shortens the derivation history by at least 1 transformation.

DeferTransformation attempts to delay an applied transformation until after its present successor in a derivation history has been applied. It returns revised bindings, and a possibly-revised delayed transform.

SwapTransformations attempts to exchange two transformations that are adjacent in a derivation history. It returns revised bindings for the exchanged transformations.

PreserveTransformation attempts to push a delta past a transformation already present in a derivation history. It returns revised bindings for the already-present transformation, and a possibly revised delta with possibly-revised bindings.

Ship releases a program for use by the customer.

```
Procedure SoftwareLifeCycle()
  Declare DerivationHistory: History, RevisedHistory, EmptyHistory
  Declare Transformation: FunctionalDelta
  Declare Integer: View, Boolean: SuccessFlag
  Declare Program: ProgramAtView, EmptyProgram
  Declare Program: Implementation, PartialImplementation
  EmptyProgram:=nil; History:=EmptyHistory:=nil
  View:=0 % Selects where along history functional deltas will get applied
  ProgramAtView:=EmptyProgram % Make program consistent with program at view
  Ship(ProgramAtView) % Ship the 1st prototype to customer, just to be systematic
  Repeat
    Guard % Let software engineer choose what he wants to do next
      View<length(History): % Move SE's view later in history
        Begin
          View:=View+1
          ProgramAtView:=ApplyTransformation(History[View],ProgramAtView)
        End
      View>0: % Move SE's view earlier in history
        Begin
          View:=View-1
          ProgramAtView:=ApplyTransformation(Π(History[1..View]),
            EmptyProgram)
            % Recompute program as seen at this view point
        End
      True: % Apply functional delta anywhere in history
        Begin
          FunctionalDelta:=
            ChooseRandomTransformation(ProgramAtView) % new requirement
          ⟨SuccessFlag,Implementation,RevisedHistory⟩:=
              Integrate(ProgramAtView,FunctionalDelta,rest(History,View+1))
          If SuccessFlag Then
            History:=History[1..View]+FunctionalDelta+RevisedHistory
            View:=View+1 % Default additional changes to "additive"
            ProgramAtView:=ApplyTransformation(FunctionalDelta,ProgramAtView)
            Ship(Implementation) % where most organizations stop
          Else
            Print "Can't implement that."
            ?? % Perhaps the new delta violates an existing delta,
            % perhaps we should complain, and require that the
            %  existing delta be explicitly deleted before proceeding.
          Fi
        End
```

```
% SoftwareLifeCycle, continued...
      View<length(History): % Delete a user requirement
        Begin
          ⟨PartialImplementation,RevisedHistory⟩:=
              BANISH(ProgramAtView,rest(History, View+1))
          ⟨SuccessFlag,Implementation,NewHistory⟩:=
              Implement(PartialImplementation)
          If SuccessFlag Then
            History:=History[1..View-1]+RevisedHistory+NewHistory
            Ship(Implementation)
          Else
            % Can't implement program derived from history
            % with deleted transformation
            % Attempt to use as much of history as possible
            ⟨SuccessFlag,Implementation,NewHistory⟩:=
              Reimplement(ProgramAtView,RevisedHistory)
              If SuccessFlag
              Then
                History:=History[1..View-1]+NewHistory
                Ship(Implementation)
              Else
                Print "Not implementable that way."
              Fi
          Fi
        End
      View>1: % Change priority of user requirements
        Begin
          % Move existing delta to left of a correctness-preserving transform
          % This can swap program deltas, too!
          ⟨SuccessFlag,DeferredTransformation,PromotedTransformation⟩:=
            DeferTransformation(ProgramAtView,History[View-1],History[View])
          If SuccessFlag Then
            % We have rearranged order of transformations.
            History[View-1]:=PromotedTransformation
            History[View]:=DeferredTransformation
          Else Print "Can't exchange."
        End
    End Guard
  End Repeat
End SoftwareLifeCycle
```

```
Function Implement(Program: S)
  Returns ⟨Boolean,Program,DerivationHistory⟩
  % Determines an implementation, returns a success flag
  % and the history of transformation steps to obtain the implemented program
  % This is the conventional scheme for transformational implementation.
  Declare Boolean: SuccessFlag
  Declare Program: Implementation
  Declare DerivationHistory: RestOfHistory
  If Implemented(S) Then Return ⟨True,S,EmptyHistory⟩
  Enumerate Transformation:T suchthat Applicable(T,S)
    ⟨SuccessFlag,Implementation,RestOfHistory⟩:=
        Implement(ApplyTransformation(T,S))
    If Success Then Return ⟨True,Implementation,T+RestOfHistory⟩
  End Enumerate T
  Return ⟨False,junk,junk⟩
End Implement

Function Reimplement(Program: S, DerivationHistory: History)
  Returns ⟨Boolean,Program,DerivationHistory⟩
  % Determines an implementation, returns a success flag
  % and the history of transformation steps to obtain the implemented program
  % Attempts to reuse the derivation history.
  % This is the "naive" version of derivation replay with backtracking.
  Declare Boolean: SuccessFlag
  Declare Program: Implementation
  Declare DerivationHistory: RestOfHistory
  If Implemented(S) Then Return ⟨True,S,EmptyHistory⟩
  If length(History)=0 Then Return Implement(S)
  If not Applicable(History[1],S) Then Return Implement(S)
    % If History was once valid for S, then Applicable is always true.
    % This test simply makes Reimplement robust in face of old histories.
  % First element of history applies. Try to use it.
  ⟨SuccessFlag,Implementation,RestOfHistory⟩:=
      Reimplement(ApplyTransformation(History[1],S),rest(History,2))
  If SuccessFlag Then Return ⟨True,Implementation,History[1]+RestOfHistory⟩
  Else
    % First element of history leads to program which is unimplementable
    Return Implement(S)
  Fi
End Reimplement
```

```
Function Integrate(Program: CurProgram, Transformation: Delta,
    DerivationHistory: History)
  Returns ⟨Boolean,Program,DerivationHistory⟩
  % Constructs a new implementation and history for the
  % program defined by ApplyTransformation(Delta,CurProgram) ...
  % by revising the DerivationHistory of CurProgram to integrate Delta
  Declare Program: Implementation, PartialImplementation
  Declare DerivationHistory: RevisedHistory, Boolean: SuccessFlag
  Declare Transformation: PreservedTransformation, RevisedDelta
  If length(History)>0
    And Not ConventionalTransformationalImplementation
  Then
    % Try to Reuse history to derive new implementation
    ⟨SuccessFlag,PreservedTransformation,RevisedDelta⟩:=
        PreserveTransformation(CurProgram,History[1],Delta)
    If SuccessFlag Then
      % We were able to preserve the original transformation
      ⟨SuccessFlag,Implementation,RevisedHistory⟩:=
          Integrate(ApplyTransformation(History[1],CurProgram),
              RevisedDelta,rest(History,2)) % revise the rest!
      If SuccessFlag Then
        % Success at revising history and obtaining an implementation
        Return ⟨True,Implementation,
            PreservedTransformation+RevisedHistory⟩
      Else
        % Not able to revise history and obtain an implementation.
        % Perhaps we can get an implementation from CurProgram.
        % If not, it is hopeless from here.
        Return Implement(ApplyTransformation(Delta,CurProgram))
      Fi
    Else
      % Can't preserve History[1] because of some inability to resolve conflict...
      % with the desired Delta so make History[1] stop bothering us.
      ⟨PartialImplementation,RevisedHistory⟩:=
        BANISH(CurProgram,History)
        % ignore PartialImplementation
      Return Integrate(CurProgram,RevisedHistory,Delta)
      % Won't loop: BANISH chops off offending transformation
    Fi
  Else
    % No more revision possible, nothing left to revise.
    Return Implement(ApplyTransformation(Delta,CurProgram))
  Fi
End Integrate
```

```
Function BANISH(Program: CurProgram,
      DerivationHistory: History)
   Returns ⟨Program,History⟩
  % This function pushes History[1] as deep into the history as possible,
  % chops the history off at that point, and returns the revised history.
  % Because we always chop the history off, banishing cannot fail;
  % at worst it returns an empty history.
  % Complication: History[1] may conflict with History[2], so we can't always
  % immediately get rid of History[1]; we solve this by (recursively)
  % getting rid of History[2] and then proceeding.
  % This procedure costs O(length(History)$^2$) to run.
  Declare Program: PartialImplementation, Boolean: SuccessFlag
  Declare DerivationHistory: RevisedHistory
  Declare Transformation: PromotedTransformation, DeferredTransformation
  Assert length(History) $\geq$ 1 % Or there's nothing to banish!
  If length(History)=1 Then Return EmptyHistory
  ⟨SuccessFlag,DeferredTransformation,PromotedTransformation⟩:=
      DeferTransformation(CurProgram,History[1],History[2])
  If SuccessFlag Then
    % We can move transformation to banish to History[2].
    % Pretend we did that, and banish it from there.
    ⟨PartialImplementation,RevisedHistory⟩:=
      BANISH(ApplyTransformation(PromotedTransformation,CurProgram),
              DeferredTransformation+rest(History,3))
    Return ⟨PartialImplementation,
        PromotedTransformation+RevisedHistory⟩
  Else
    % Transformation we wish to banish is blocked by rightmost neighbor.
    % So banish rightmost neighbor, shortening history, and try again.
    % Safe to banish rightmost neighbor for two reasons:
    % 1) This procedure can be conservative (because the Revise
    %    procedure will work even if Banish throws away everything!
    % 2) The rightmost neighbor depends on transformation we are trying to banish;
    %    if we succeed in banishing it, the rightmost neighbor's preconditions
    %    will not be present, and the rightmost neighbor can't be saved either.
    ⟨PartialImplementation,RevisedHistory⟩:=
        BANISH(ApplyTransformation(History[1],CurProgram),rest(History,2))
        % ignore PartialImplementation
    Assert length(RevisedHistory)<length(History)-1
    Return BANISH(CurProgram,History[1]+RevisedHistory)
  Fi
End BANISH
```

```
Function DeferTransformation(Program: Before,
    Transformation: $t_1^{\ell_1}$, Transformation: $t_2^{\ell_2}$)
  Returns ⟨Boolean, Transformation, Transformation⟩
  % Find $t_3$, $\ell'_1$ and $\ell'_2$ such that $t_2^{\ell_2}(t_1^{\ell_1}(Before)) = t_3^{\ell'_1}(t_2^{\ell'_2}(Before))$
  % Here we have a British Museum Algorithm, to simplify understanding.
  % This can be done much more efficiently for any particular transform type,
  % e.g., tree transforms.
  % Note that result may not be unique.
  ⟨SuccessFlag, DeferredTransformation, PromotedTransformation⟩:=
      SwapTransformations(Before, $t_1^{\ell_1}$, $t_2^{\ell_2}$) % try the easy case
  If SuccessFlag Then Return ⟨true, DeferredTransformation, PromotedTransformation⟩
  Else
    % Can't simply swap the transformations.
    Enumerate $t_3$
      Enumerate $\ell'_1$
        Enumerate $\ell'_2$
          If ApplyTransformation($t_2^{\ell_2}$, ApplyTransformation(${t_1}^{\ell_1}$, Before))
              = ApplyTransformation($t_3^{\ell'_1}$, ApplyTransformation($t_2^{\ell'_2}$, Before))
          Then Return ⟨true, $t_3^{\ell'_1}$, $t_2^{\ell'_2}$⟩
        End Enumerate $\ell'_2$
      End Enumerate $\ell'_1$
    End Enumerate $t_3$
    Return ⟨false, junk, junk⟩
  Fi
End DeferTransformation

Function SwapTransformations(Program: Before,
    Transformation: $t_1^{\ell_1}$, Transformation: $t_2^{\ell_2}$)
  Returns ⟨Boolean, Transformation, Transformation⟩
  % Find $\ell'_1$ and $\ell'_2$ such that $t_2^{\ell_2}(t_1^{\ell_1}(Before)) = t_1^{\ell'_1}(t_2^{\ell'_2}(Before))$
  % Here we have a British Museum Algorithm, to simplify understanding.
  % This can be done much more efficiently for any particular transform type,
  % e.g., tree transforms.
  Enumerate $\ell'_1$
    Enumerate $\ell'_2$
      If ApplyTransformation($t_2^{\ell_2}$, ApplyTransformation(${t_1}^{\ell_1}$, Before))
          = ApplyTransformation($t_1^{\ell'_1}$, ApplyTransformation($t_2^{\ell'_2}$, Before))
      Then Return ⟨true, $t_1^{\ell'_1}$, $t_2^{\ell'_2}$⟩
    End Enumerate $\ell'_2$
  End Eumerate $\ell'_1$
  Return ⟨false, junk, junk⟩
End SwapTransformations
```

```
Function PreserveTransformation(Program: CurProgram,
    Transformation: $c_i^{\ell_1}$, Transformation: $t^{\ell_2}$)
  Returns ⟨Boolean, Transformation, Transformation⟩
  % Attempts to preserve a transformation in the face of a functional delta.
  % This function computes new binding $\ell_1'$ for property-preserving transform $c_i$,
  % a possibly new $t_{new}$ and new binding $\ell_2'$ such that:
  %    $c_i^{\ell_1'}(t^{\ell_2}(CurProgram)) = t_{new}^{\ell_2'}(c_i^{\ell_1}(CurProgram)$, or returns failure.
  % Here we have a British Museum Algorithm, to simplify understanding.
  % This can be done much more efficiently for any particular transform type,
  % e.g., tree transforms.
  Enumerate $t_{new}$ % try replacements for $t$
    Enumerate $\ell_1'$ % try new binding sites for $c_i$
      Enumerate $\ell_2'$ % try binding sites for $t_{new}$
        If ApplyTransformation($c_i^{\ell_1'}$,ApplyTransformation($t^{\ell_2}$,CurProgram))
          = ApplyTransformation($t_{new}^{\ell_2'}$,ApplyTransformation($c_i^{\ell_1}$,CurProgram))
        Then Return ⟨true,$c_i^{\ell_1'}$,$t_{new}^{\ell_2'}$⟩
      End Enumerate $\ell_2'$
    End Enumerate $\ell_1'$
  End Enumerate $t_{new}$
  Return ⟨false,junk,junk⟩
End PreserveTransformation
```

Appendix C
Algebras used in linear replay example

The following algebras provide the domain axioms, and therefore the transformations used in Figure 7.21.

```
stack =     LAMBDA trivial IS
            trivial WITH data RENAMED element
            UNION
            ALGEBRA
            SORTS:  stack, element
            OPS:    empty → stack
                    push(stack,element) → stack
                    pop(stack) → stack
                    top(stack) → element
                    2nd(stack) → element
            EQNS:   pop(push(stack,element))=stack
                    top(push(stack,element))=element
                    2nd(push(push(stack,element2),element1))=element2
```

Figure C.1: The stack algebra

```
tinylisp =  ALGEBRA
            SORTS:  atom, seq
            OPS:    listify(atom) → seq
                    atomize(seq) → atom
                    nil → seq
                    cons(seq,seq) → seq
                    car(seq) → seq
                    cdr(seq) → seq
                    cadr(seq) → seq
                    list(seq) → seq
            EQNS:   atomize(listify(atom))=atom
                    listify(atomize(seq))=seq
                    car(cons(seq1,seq2))=seq1
                    cdr(cons(seq1,seq2))=seq2
                    cadr(seq)=car(cdr(seq))
                    cons(seq,nil)=list(seq)
```

Figure C.2: Algebra for Lisp fragment

Bibliography

[ABFP86] Guillermo Arango, Ira Baxter, Peter Freeman, and Christopher Pidgeon. TMM: Software Maintenance by Transformation. *IEEE Software*, 3(3):27–39, May 1986.

[Agr86] William W. Agresti. What are the New Paradigms? In William W. Agresti, editor, *New Paradigms for Software Development*. IEEE Press, 1986. ISBN 0-8186-0707-6.

[AHT90] James Allen, James Hendler, and Austin Tate, editors. *Readings in Planning*. Morgan Kaufmann, San Mateo, California, 1990. ISBN 1-55860-130-9.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1974. ISBN 0-201-00029-6.

[AI87] Jose A. Ambros-Ingerson. Integrating Planning, Execution and Monitoring. Master's thesis, Department of Computer Science, University of Essex, 1987.

[AIS88] Jose A. Ambros-Ingerson and Sam Steel. Integrating Planning, Execution and Monitoring. In *Proceedings of AAAI-88*, Minneapolis, August 1988.

[All86] Lloyd Allison. *A Practical Introduction to Denotational Semantics*. Cambridge University Press, Cambridge, England, 1986.

[All90] Dean T. Allemang. *Understandings Programs as Devices*. PhD thesis, Ohio State University, Columbus, Ohio, 1990.

[AM75] Michael A. Arbib and Ernest G. Manes. *Arrows, Structures, and Functors: The Categorical Imperative*. Academic Press, New York, 1975.

[Ara88] Guillermo Arango. *Domain Engineering for Software Reuse*. PhD thesis, Department of Information and Computer Science, University of California at Irvine, July 1988. Available as Advanced Software Engineering Project RTP086.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Massachusetts, 1986. ISBN 0-201-10088-6.

[Bal85a] Robert Balzer. A 15 Year Perspective on Automatic Programming. *IEEE Transactions on Software Engineering*, SE-11(11):1257–1268, November 1985.

[Bal85b] Robert Balzer. Automated Enhancement of Knowledge Representations. In *Proceedings of Ninth International Joint Conference of Artificial Intelligence*, pages 203–207, Los Angeles, August 1985. ISBN 0-934613-02-8.

[Bar88] David Barstow. Automatic Programming for Streams II: Transformational Implementation. In *Proceedings of 10th International Conference on Software Engineering*, pages 439–447, Singapore, April 1988. IEEE. ISBN 0-8186-0849-8.

[Bar89] David Barstow. Automatic Programming for Device Control Software. Technical Report (unnumbered), Schlumberger Laboratory for Computer Science, PO Box 200015, Austin, Texas 78720-0015, 1989.

[Bau77] F. L. Bauer. Notes on the project CIP: Outline of a transformation system. Technical Report TUM-INFO-7729, Institut fur Informatik, Technische University Munchen, Munich, West Germany, 1977.

[Bax86] Ira Baxter. Domain Connection Discovery. Technical Report STP-108-87. Also available as RTP067, Advanced Software Engineering Project, Information and Computer Sciences Department, University of California at Irvine, MCC Corporation, Software Technology Program, October 1986.

[Bax87a] Ira D. Baxter. PCL: A Production Control Language (A Proposal). Technical Report STP-375-87, Microelectronics and Computer Technology Corporation, Software Technology Program, September 1987. PCL is an early version of TCL. Also available as RTP080, Advanced Software Engineering project, Department of Information and Computer Science, University of California at Irvine, Irvine, California.

[Bax87b] Ira D. Baxter. Propagation of Change In Transformational Systems. Technical Report RTP076, University of California at Irvine, Information and Computer Sciences Department, Advanced Software Engineering, February 1987.

[Bax88] Ira D. Baxter. Lexical Searching: Reducing Search in Nearly Decomposable Spaces. Technical Report RTP096, Advanced Software Engineering Project, Department of Information and Computer Science, University of California at Irvine, December 1988.

[BBG$^+$78] F. L. Bauer, M. Broy, R. Gnatz, H. Partsch, P. Pepper, and H. Wossner. Towards a wide spectrum language to support program specification and program development. *SIGPLAN Notices*, 13(12):15–23, 1978.

[BCC89] P. Benedusi, A. Cimitile, and U. De Carlini. A Reverse Engineering Methodology to Reconstruct Hierarchical Data Flow Diagrams for Software Maintenance. In *Proceedings of Conference on Software Maintenance 1989*, pages 180–189, Miami, Florida, October 1989. IEEE Computer Society Press. ISBN 0-8186-1965-1, IEEE Catalog Number 89CH2744-1.

[BD77] R. M. Burstall and John Darlington. A Transformational System for Developing Recursive Programs. *Journal of ACM*, 24(1):44–67, January 1977.

[BEH$^+$87] F. L. Bauer, H. Ehler, A. Horsch, B. Moller, H. Partsch, O. Paukner, and P. Pepper. *The Munich Project CIP*. Springer-Verlag, 1987. Lecture Notes in Computer Science 292.

[BF81] Avron Barr and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence, Volume I*. William Kaufmann, Inc., Los Altos, California, 1981. ISBN 0-86576-005-5.

[BFKM85] Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-based Programming*. Addison-Wesley, 1985. ISBN 0-201-10647-7.

[BGW82] R. Balzer, N. Goldman, and D. Wile. Operational Specification as the Basis for Rapid Prototyping. *ACM Sigsoft Software Engineering Notes*, 7(5):3–16, December 1982.

[Big88] Ted J. Biggerstaff. Design Recovery for Maintenance and Reuse. Technical Report STP-378-88, Software Technology Program, Microelectronics and Computer Corporation, November 1988. Also published in IEEE Computer, July 1989.

[Big89a] Ted J. Biggerstaff. Design Recovery for Maintenance and Reuse. *IEEE Computer*, 22(7):36–49, July 1989. Also available as MCC Technical Report STP-378-88.

[Big89b] Ted J. Biggerstaff. DESIRE: A System for Design Recovery. Technical Report STP-081-89, Microelectronics and Computer Corporation, 1989.

[BM84] J. M. Boyle and M. N. Muralidharan. Program Reuseability through Program Transformation. *IEEE Transactions on Software Engineering*, SE-10(5):575–588, 1984.

[BMPP89] Fredrich Ludwig Bauer, Bernhard Moller, Helmut Partsch, and Peter Pepper. Formal Program Construction by Transformations– Computer-Aided, Intuition-Guided Programming. *IEEE Transactions on Software Engineering*, 15(2):165–180, February 1989.

[Boe81] Barry Boehm. *Software Engineering Economics*. Prentice-Hall, New Jersey, 1981. ISBN 0-13-822122-7.

[Bor89] Ellen Ariel Borison. *Program Changes and the Cost of Selective Recompilation*. PhD thesis, Carnegie Mellon University, 1989.

[Boy84] James. M. Boyle. Lisp to FORTRAN – Program Transformation Applied. In Peter Pepper, editor, *Program Transformation and Programming Environments*, pages 291–298. Springer-Verlag, New York, 1984.

[BPPW80] M. Broy, H. Partsch, P. Pepper, and M. Wirsing. Semantic Relations in Programming Lanaguages. In S. H. Lavington, editor, *Information Processing 80*, pages 101–106, New York, 1980. North-Holland Publishing Company.

[BPW80] Manfred Broy, Peter Pepper, and Martin Wirsing. On Relations Between Programs. In *Proceedings of the 4th International Symposium on Programming*, pages 59–78. North-Holland, April 1980. Lecture Notes in Computer Science #83.

[Bro75] Fred P. Brooks, Jr. *The Mythical Man-Month.* Addison-Wesley Publishing Co., Reading, Massachusetts, 1975.

[BS84] Bruce G. Buchanan and Edward H. Shortliffe. *Rule-Based Expert Systems.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1984. ISBN 0-201-10172-6.

[BU86] Boumediene Belkhouche and Joseph E. Urban. Direct Implementation of Abstract Data Types from Abstract Specifications. *IEEE Transactions on Software Engineering*, SE-10(5):649–661, May 1986.

[Car85] J. Carbonell. Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition. Technical Report CMU-CS-85-115, Carnegie-Mellon University, 1985. Also available in Machine Learning: An Artificial Intelligence Approach, R. Michalski, J. Carbonell and T. Mitchell, eds., pages 371-392, Morgan Kaufmann, Los Altos, CA 1986.

[CB88] J. Conklin and M. Begeman. gIBIS: A Tool for Exploratory Policy Discussion. *ACM Transactions on Office Management Systems*, October 1988.

[CB89] J. Conklin and M. Begeman. gIBIS: A Tool for all Reasons. *American Society for Information Science*, pages 200–213, May 1989. MCC Technical Report Number STP-252-88.

[CC90] Elliot J. Chikofsky and James H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1), January 1990.

[Cha87] David Chapman. Planning for Conjunctive Goals. *Artificial Intelligence*, 32(3):333–377, July 87.

[Che84] Thomas E. Cheatham, Jr. Reusability Through Program Transformation. *IEEE Transactions on Software Engineering*, SE-10(5):589–594, September 1984.

[CHT81] T. E. Cheatham, Jr., G. H. Holloway, and J. A. Townley. Program refinement by transformation. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 430–437, San Diego, California, March 1981. Reprinted in **New Paradigms for Software Development**, William W. Agresti, ed., IEEE, 1986, ISBN 0-8186-0707-6.

[Cle88] J. Cleaveland. Building Application Generators. *IEEE Software*, 5(6):25–33, July 1988.

[CM85] Eugene Charniak and Drew McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1985. ISBN 0-201-11945-5.

[Con87] Jeff Conklin. A Survey of Hypertext. Technical Report STP-356-86 (Rev. 2), MCC Software Technology Program, December 1987.

[CRM79] E. Charniak, C. Riesbeck, and D. McDermott. Data Dependencies. In *Artificial Intelligence Programming*, chapter 16. L. E. Erlbaum, Baltimore, 1979.

[CS89] A. Colbrook and C. Smythe. The Retrospective Introduction of Abstraction into Software. In *Proceedings of Conference on Software Maintenance 1989*, pages 166–173, Miami, Florida, October 1989. IEEE Computer Society Press. ISBN 0-8186-1965-1, IEEE Catalog Number 89CH2744-1.

[CSM$^+$79] B. Curtis, S. B. Sheppard, P. Milliman, M.A. Borst, and T. Love. Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstad and McCabe Metrics. *IEEE Transactions Software Enginering*, SE-5(2), March 1979.

[CT85] Ken Currie and Austin Tate. O-plan: Control in the open planning architecture. *Expert Systems*, 85:225–240, 1985. Reprinted in Readings in Planning, J. Allen, J. Hendler and Austin Tate, eds., 1990, Morgan Kaufmann, San Mateo, California, ISBN 1-55860-130-9, pages 361-368.

[CTH79] T. E. Cheatham, Jr., J. A. Townley, and G. H. Holloway. A System for Program Refinement. In *Proceedings of the 4th International Conference on Software Engineering*, pages 53–63, September 1979.

[dDR$^+$78] Johan deKleer, Jon Doyle, Charles Rich, Guy L. Steele Jr., and Gerald Jay Sussman. AMORD, a Deductive Procedure System. Technical Report MIT AI Memo 435, Massachusetts Institute of Technology, January 1978.

[dDSS77] Johan deKleer, Jon Doyle, Guy L. Steele, and Gerald Jay Sussmann. AMORD: Explicit Control of Reasoning. In *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, pages 116–125. ACM, August 1977. SIGPLAN Notices 12(8) and *SIGART Newsletter*, No. 64; reprinted in **Readings in Knowledge Representation**, Morgan Kaufmann Publishers, Inc., 1985, pp. 345-356.

[deK84] Johan deKleer. Choices without Backtracking. In *Proceedings of AAAI-84*, pages 79–85, University of Texas at Austin, Austin, Texas, August 1984. AAAI.

[deK86a] Johan deKleer. An Assumption-Based Truth Maintenance System. *Artificial Intelligence*, 28(2):127–162, March 1986.

[deK86b] Johan deKleer. Problem Solving with the ATMS. *Artificial Intelligence*, 28(2):197–224, March 1986.

[DFM90] Thomas Dean, R. James Firby, and David Miller. Hierarchical planning involving deadlines, travel time and resources. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 369–386. Morgan Kaufmann, San Mateo, California, 1990. ISBN 1-55860-130-9 (reprinted from Computational Intelligence, Volume 4 Number 4, pp. 381-398).

[Dia85] Ruben Prieto Diaz. *A Software Classification Scheme.* PhD thesis, University of California at Irvine, 1985.

[DKMW89] F. Daube, E. Kant, W. MacGregor, and J. Wald. Automatic Synthesis of Finite Difference Programs. Technical Report (unnumbered), Schlumberger Laboratory for Computer Science, PO Box 200015, Austin, Texas 78720-0015, 1989.

[Doy78] Jon Doyle. Truth Maintenance Systems for Problem Solving. Technical Report TR-419, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, January 1978.

[Doy79] Jon Doyle. A Truth Maintenance System. *Artificial Intelligence*, 12(3):231–272, June 1979.

[Doy83] Jon Doyle. The Ins and Outs of Reason Maintenance. In *Proceedings IJCAI-83*, pages 349–351. AAAI, 1983.

[Ehr78] Hartmut Ehrig. Introduction the the Algebraic Theory of Graph Grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer-Verlag, New York, 1978. Proceedings of an International Workshop.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* Springer-Verlag, New York, 1985. EATCS Monographs on Theoretical Computer Science.

[Fai85] Richard E. Fairley. *Software Engineering Concepts.* McGraw-Hill Book Company, New York, 1985. ISBN 0-07-19902-7.

[Fea79] Martin S. Feather. *A System for Developing Programs by Transformation.* PhD thesis, University of Edinburgh, 1979.

[Fea82] Martin S. Feather. A system for assisting program transformation. *ACM Transactions on Programming Language and Systems*, 4(1):1–20, 1982.

[Fea84] Martin S. Feather. Specification and Transformation: Automated Implementation. In P. Pepper, editor, *Program Transformation and Programming Environments: Report on a Workshop directed by F. L. Bauer and H. Remus*, pages 223–230. Springer-Verlag, New York, 1984.

[Fea86] Martin S. Feather. A Survey and Classification of some Program Transformation Approaches and Techniques. Technical report, Information Sciences Institute, University of Southern California, April 1986. Presented at IFIP WG2.1 Working Conference on Program Specification and Transformation, Bad Toelz, Germany, April 1986.

[Fea89a] Martin S. Feather. Constructing specifications by combining parallel elaborations. *IEEE Transactions on Software Engineering*, 15(2):198–208, February 1989.

[Fea89b] Martin S. Feather. Detecting Interference when Merging Specification Evolutions. In *Proceedings, Fifth International Workshop on Software Specification and Design*, pages 169–176, Pittsburgh, Pennsylvania, May 1989. Published as ACM SIGSOFT Engineering Notes, Volume 14, Number 3, May 1989.

[Fic80] Stephen Fickas. Automatic Goal-directed Program Transformation. In *AAAI-80 Proceedings*, pages 68–70, Palo Alto, California, 1980. AAAI.

[Fic82] Stephen Fickas. *Automating the Transformational Development of Software*. PhD thesis, University of California at Irvine, 1982.

[Fic85] Stephen Fickas. Automating the transformational development of software. *IEEE Transactions on Software Engineering*, SE-11(11):1268–1277, November 1985.

[Fic87] Stephen Fickas. Automating the software specification process. Technical Report 87-05, University of Oregon, Eugene, Oregon, 1987.

[Fik75] Richard E. Fikes. Deductive Retrieval Mechanism for State Description Models. In *Proceedings IJCAI-4*, Tblisi, USSR, September 1975. AAAI.

[For82] Charles L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.

[Fre80] Peter Freeman. Reusable Software Engineering: A Statement of Long-Range Research Objectives. Technical Report UCI-ICS-TR159, Information and Computer Science Department, University of California at Irvine, November 1980.

[Fre87] Peter Freeman. *Software Perspectives: The System is the Message*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1987. ISBN 0-201-11969-2.

[GB78] Cordell Green and David Barstow. On Program Synthesis Knowledge. *Artificial Intelligence*, 10:241–279, 1978.

[Geo87] Michael P. Georgeff. Planning. *Annual Review Computer Science*, 2:359–400, 1987. Reprinted in Readings in Planning, J. Allen, J. Hendler and Austin Tate, eds., 1990, Morgan Kaufmann, San Mateo, California, ISBN 1-55860-130-9.

[GKS86] David Garlan, Charles W. Krueger, and Barbara J. Staudt. A Structural Approach to the Maintenance of Structure-Oriented Environments. In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 160–170, Palo Alto, California, December 1986. ACM.

[GMM+78] M. J. C. Gordon, A. J. R. G. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in LCF. In *Proceedings, 5th ACM POPL Symposium*, pages 119–130, Tucson, Arizona, 1978. ACM.

[GMW79] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.* Springer-Verlag, New York, 1979. Lecture Notes in Computer Science Volume 78.

[Gol84] R. Goldblatt. *Topoi: The Categorial Analysis of Logic.* North-Holland, New York, 1984.

[Gol89] Allen Goldberg. Reusing Software Developments. Technical Report (none), Kestrel Institute, August 1989. 3260 Hillview Avenue, Palo Alto, CA 94304.

[Gov71] Philip Babcock Gove, editor. *Webster's Third New International Dictionary.* G. C. Merriam and Company, Springfield, Massachusetts, USA, 1971.

[Gui83] T. Guimaraes. Managing application program maintenance expenditures. *Communications of the ACM*, 26(10):739–746, October 1983.

[HA87] Michael N. Huhns and Ramon D. Acosta. Argo: An Analogical Reasoning System for Solving Design Problems. Technical Report Technical Report Number AI/CAD-092-87, Microelectronics and Computer Technology Corporation, Austin, Texas, 1987.

[Har86] R. Harper. Introduction to standard ML. Technical Report Report ECS-LFCS-86-14, Laboratory for Foundations of Computer Science, University of Edinburgh, 1986.

[Hec88] R. Heckmann. A Functional Language for the Specification of Complex Tree Transforms. In *Proceedings of European Symposium On Programming '88*, pages –, January 1988. to appear, ref'd by Krieg-Bruckner87a, month is wrong.

[HH88] D. P. Hale and D. A. Haworth. Software Maintenance: A Profile of Past Empirical Research. In *Proceedings of Conference on Software Maintenance 1988*, pages 236–240, Phoenix, Arizona, October 1988. IEEE Computer Society Press. ISBN 0-8186-0879X, IEEE Catalog Number 88CH2615-3.

[HKP87] Annegret Habel, Hans-Jorg Kreowski, and Detlef Plump. Jungle Evaluation. In *Recent Trends in Data Type Specification: 5th Workshop on Specification of Abstract Data Types*, pages 92–112, Gullane Scotland, 1987. Springer-Verlag, New York. Lecture Notes in Computer Science, Volume 332.

[HL78] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[HMM86] R. Harper, D. B. MacQueen, and R. Milner. Standard ML. Technical Report Report ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, University of Edinburgh, 1986.

[HN90] Mehdi T. Harandi and Jim Q. Ning. Knowledge-Based Program Analysis. *IEEE Software*, 7(1), January 1990.

[HPR87] Susan Horwitz, Jan Prinz, and Tom Reps. Integrating non-interfering versions of programs. Technical Report Technical Report #690, University of Wisconsin, March 1987.

[HPR88] Susan Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 133–145, San Diego, California, January 1988. ACM, New York, 1988.

[HR88] Susan Horwitz and Thomas Reps. ??sufficient slices are sufficient to distinguish different computations. In *1988 SIGPLAN POPL*, 1988.

[Imp86] Imperial Software. A Development Environment for Functional Languages. Technical Report IST Project No. 7003, Technical Report Tr.6, Imperial Software Technology, England, November 1986.

[JF90] W. Lewis Johnson and Martin Feather. Building An Evolution Transformation Library. In *Proceedings of the 12th International Conference on Software Engineering.* IEEE Computer Society Press, March 1990.

[Joh80] S. C. Johnson. Language Development Tools on the Unix System. *IEEE Computer*, 13(8), August 1980.

[Joh86] W. Lewis Johnson. *Intention-based Diagnosis of Novice Programming Errors.* Morgan Kaufmann, Palo Alto, California, 1986.

[Kam89] Subbarao Kambhampati. *Flexible Reuse and Modification in Hierarchical Planning: A Validation Structure Based Approach.* PhD thesis, University of Maryland, October 1989. Tech report numbers CAR-TR-469, CS-TR-2334, Computer Vision Laboratory, Center for Automation Research, College Park Maryland, 20742-3411.

[Kan79] Elaine Kant. *Efficiency considerations in program synthesis: A knowledge-based approach.* PhD thesis, Computer Science Department, Stanford University, 1979.

[Kan81] Elaine Kant. *Efficiency in Program Synthesis.* UMI Research Press, Ann Arbor, Michigan, 1981.

[KB81] Elaine Kant and David R. Barstow. The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Program Synthesis. *IEEE Transactions on Software Engineering*, September 1981. Reprinted in: New Paradigms for Software Development, W. Agresti, editor, IEEE Press, 1986, ISBN 0-8186-0707-6, pp. 257-270.

[KB88] Bernd Krieg-Bruckner. The PROSPECTRA Methodology of Program Development. In *IFIP/IFAC Working Conference on Hardware and Software for Real Time Process Control*, pages 1–15, Warsaw, Poland, 1988. North-Holland, New York.

[KB89a] B. Krieg-Bruckner. ESPRIT Project Report #390: Algebraic Specification with Functionals in Program Development by Transformation. In *Esprit 89: Proceedings of the 6th Annual Esprit Conference.* Luwer Academic Publishers, Dordrecht, The Netherlands, November 1989. Edited by Commission of the European Communities, Directorate-General Telecommunications, Information Industries and Innovation.

[KB89b] Bernd Krieg-Bruckner. Algebraic Specification and Functionals for Transformational Program and Meta Program Development. In *TAPSOFT '89: Proceedings of the International Joint Conference on Theory and Practice of Software Development*, pages 36–59, Barcelona, Spain, March 1989. Springer-Verlag. LNCS volume 352.

[Kib78] Dennis F. Kibler. *Power, Efficiency, and Correctness of Transformation Systems.* PhD thesis, University of California at Irvine, Irvine California, 1978.

[Kil73] Gary Kildall. A Unified Approach to Global Program Optimization. In *Conference Record of ACM Symposium on Principles of Programming Languages*, pages 194–206, Boston, Massachusetts, October 1973. ACM.

[Kor85] Richard E. Korf. Iterative Deepening A*: An Optimal Admissible Tree Search. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Vol. 2*, pages 1033–1036, Los Angeles, August 1985.

[Kor87] Richard E. Korf. Planning as Search: A Quantitative Approach. *Artificial Intelligence*, 33:65–88, 1987. Reprinted in Readings in Planning, pages 566-577 Allen, Hendler, Tate eds., 1990, Morgan Kaufmann, Inc.

[LD89] Michael Lowry and Raul Duran. Chapter XX: Knowledge-based Software Engineering. In *The Handbook of Artificial Intelligence, Volume 4*, pages 242–322. Addison-Wesley, Reading, Massachusetts, 1989.

[Lei87] J. C. S. P. Leite. Requirements techniques and languages. Technical Report RTP-090, Information and Computer Sciences Department, University of California at Irvine, 1987.

[Lei88] J. C. S. P. Leite. *Viewpoint Resolution in Requirements Elicitation.* PhD thesis, University of California at Irvine, 1988.

[LF82] Philip E. London and Martin S. Feather. Implementing Specification Freedoms. *Science of Computer Programming*, 2:91–131, 1982.

[Lif86] Vladimir Lifschitz. On the semantics of STRIPS. In Michael P. Georgeff and Amy L. Lansky, editors, *Reasoning about Actions and Plans*, pages 1–10. Morgan Kaufmann Publishers, Inc, Los Altos, California, 1986. Reprinted in Readings in Planning, pages 523-530, Allen, Hendler, Tate eds., 1990, Morgan Kaufmann, Inc.

[Lon78] P. London. A dependency-based modelling mechanism for problem solving. In *AFIPS Conference Proceedings*, pages 263–274. AFIPS, 1978. Volume 47.

[LQ89] Mark A. Linton and Russel W. Quong. A Macroscopic Profile of Program Compilation and Linking. *IEEE Transactions on Software Engineering*, 15(4):427–436, 1989.

[LS80] B. P. Lientz and E. B. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations.* Addison-Wesley, Menlo Park, 1980.

[LS86] Stanley Letovsky and Elliot Soloway. Delocalized Plans and Program Comprehension. *IEEE Software*, 3(3):41–49, May 1986.

[Lub89] Mitchell D. Lubars. Representing Design Dependencies in the Issue-Based Information System Style. Technical Report STP-426-89, Microelectronics and Computer Technology Corporation, Austin, Texas, November 1989.

[LvHKB87] D. C. Luckham, F. W. von Henke, and B. Krieg-Bruckner. *Anna, a Language for annotating Ada Programs, Reference Manual.* Springer-Verlag, 1987. Lecture Notes in Computer Science 260.

[Mar86] Peter Marks. What is Leonardo? Technical Report MCC Technical Report STP-141-86, Microelectronics and Computer Technology Corporation, Austin, Texas, May 1986.

[MB87] Jack Mostow and Mike Barley. Automated Reuse of Design Plans. Technical Report Working Paper Number 53, Rutgers AI/Design Project, February 1987. submitted to International Conference on Engineering Design, Boston, MA, August 1987.

[McC87] Robert D. McCartney. Synthesizing Algorithms with Performance Constraints. In *Proceedings 6th National Conference on Artificial Intelligence*, pages 149–154, Seattle, Washington, July 1987. AAAI.

[McC88] Robert McCartney. *Synthesizing algorithms with performance constraints.* PhD thesis, Brown University, 1988. Brown University Department of Computer Science Technical Report No. CS-87-28, December, 1987.

[McD82] Drew McDermott. DUCK: A Lisp-based Deductive System. Technical report, Department of Computer Science, Yale University, 1982.

[McD83] D. McDermott. Contexts and Data Dependencies: A Synthesis. *IEEE Pattern Analysis and Machine Intelligence*, 5(3):239–246, 1983. Earlier version available from Yale University.

[MD80] Drew McDermott and Jon Doyle. Non-Monotonic Logic I. *Artificial Intelligence*, 13(2):41–72, April 1980.

[MDG86] Ali Milli, Jules Desharnais, and Jean Raymond Gagne. Formal models of stepwise refinement. *ACM Computing Surveys*, 18(3):231–276, September 1986.

[MF89a] Jack Mostow and Greg Fisher. Replaying Transformational Derivations of Hueristic Search Algorithms in DIOGENES. Technical Report Rutgers AI/Design Project Working Paper Number 113-1, Rutgers University, Department of Computer Science, AI/VLSI Project, 1989. Available in Proceedings of the AAAI 1989 Spring Symposium on AI and Software Engineering, Palo Alto, CA March 1989.

[MF89b] Jack Mostow and Greg Fisher. Replaying Transformational Derivations of Hueristic Search Algorithms in DIOGENES. In *Proceedings of the DARPA Workshop on Case-Based Reasoning*, pages 94–99, Holiday Inn, Pensacola Beach, Florida, May 1989.

[MM88] David A. Marca and Clement L. McGowan. *SADT: Structured Analysis and Design Technique.* McGraw-Hill, New York, 1988.

[Mos85a] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, Cambridge, Massachusetts, 1985. ISBN 0-262-13200-1Q.

[Mos85b] J. Mostow. Towards better models of the design process. *AI Magazine*, 6(1):44–56, 1985.

[Mos85c] Jack Mostow. Some Requirements for Effective Replay of Derivations. In *Proceedings of 3rd International Machine Learning Workshop*, pages 129–132, Skytop, Pennsylvania, June 1985.

[Mos86] Jack Mostow. Why are design derivations hard to replay? In T. Mitchell, J. Carbonell, and R. Michalski, editors, *Machine Learning: A Guide to Current Research*, Hingham, Massachusetts, 1986. Kluwer. Revised and condensed version of paper in **Proceedings of the 3rd International Machine Learning Workshop**.

[MS86] Joao P. Martins and Stuart C. Shapiro. Theoretical Foundations for Belief Revision. In Joseph Y. Halpern, editor, *Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning About Knowledge*, pages 383–398. Morgan-Kaufmann, March 1986. ISBN 0-934613-04-4.

[MSNT88] A. Maggiolo-Schettini, M. Napoli, and G. Tortora. Web structures: A tool for representing and manipulating programs. *IEEE Transactions on Software Engineering*, 14(11), November 1988.

[Nag78] Manfred Nagl. A Tutorial and Bibliographical Survey on Graph Grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer-Verlag, New York, 1978. Proceedings of an International Workshop.

[Nei80] James Milne Neighbors. *Software Construction Using Components.* PhD thesis, University of California at Irvine, 1980. Available as ICS Tech Report 160.

[Nei84a] James Neighbors. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering*, SE-10(5):–, September 1984.

[Nei84b] James M. Neighbors. Draco 1.3 Users Manual. Technical Report Technical Report 230, Information and Computer Sciences Department, University of California at Irvine, October 1984.

[Nei89] James M. Neighbors. Draco: A Method for Engineering Reusable Software Systems. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability*, pages 295–320. ACM Press, New York, 1989. ISBN 0-201-08017-6.

[Nin89] Jim Qun Ning. *A Knowledge-Based Approach to Automatic Program Analysis.* PhD thesis, University of Illinois at Urbana-Champaign, 1989. UIUCDCS-R-89-1548.

[NSM85] R. Neches, W. R. Swartout, and J. D. Moore. Enhanced Maintenance and Explanation of Expert Systems Through Explicit Models of Their Development. *IEEE Transactions on Software Engineering*, SE-11(11):1337–1350, 1985.

[Our89] Dirk Ourston. Program Recognition. *IEEE Expert*, 4(4):36–49, 1989.

[Pag81] Frank G. Pagan. *Formal Specification of Programming Languages: A Panoramic Primer.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1981. ISBN 0-13-329052-2.

[Pai81] Robert A. Paige. *Formal Differentiation: A Program Synthesis Technique.* University Microfilms International, Ann Arbor, Michigan, 1981.

[Pai86] Robert Paige. Programming with Invariants. *IEEE Software*, 3(1):56–69, January 1986.

[Par72] David Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Par76] David L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2, March 1976. verify.

[Pau87] Lawrence C. Paulson. *Logic and computation: Interactive proof with Cambridge LCF.* Cambridge University Press, Cambridge, England, 1987. ISBN 0-521-34632-0.

[PC86] David Lorge Parnas and Paul C. Clements. A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering*, SE-12(2):251–257, February 1986.

[Pea84] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1984. ISBN 0-201-05594-5.

[Pet87] Charles J. Petrie, Jr. Revised Dependency-Directed Backtracking for Default Reasoning. In *Proceedings, AAAI 87 Sixth National Conference on Artificial Intelligence*, pages 167–172, Los Altos, Calif, July 1987. Morgan-Kaufmann Publishers.

[PGLS88] N. Prywes, X. Ge, I. Lee, and M. Song. Reverse Software Engineering. Technical Report MS-CIS-88-99, Computer and Information Science Department, University of Pennsylvania, 1988.

[Pid90] Christopher W. Pidgeon. *Analyzing Decision Making in Software Design.* PhD thesis, University of California at Irvine, February 1990. UCI Tech Report 90-16.

[Pos43] Emil L. Post. Formal reductions of the general combinatorial problem. *American Journal of Mathematics*, 65:197–268, 1943.

[PP89] Francesco Parisi-Presicce. Modular System Design Applying Graph Grammars Techniques. In *Automata, Languages and Programming*, pages 621–636, Stresa, Italy, 1989. Springer-Verlag, New York. Lecture Notes in Computer Science Volume 372.

[Pre87] Roger Pressman. *Software Engineering, A Practicioners Approach.* McGraw-Hill, New York, 1987.

[PS83] H. Partsch and R. Steinbruggen. Program Transformation Systems. *Computing Surveys*, 15(3):199–236, March 1983. Reprinted in **New Paradigms for Software Development**, William W. Agresti, ed., IEEE, 1986, ISBN 0-8186-0707-6.

[Pyl87] Z. Pylyshyn, editor. *The Robot's Dilemma.* Ablex Publishing, Norwood, New Jersey, 1987. ISBN 0-89391-371-5.

[RD88] A. Ricketts and J. C. Delmonaco. Software Re-engineering with Retrofit. In *Computer Programming Management.* Auerbach Publishers, 1988.

[Rea86] Reasoning Systems Incorporated. *REFINE User's Guide.* Reasoning Systems, Inc., Palo Alto, 1986.

[Rep84] Thomas W. Reps. *Generating Language-Based Environments.* PhD thesis, Cornell University, 1984. Available from MIT Press, 1984. ISBN 0-262-18115-0.

[ROL90] Spencer Rugaber, Stephen B. Ornburn, and Richard J. LeBlanc, Jr. Recognizing Design Decisions in Programs. *IEEE Software*, 7(1), January 1990.

[Ros77] Douglas Ross. Structured Analysis (SA): A Language for Communicating Ideas. *Transactions on Software Engineering*, SE-3(1), January 1977.

[RPTU84] C. V. Ramamoorthy, A. Prakash, W. Tsai, and Y. Usuda. Software Engineering: Problems and Perspectives. *IEEE Computer*, October 1984.

[Rus85] David M. Russinoff. An Algorithm for Truth Maintenance. Technical Report AI-062-85, Microelectronics and Computer Technology Corporation, April 1985.

[RW88] Charles Rich and Richard C. Waters. The programmer's apprentice project: A research overview. *IEEE Computer*, 21(11), November 1988. Also available from MIT AI Laboratory, Massachusetts Institute of Technology.

[RW90] Charles Rich and Linda M. Wills. Recognizing a Program's Design: A Graph-Parsing Approach. *IEEE Software*, 7(1), January 1990.

[SA89] D. M. Steier and A. P. Anderson. *Algorithm Synthesis: A Comparative Study.* Springer-Verlag, New York, 1989. ISBN 0-387-96960-8.

[Sac74] E. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974. Reprinted in Readings in Planning, J. Allen, J. Hendler and Austin Tate, eds., 1990, Morgan Kaufmann, San Mateo, California, ISBN 1-55860-130-9.

[Sac77] E. Sacerdoti. *A Structure for Plans and Behavior.* Elsevier North-Holland, New York, 1977. ISBN 0-444-00209-X.

[Sca86] Walt Scacchi. Gist: An Operational Knowledge Specification Language. Technical Report Draft Technical Report, University of Southern California, Information Sciences Institute Institute, April 1986.

[Sch87] N. F. Schneidewind. The State of Software Maintenance. *IEEE Transactions on Software Engineering*, SE-13(3):303–310, 1987.

[Sch89] James G. Schmolze. Guaranteeing serializable results in synchronous parallel production systems. Technical Report Technical Report 89-5, Department of Computer Science, Tufts University, October 1989.

[Sha83] Ehud Y. Shapiro. *Algorithmic Debugging.* MIT Press, Boston, Massachusetts, 1983.

[SHFN76] T. A. Standish, D. C. Harriman, D. F.Kibler, and J. M. Neighbors. The Irvine Program Transformation Catalogue. Technical Report Technical Report, Department of Information and Computer Science, University of California at Irvine, January 1976.

[Sil86] Bernard Silver. *Meta-level Inference*. Elsevier Science Publishers, New York, N.Y., 1986. ISBN 0-444-87900-5.

[Sin83] M. Sintzoff. Understanding and Expressing Software Construction. In P. Pepper, editor, *Program Transformation and Programming Environments*, pages –. Springer-Verlag, 1983. ISBN 3-540-12932-4.

[SJ85] E. Soloway and W. L. Johnson. PROUST: Knowledge-Based Program Understanding. *IEEE Transactions on Software Engineering*, SE-11(3):267–275, March 1985.

[SJ88] Harry M. Sneed and Gabor Jandrasics. Inverse Transformation of Software from Code to Specification. In *Proceedings- Conference on Software Maintenance 1988*, pages 102–109, Phoenix, Arizona, October 1988. ISBN 0-8186-0879X, IEEE Catalog Number 88CH2615-3.

[SKW85] D. R. Smith, G. B. Kotik, and S. J. Westfold. Research on Knowledge-Based Software Environments at Kestrel Institute. *IEEE Transactions on Software Engineering*, SE-11(11):1278–1295, November 1985.

[SM84] Louis I. Steinberg and Tom M. Mitchell. A Knowledge Based Approach to VLSI CAD: The Redesign System. In *Proceedings 21st Design Automation Conference*, pages 412–418. IEEE, 1984.

[SM85] L. I. Steinberg and T. M. Mitchell. The Redesign System: a knowledge-based approach to VLSI CAD. *IEEE Design and Test*, vol???:45–54, February 1985.

[Smi85] Douglas R. Smith. Top-Down Synthesis of Divide-and-Conquer Algorithms. *Artificial Intelligence*, 27:43–96, 1985.

[Smi89] Douglas R. Smith. KIDS: A Semi-Automatic Program Development System. Technical report, Kestrel Institute, Palo Alto, California 94304, October 1989. To appear, Special Issue on Formal Methods, IEEE Transactions on Software Engineering.

[Sne89] Harry M. Sneed. The Myth of Top-Down Software Development and its Consequences for Software Maintenance. In *Proceedings of Conference on Software Maintenance 1989*, pages 22–29, Miami, Florida, October 1989. IEEE Computer Society Press. ISBN 0-8186-1965-1, IEEE Catalog Number 89CH2744-1.

[Sol87] Elliot Soloway. I Can't Tell What in the Code Implements What in the Specs, 1987. Talk.

[Sow84] John F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine.* Addison-Wesley, 1984. ISBN 0-201-14472-7.

[Sri91] Yellamraju V. Srinivas. *Pattern Matching: A Sheaf-Theoretic Approach.* PhD thesis, University of California at Irvine, 1991. Forthcoming.

[ST88] Donald Sanella and Andrzej Tarlecki. Toward Formal Development of Programs from Algebraic Specifications: Implementations Revisited. *Acta Informatica*, 23:233–281, 1988.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Swa82] W. Swartout. On the Inevitable Intertwining of specification and implementation. *Communications of the ACM*, 25(7):438–440, July 1982.

[Tat77] A. Tate. Generating Project Networks. In *Proceedings IJCAI-77*, pages 888–893, Cambridge, Massachusetts, 1977. AAAI. Reprinted in Readings in Planning, J. Allen, J. Hendler and Austin Tate, eds., 1990, Morgan Kaufmann, San Mateo, California, ISBN 1-55860-130-9.

[TM87] Wladyslaw M. Turski and Thomas S. E. Maibaum. *The Specification of Computer Programs.* Addison-Wesley, New York, 1987.

[vdB81] Peter van den Bosch. *The Translation of Programming Languages through the use of a Graph Transformation Language.* PhD thesis, Department of Computer Science, University of British Columbia, Vancouver B. C., Canada, 1981.

[Wal77] R. Waldinger. Achieving Several Goals Simultaneously. In E. Elcock and D. Michie, editors, *Machine Intelligence 8*, pages 94–136. Ellis Horwood, Chichester, Great Britain, 1977.

[Wat88] R. C. Waters. Program Translation via Abstraction and Reimplementation. *IEEE Transactions on Software Engineering*, 14(8):1207–1228, August 1988.

[WCM89] M. Ward, F. W. Calliss, and M. Munro. The Maintainer's Assistant. In *Proceedings of Conference on Software Maintenance 1989*, pages 307–315, Miami, Florida, October 1989. IEEE Computer Society Press. ISBN 0-8186-1965-1, IEEE Catalog Number 89CH27441-1.

[Wed85] John D. Wedo. Structured Program Analysis Applied to Software Maintenance. In *Proceedings of Conference on Software Maintenance-1985*, pages 28–34, Washington DC, 1985. IEEE. ISBN 0-8186-0648-7.

[WHR78] D. A. Waterman and F. Hayes-Roth. An Overview of Pattern-Directed Inference Systems. In D. A. Waterman and F. Hayes-Roth, editors, *Pattern-Directed Inference Systems*, pages 3–22. Academic Press, New York, 1978.

[Wil83] D. Wile. Program Developments: Formal Explanations of Implementations. *Communications of the ACM*, 26(11):902–911, November 1983. Also available from University of Southern California, Information Sciences Institutes as report ISI/RR-81-99, which includes the appendices mentioned by, but frustratingly missing from, the ACM version.

[Wil86] David S. Wile. Local Formalisms: Widening the Spectrum of Wide Spectrum Languages. In L. G. L. T. Meertens, editor, *Proceedings of IFIP WG2.1 Working Conference on Programme Specifications and Transformations*, pages –, Bad-Tolz, West Germany, April 1986.

[Wil87] Linda M. Wills. Automated Program Recognition. Master's thesis, Massachusetts Institute of Technology, 1987.

[Wil88] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann Publishers, Inc., Los Altos, California, 1988. ISBN 0-934613-94-X.

[WML$^+$89] Chris Wild, Kurt Maly, Lianfang Liu, Jann-Shinn Chen, and Ting Xu. Decision-Based Software Development: Design and Maintenance. In *Proceedings of Conference on Software Maintenance 1989*, pages 297–306, Miami, Florida, October 1989. IEEE Computer Society Press. ISBN 0-8186-1965-1, IEEE Catalog Number 89CH2744-1.

[YNT86] Stephen S. Yau, Robin A. Nicholl, and Jeffrey J-P Tsai. An Evolution Model for Software Maintenance. In *Proceedings, COMPSAC-86*, pages 440–446. IEEE, October 1986.

[YNTL88] S. S. Yau, R. A. Nicholl, J. J.-P. Tsai, and S.-S. Liu. An Integrated Life-Cycle Model for Software Maintenance. *IEEE Transactions on Software Engineering*, 14(14):1128–1144, August 1988.