# USING DESIGN HISTORY

Baykal Cakici, Imranul Islam

## Outline

- A Process for Consolidating and Reusing Design Knowledge
- Software Change Through Design Maintenance

## A Process for Consolidating and Reusing Design Knowledge

Guillermo Arango, Eric Schoen, and Robert Pettengill

**Presenter:** Imranul Islam

## Overview

- Motivation
- Applications
- Strategy
- Books
- Contents of Books
- Evolution of Books
- Technology Book Structures
- Example 1
- Composition of Design Rationale
- Example 2
- Tool Environment
- Conclusions

## Why Consolidate Knowledge?

- Design evolution and maintenance is the dominant activity.
- Industry data suggests that design evolution accounts form 70% to 90% of the cost of a software system.
- The lack of understanding of the existing implementations account for most of the risks involved in software development.
- The authors observed that one to two thirds of the evolution costs can be traced back to designer's and maintainers lack of understanding of the consequences of incremental change.

## Applicability

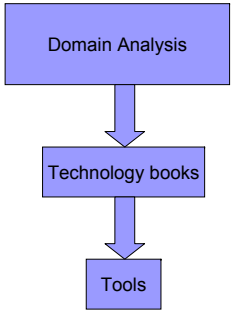
The projects in which consolidation may be used:

- Systems inherently difficult to understand
- Long-lived systems
- Systems customized for different applications
- Products with strict organizational constraints

The Projects of this paper:

- Design cycles of 6 to 12 months
- The designs are highly constrained
- Embedded software
- Lifespan of 20 years
- High cost of maintenance

## Strategy




## Strategy (Cont.)

- Domain Analysis
  - Techniques to consolidate critical analysis and design decisions for product families; the knowledge is not really product specific.
- Technology Books
  - Representation of reusable information in structured form.
- Tools
  - Devices for facilitating information reuse.
- The authors benefited from the strategy with minimal computer support.
- They observed that reuse of analysis and designs is more useful than reuse of software.

## Technology Books

- They are analogous to engineering handbooks.
- They consolidate the best knowledge.
- Their information is narrowly defined.
- Solutions to a complex design problem draw from different domains.
- Technology books support domains specific to a certain generation of a product family.
- An embedded system of 32K lines of code may produce 10 technology books.
- These are object oriented databases
- Relations with well-defined semantics are present
  - They make the book navigable so that the information it contains can be reasoned about.

## Product Books

- They consolidate knowledge specific to individual system instances.
- They include specialized versions of analyses drawn from technology books.
- They also include histories of deliberations.

## The process of consolidation

- Define a language for the problem description.
- Produce formal models of the solutions to the problem.
- Demonstrate that the formal model explains our system.
- Identify good designs that map solutions to the selected implemented technology.
- Explicitly specify issues, assumptions, constraints, and dependencies in the designs.
- Explicitly show how the reusable software modules relate to the designs.

Technology books collect and organize the result of this process.

## Contents of a Technology Book

Technology books contain:

- Analytical models of classes of solutions.
- Language definitions: more than the usual definition of terms.
- Computational design models for these analyses.
- Implementation of these designs.
- Explanations
  - Justify the implementations in terms of designs, and the designs in terms of the underlying analytical model.
  - As formal as mathematical proofs.

## Technology Book Structures

- The books use a number of tags.
  - The tags represent a careful compromise between usability and formality.
  - Semantic tags identify, for example, issues, definitions, and design decisions.
  - Syntactic tags identify, for example, authors, equations, and enumerations.
- Information is stored in typed nodes and in relation between nodes.
  - A typical node may include encapsulated documents or other information fragments.
- The information nodes are organized into taxonomies according to their types.
  - The major hierarchies include project entities, work product, resources, and analyses.
- The relationships are organized into a taxonomy based on their semantics and their use in building blocks.
  - The general categories of relations include: history, derivation, use, justification, and ownership.

## Why Recording Analytical Models is important?

- Recovering a mathematical model governing a DSP task from 12K lines of assembly code took two scientists months.
- Sometimes synthesizing models may become as difficult as proving theorems in physics and mathematics.
  - This high cost of recovering critical knowledge motivates formalizing it.

## Evolution of the Communication Medium

- Informal documentation
  - Documentation is paper based.
  - First technology books were actual books.
  - The strength of this approach not based on technology, but the methodology.
- Structured Documentation
  - Semiformal approach using template documents and commercial document preparation systems.

## Structured Documentation

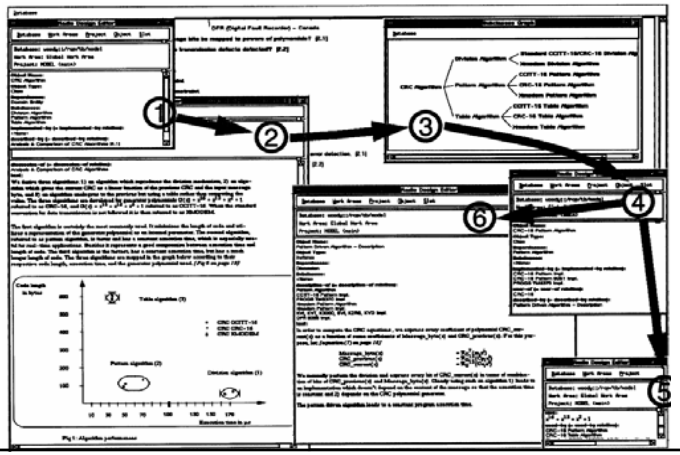
- The engineers stored information in tagged documents for easy identification and reuse.
- Typical tags include: requirement, analysis, issue, and software module.
- Encoded dependencies between related paragraphs using cross-references.
- Paragraphs were also tagged to communicate type of information such as OC for output constraints, and AS for assumptions.
- Proved to be a powerful educational and technology transfer mechanism.

## On-line Repository

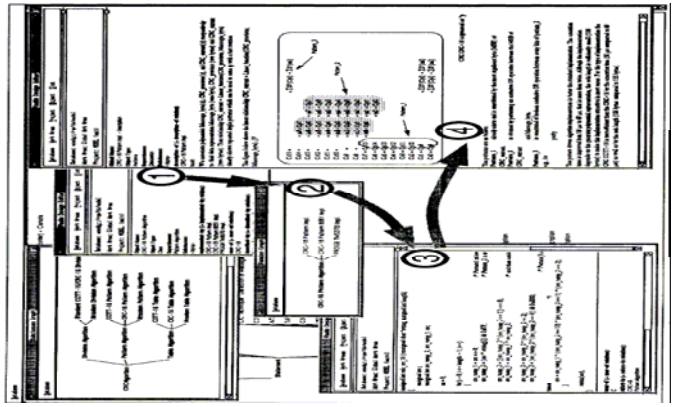
- The repository is an ObjectStore database.
- The paragraphs from the structured documentations became objects for the repository.
- There were tools to parse the documentation to extract information.
- Interactive tools navigated relationships which included classifications, and IBIS like deliberation networks.

Final stage of the evolution is the suite of RADIO tools which is discussed later.

## Example 1

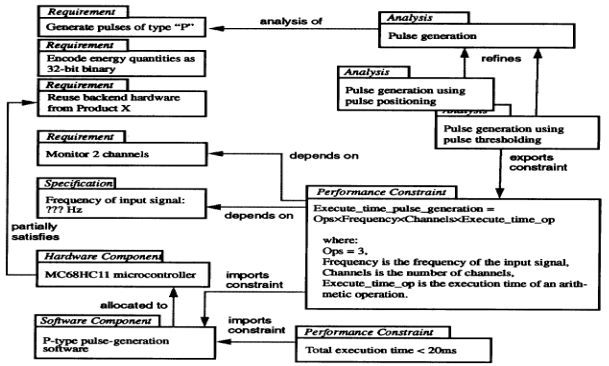


## Example 1(Cont.)



## Composition of Design Rationales

- How do the changes in data representation from floating point binary affect system performance?



## Design Rationales

- Rationales that are indirect:
  - Design history
  - Trace of deliberations and negotiations
  - Traces of work product developments
- Formal Rationales
  - Basic principles, assumptions, and analytical derivations.

## Tool Environment

- RADIO
  - An environment to access technology books.
- Consists of
  - An object oriented database
  - User interface
  - A number of auxiliary support tools
- 75K lines of C++ custom code

## The Master Repository

- ObjectStore commercial object oriented database
- Data can be in arbitrarily complex forms
- Accessed in a client-server architecture
- ObjectStore provides transactions and concurrency control
- ObjectStore's version control facility is used

## Modeling Language

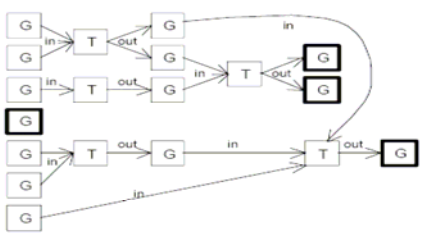
- Books contain semiformal information.
- The formal portion structures knowledge.
- The structured information is stored in objects defined in DOLL.
- DOLL allows designers to create and modify objects interactively.
- DOLL can also generate files from the technology books.
- Informal information contain text, pictures, tables, and equations.
  - Paragraphs attached to DOLL objects.

## User Interface and Architecture

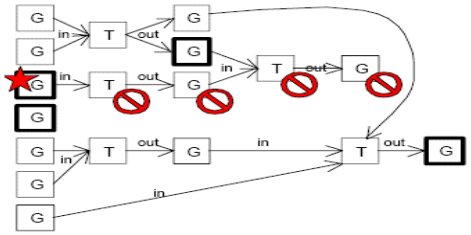
- Helps the user navigate the technology book.
- Helps the user add and extract information to and from the book.
- Designed in Motif.
- The interface is Intelligent.
- Includes a mail system
- The RADIO architecture is open.
- Portable Common Tool Environment (PCTE) or Common Object Request Broker Architecture (CORBA) will eventually be a common substrate.

## Further Analysis

- Why is this system more effective than a version management system?




## Updating the History Graph




## Conclusions

- Technology books are electronic design notebooks.
- However, they are domain specific not product specific.
- They are in effect a history graph.

## Conclusions (Cont.)

- The process has been applied to embedded software development.
- Engineering groups have applied the process for actual product development.
- Designers have found knowledge consolidation to be intellectually challenging.
- Reuse of consolidated knowledge has discovered at least one product flaw.
- The engineering groups have reported a reduction of two-thirds the number of design iterations.
- Technology books act as an effective means of communication between marketing personnel and design engineers.

## Further Information


1. T. A. Standish. An essay on software reuse. IEEE Transactions on Software Engineering.
2. Jahnke, Wadsack, Zundrof.A History Concept for Design Recovery Tools. IEEE Computer Society Press.
3. R. Prieto-Diaz and G. Arango, editors, Domain Analysis and Software Systems Modeling. IEEE Computer Society Press.
4. G. Arango and E. Schoen. Using product models to compose rationales. In AAAI-92 Workshop on Design Rationale Capture and Use.


# Software Change Through Design Maintenance


Ira D. Baxter, Christopher W. Pidgeon

**Presenter:** Baykal Cakici


## Overview

- Conventional software engineering focuses on a small part of the software life cycle
  - the design and implementation of a product.
- The bulk of the lifetime cost is in the maintenance phase
  - Very little theory and fewer tools to manage the maintenance activity.
  - A fundamental cause of the difficulty is the failure to preserve design information.

## Overview

- An alternative paradigm:
  - make the design the central focus of the *construction* process
    - get code as a byproduct;
  - make the design the central focus of the *maintenance* process
    - preserve revised designs and get code as a byproduct.
- A transformational scheme for accomplishing this, called ***Design Maintenance System***, is presented

## Outline

- Introduction
- What's in a Design?
- Transformation Systems
- Capturing a Transformational Design
- Design Modification
- An Example
- Reverse Engineering
- Supporting Technology
- Implementation
- Conclusion

## Introduction

- The average lifetime of software is about 10 years.
- Most of the lifecycle costs for software occur during the so-called *maintenance phase.*
- Incomplete or nonexistent system documentation was ranked in the top four problems
- Two major obstacles:
  - understanding the program to be modified,
  - validating the modification while assuring that the remainder of the program is not accidentally affected.

## Introduction

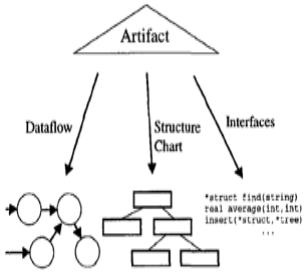
- **Observation:** Better processes for producing and maintaining system documentation for generated programs would reduce maintenance costs.
- **Recommendation:** software development process should treat the design as the **major product**, with the implementation (code) being merely a useful **byproduct**.

## Introduction

- Informal designs are subject to wide interpretation.
  - Variability limits their value.
  - The burden of details to be managed makes this very hard.
- Formal development methodology: ***Design Maintenance System (DMS)***
  - Transformationally constructs and records the design of software.
  - Design may be incrementally modified by the DMS to produce revised versions of the software.

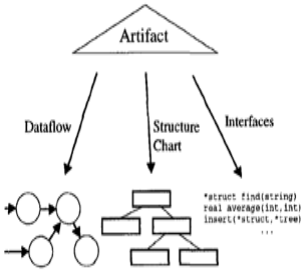
## What's in a Design?

- Most design notations can be considered as projections of the completed artifact, under which some chosen aspect of the artifact is displayed.
  - call graphs (structure charts), data flow diagrams, state machines, interface specifications, etc.




## What's in a Design?

- The design process consists of:
  - choosing sets of projections that are believed to be able to construct the final artifact
  - acquiring construction hints from those projections.
- Questions about the artifact are answered by inspecting the projections.




## What's in a Design?

- The flaw with this notion of design is the ***absence of rationale;***
  - projections do not explain ***why*** the artifact organized the way that it is.
- Without such rationale, one can hardly hope to explain the artifact.
- One way to capture a rationale is to understand how the artifact was constructed, and why the construction works.

## What's in a Design?

- **DMS** provides this information. It allows to capture the design rationale as:
  - A specification of the desired task (both functionality and performance)
  - A derivation of the implementation from the specification that explains the final program
  - A justification of the derivation steps.

## Transformation Systems

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

## Transformation Systems

- Transforms are functions from specifications to specifications (***t:S→S) .***
- Many transforms are actually optimizations, such as the ***eliminate-additive-identity*** transform:

  ***x+0→x***

## Transformation Systems

- When a ***transform*** is applied at a particular location in a specification, we obtain a ***transformation*** of the specification.
- The italicized names are parameters of the transform, and are consistently substituted where the transform is used.

## Transformation Systems

- The place where the transform has been applied is called the ***locator.***
  - The value of the locator depends on the underlying representation of the specification, **e.g.,** a path for a tree.
- Example for paragraph shaped specifications:
  **@ *line number:token number***

## Transformation Systems

- ***Example****:*
  *eliminate-additive-identity @3:1*

```
1 do j= 1 to10
2    s=s+0
3    p=p+0
4 end
```

- ***Solution:*** change the specification by binding x to **p** and rewriting **p+0** as **p.**

```
1 do j= 1 to10
2    s=s+0
3    p=p
4 end
```

## Transformation Systems

- Full specification has two conceptual parts:
  - **Functional** specification (what the desired program should do),
  - **Performance** specification (how well it should do it).
- Functional specifications may be written as abstract programs, as input-output constraints, or in problem domain specific notations.
- Performance specifications are often stated in terms of desired target languages, speed, complexity.

## Transformation Systems

- A program specification may be very abstract or describe a very complex system,
  - where a large number of transformations may need to be applied to implement the specification at the desired level of performance.
- **Metaprograms** are used to control the selection and application of the transforms since manual application of large numbers of transformations is impractical
  - If a metaprogram cannot decide locally what to do, then it may backtrack to try alternatives.

## Capturing a Transformational Design

- Theoretically, a specification of the artifact **is** sufficient to provide a rationale.
  - It is possible to work forward from the specification to rediscover the purpose of each part of the artifact.
- However, we **DO NOT** want to effectively redesign it each time we need an explanation.

## Capturing a Transformational Design

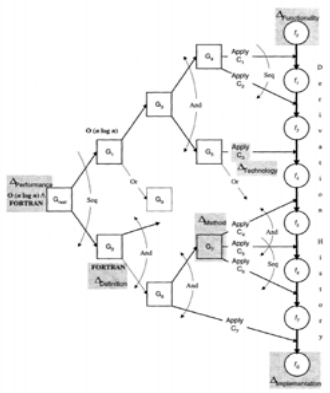
- The assumption is that either the programmer or the transformation system worked hard:
  - to discover which transforms to apply,
  - to determine exactly where to apply them,
  - to achieve the desired level of performance.
- To explain a transformationally derived program, we only need:
  - the derivation history,
  - the sequence of transformations applied to the functional specification

## Capturing a Transformational Design

- The choice of the individual transformations is explained by the effect the individual transformation has on achieving the performance specification.
- So, if we record how the overall performance specification is broken into subspecifications over smaller locales, we obtain a ***design history.***
  - This includes a **derivation history** - the complete explanation of how the performance and functional specifications are met.

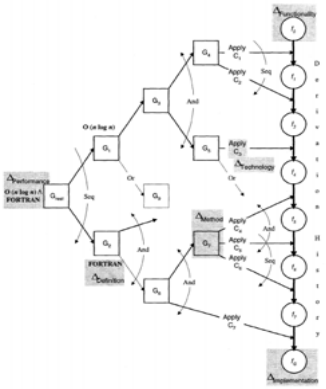
## Capturing a Transformational Design

- **Example:** an abstract design history.
- The initial functional specification, $f_o$, was transformed by application of transformations $c_1$, $c_2$, etc. until the final implementation, $f_G$ was obtained.
- The performance specification, $G_{rest}$ was recursively partitioned by choosing methods that achieve individual performance levels;



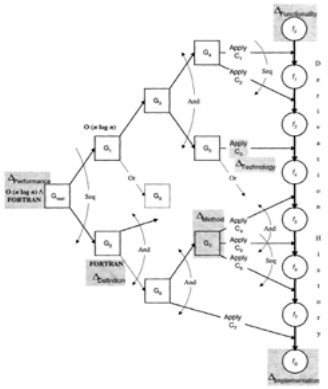
## Capturing a Transformational Design

- At the end, low-level methods apply particular transformations.
- Such a design history provides a complete explanation of the final artifact, $f_G$.



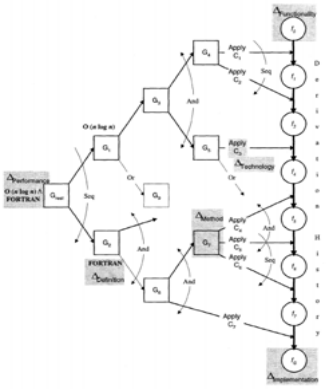
## Design Modification

- It is possible to incrementally revise the design history to produce a new one.
- Gray boxes represent the various kinds of changes $\Delta_{type}$ that can affect the final artifact.
  - Some changes affect the functionality
  - Some changes affect the performance



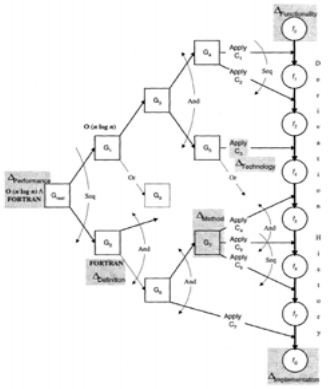
## Design Modification

- Each change can cause complex ripples in the structure of the design history.
- The key to revising the derivation history is to take advantage of the ability to change the order in which the transformations were originally applied.



## Design Modification

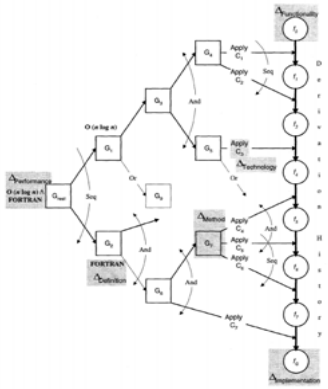
- Essentially, we wish to;
  - Preserve transformations when possible
  - Remove transformations that are no longer useful
- Start the process with a functionality delta applicable to the initial specification.



## Design Modification

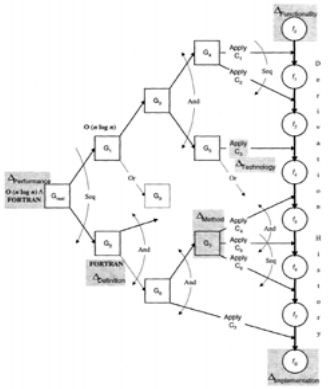
- For each intermediate specification (including the initial spec), there is a transformation leading to the next intermediate specification, and a delta describing the change required.
- To determine if a transformation ***t*** can be preserved in the face of a delta, determine if

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



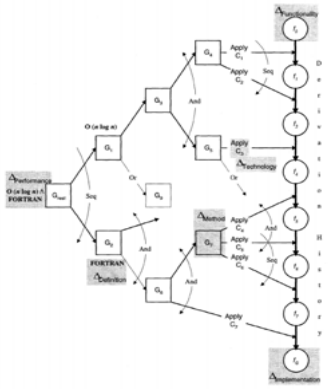
## Design Modification

- If true, then the implementation step accomplished by the transformation is not affected by the change we wish to make.
- If there is no effect, the transformation can be preserved and is copied to a new derivation history.



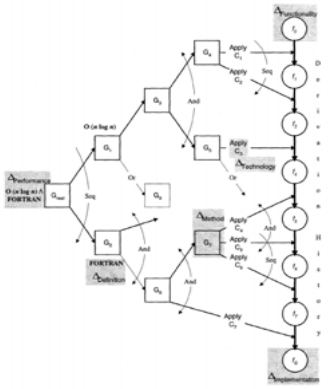
## Design Modification

- If the implementation transform lowers the abstraction level, the delta may also shift levels, to express the change at the lower level of abstraction.
- If we are unable to decide, then we ***banish*** the transformation.
- Banishment is accomplished by commuting the offending transformation with its immediate follower in the derivation history.



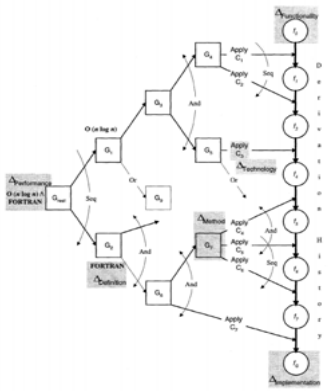
## Design Modification

- DMS walks down the derivation history, deciding whether it must preserve or banish the implementation transformation at each intermediate specification.
- When a transform is reached that cannot be preserved, and cannot be banished because the rest of the transforms depend on it, then no more transforms can be preserved, and **DMS** stops.



# Design Modification

- The already-preserved transformations form a legitimate prefix of a complete derivation for the revised specification.
  - DMS then switches over to a more conventional transformation implementation style to complete the new derivation.



# An Example

- Start with an initial program that accumulates the total price of a set of order records kept in a file
  - Each order record contains an item quantity and a price.
- Final Result will be a practical program in a BASIC-like language

*Original derivation history*

```
sum(i,length(order),order(i).quantity*order(i).cost)
```

# An Example

- The abstract functional specification for the original problem is given.
  - Each box represents an intermediate functional specification derived from the one above it
  - Each intermediate step has exactly the same functionality as the one preceding it.

*Original derivation history*

```
sum(i,length(order),order(i).quantity*order(i).cost)
```

# An Example

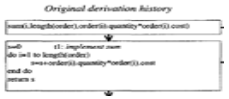
- Each downward arrow connecting boxes represents the application of a single transformation (t1, t2, ...).
- The nature of the transformation corresponding to the arrow immediately above it is shown in italics at the top each box.
- The individual transformations are justified by the performance enhancement each makes.

*Original derivation history*

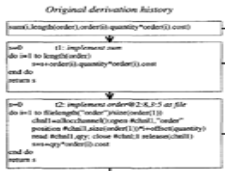
```
sum(i,length(order),order(i).quantity*order(i).cost)
        ↓
s=0          t1: implement sum
do i=1 to length(order)
      s=s+order(i).quantity*order(i).cost
end do
return s
```

## An Example


## An Example

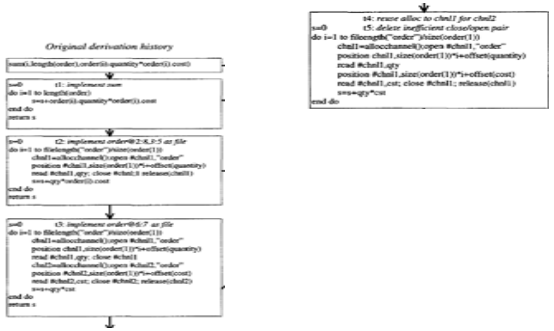



## An Example

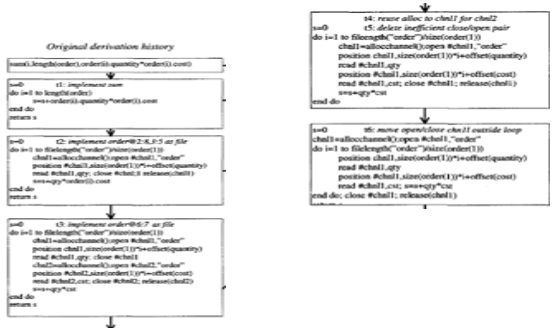



## An Example

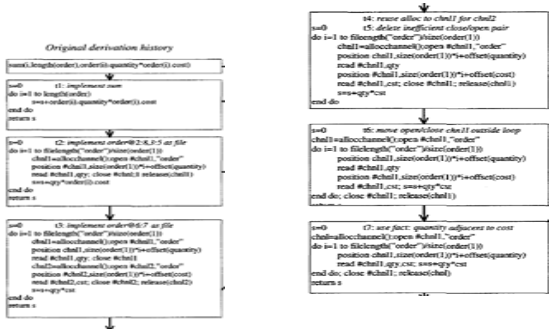
# An Example



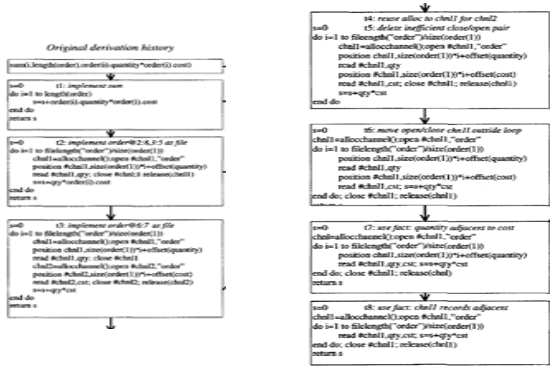

# An Example




# An Example




# An Example



## An Example

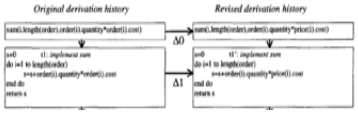
- Now, a new need arises: our manager wishes to keep order quantities in separate files from the price per item.
- This is reflected by the revised abstract functional specification in the upper right box.
- One way to handle such a change request is to simply re-implement the program.
  - However, we assumed that the discovery of transformations used in the original implementation was hard;


## An Example

- We do not expect the discovery to be any easier in a new implementation.
- **DMS** shows how and when transformations used in a prior implementation can be reused in the new implementation, avoiding the rediscovery costs.

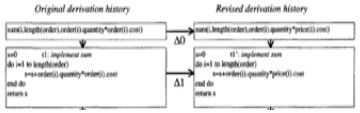

## An Example

- The arrows crossing from left to right show how formal deltas tie the original and new derivation histories together.
- Manager provides Δ0; **order→price@1:17**



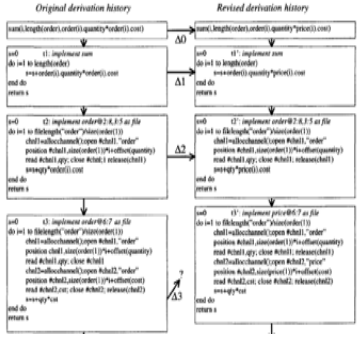

## An Example

- The **DMS** determines that *t1* can be reused as *t1* ', because *t1* does not affect anything related to the **order@1:17**
- The delta must change to reflect the "movement" of the code caused by implementing the loop, giving Δ1.



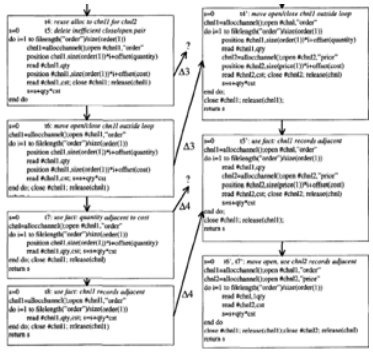
# An Example

- Similarly, *t2* and *t3* can be reused, changing the locators on the delta, giving ∆**2** and ∆**3** respectively.
- Transform *t3'* has not really changed;
  - t3' is written with the variable part bound to the entity *@6:7* ***(price)***




# An Example

- Now both *t4* and *t5*, and their dependent, *t7* cannot be preserved.
- The **DMS** effectively rearranges the order of the original derivation history to delay the application of the failing transformations until last.
  - This moves *t6* and *t8* upward




# Reverse Engineering

- Imagine we only have
  - the system code
  - some informal, inaccurate documentation
  - some understanding of the code distributed across the maintainers.
- Consequently, the typical organization could not carry out this method for maintenance by design modification directly.

# Reverse Engineering

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

## Reverse Engineering

- It is assumed that with sufficient clichés, a complete tiling of the code can be obtained.
- There are a number of flaws to this approach:
  1. ?
  2. ?
  3. ?
  4. ?
  5. ?
  6. ?

## Reverse Engineering

- It is assumed that with sufficient clichés, a complete tiling of the code can be obtained.
- There are a number of flaws to this approach:
  1. There is an assumption that one can get a complete set of widely acceptable clichés.
  2. There is the potential of huge computational demands if one attempts to tile a large system at once.

## Reverse Engineering

3. A complete tiling of the code only raises the abstraction level somewhat.
   - For a large system, it would seem that one should tile the tiles repeatedly to get to the highest level of abstraction possible.
4. The purpose of Reverse Engineering (RE) is generally to aid informal understanding of the code.
   - RE usually results in the production of informal documents under the implicit assumption that the code will be constant.
5. Since code maintenance always changes the code, the RE activity must be repeated for each maintenance event.

## Reverse Engineering

6. Implemented code has all kinds of optimizations that entangle the implementation of abstractions, which disables recognition of clichés.

- One approach for obtaining a design history is to generalize cliché recognition in a way that solves these problems.
- The key observation is that every cliché is a **<abstraction, code template>** pair,
  - can be treated as a transformation rule.
- Then use the transformation engine to recognize clichés and abstract them.

## Supporting Technology

- There are two challenges to implementing the DMS vision.
  - One must have sufficient integrated infrastructure to carry out the steps.
  - It must scale reasonably well.
- Required infrastructure includes:
  1. **A *means for representing the program to be maintained*.**
  2. **A *graph rewrite engine*** to apply individual transformations to the program representation

## Supporting Technology

3. ***Tools to manage a database*** of notations, abstractions, and transforms that might be used in the application.
4. ***Reverse-engineering tools,*** which use the rewrite engine to recognize clichés taken from domains.
5. ***A domain-notation driven structure editor*** to allow maintainers to inspect and point at portions of partially derived applications in the appropriate domain notation.

## Implementation

- Scale management for DMS occurs at two levels:
  - The size of the application system
  - The number of engineers who maintain it.
- DMS is more effective for systems with hundreds of thousands of lines, but cannot be handled by individual maintainers.
- DMS is implemented in a parallel processing language, ***Parlanse,*** running on Windows NT multiprocessor workstations.

## Implementation

- DMS will not generate all 10 million lines of code when a change is made, but it might have to inspect a significant fraction of the 10 million transformations it has stored as the design of that system.
- It is not a requirement that DMS be able to automatically generate and apply transformations by itself. However, DMS will have a built-in implementation of a programmable Transformation Control Language to provide some automation.

## Conclusion

- This paper shows a scheme for capturing the design of transformationally synthesized code.
  - Given the design, incremental changes can be installed by use of mechanical procedures and some additional transformational synthesis.
- Such capabilities should decrease the cost of maintenance, and therefore significantly lower the cost of software.
- ***Suggestion:*** treat the design (**rather than code**) as the primary product of the software process.