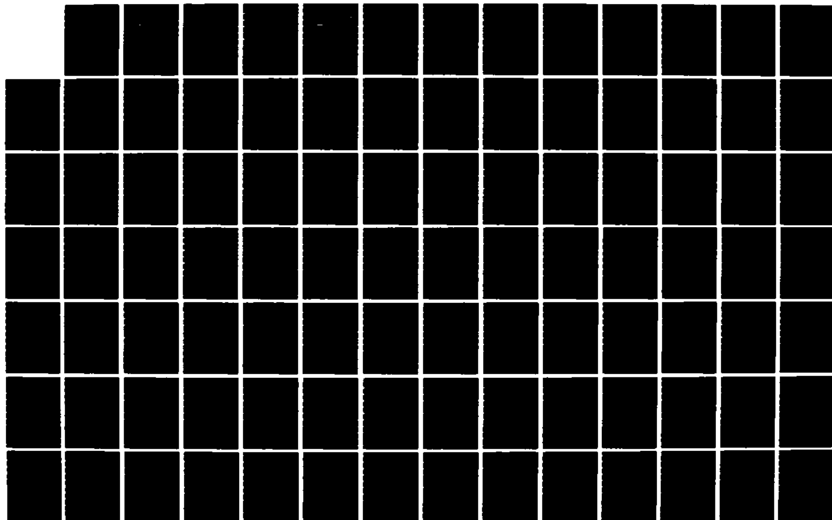
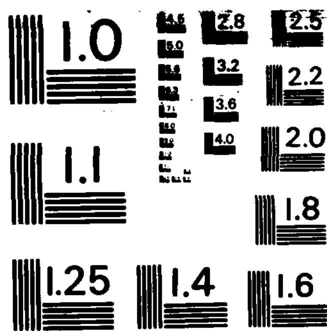
AD-A139 860 AUTOMATING THE TRANSFORMATIONAL DEVELOPMENT OF SOFTWARE VOLUME 1(U) UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFORMATION S.. S F FICKAS MAR 83 1/3

UNCLASSIFIED ISI/RR-83-108 NSF-MCS79-18792 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ISI/RR-83-108
March 1983

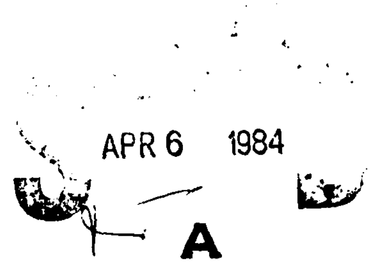
AD A139860

Stephen F. Fickas

University of Southern California

Automating the Transformational Development of Software
Volume 1

DTIC FILE COPY



INFORMATION SCIENCES INSTITUTE

213/822-1511
4676 Admiralty Way/Marina del Rey/California 90291-6695

84 04 03 024

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

Field	Content
1. REPORT NUMBER	ISI/RR-83-108
2. GOVT ACCESSION NO.	AD-A139 86[illegible]
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle)	Automating the Transformational Development of Software Volume 1
5. TYPE OF REPORT & PERIOD COVERED	Research Report
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s)	Stephen F. Fickas
8. CONTRACT OR GRANT NUMBER(s)	MCS-7918792
9. PERFORMING ORGANIZATION NAME AND ADDRESS	USC/Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
11. CONTROLLING OFFICE NAME AND ADDRESS	National Science Foundation 1800 G St. N.W. Washington, D.C. 20550
12. REPORT DATE	March 1983
13. NUMBER OF PAGES	198
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	....
15. SECURITY CLASS. (of this report)	Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	

16. DISTRIBUTION STATEMENT (of this Report)

This document is approved for public release and sale; distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

....

18. SUPPLEMENTARY NOTES

This report was the author's Ph.D. dissertation at the University of California, Irvine, Department of Information and Computer Science. The author's current address is Department of Computer Science, University of Oregon, Eugene, OR 97403.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

automated software development, automation and documentation of software development, interactive software development system, problem solving, transformational implementation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(OVER)

APR 6 1984

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT

This report proposes a new model of software development by transformation. It provides a formal basis for automating and documenting the software development process. The current manual transformation model has two major problems: 1) long sequences of low-level transformations are required to move from formal specification to implementation, and 2) the problem-solving used to reach an implementation is not recorded. Left implicit (and undocumented) are the goals and methods that lead to transformation applications, and the criteria used to select one transformation over another. The new model, as incorporated in a system called Glitter, explicitly represents transformation goals, methods, and selection criteria. Glitter achieves a user-supplied goal by carrying out the problem-solving required to generate an appropriate sequence of transformation applications. For example, the user asks Glitter to eliminate a data structure that would be expensive to store or a function costly to compute. Glitter achieves this by locating all references to the offending construct and devising an appropriate substitution for each. Glitter was able to automatically generate 90 percent of the planning and transformation steps in the examples studied. This report is published in two volumes. Volume 1 contains the text of the report; Volume 2 is a set of seven appendices relating to and illustrating the text in Volume 1.

ISI/RR-83-108
March 1983

Stephen F. Fickas

University of Southern California

Automating the Transformational Development of Software Volume 1

Accession For
NTIS GRA&I
Dist
A-1

INFORMATION SCIENCES INSTITUTE

213/822-1511
4676 Admiralty Way/Marina del Rey/California 90291-6695

This research is supported by the National Science Foundation under Contract No. MCS-7918792. Views and conclusions contained in this report are the author's and should not be interpreted as representing the official opinion or policy of NSF, the U.S. Government, or any person or agency connected with them.

Contents

List of Figures vii

Acknowledgments viii

Chapter 1: Introduction 1
- 1.1 A Problem Solving Approach to Software Design 2
- 1.2 The Software Problem From a Mechanization Point of View 4
- 1.3 The Transformational Implementation Model 6
- 1.4 The TI Model as a Foundation 8
  - 1.4.1 General Automation Issues 11
  - 1.4.2 Advantages of the TI model from an automation perspective 13
  - 1.4.3 Needed Enhancements of the TI model 13
- 1.5 Thesis Layout 17

Chapter 2: The Glitter System 19
- 2.1 Components of the TI model 19
- 2.2 Components of the Glitter model 20
  - 2.2.1 The User's Role 23
- 2.3 Glitter Interface 24
  - 2.3.1 Top Level 25
    - 2.3.1.1 Set faith mode (FaithMode) 25
    - 2.3.1.2 Pretty print planning state (PrPlan) 26
    - 2.3.1.3 Pretty print program state (PrProg) 27
  - 2.3.2 Manipulating the development state (DevTree) 27
    - 2.3.2.1 Print development tree (PrTree) 27
    - 2.3.2.2 Change states (NewState) 28
    - 2.3.2.3 Print current path (PrPath) 30
  - 2.3.3 Manipulating the planning space (PlanSpace) 30
    - 2.3.3.1 Mark goal as achieved (MarkGoal) 30
    - 2.3.3.2 Post goal (Post) 31
    - 2.3.3.3 Print candidates (PrCSet) 32
    - 2.3.3.4 Critique method (Critique) 33
    - 2.3.3.5 Choose method (Choose) 33
- 2.4 Hearsay-III Implementation 34
  - 2.4.1 State/Space Representation 35

Chapter 3: The Gist Specification Language 37
- 3.1 The Package Router Problem 38

3.2 Relational Model of Information 39
3.3 Predicates and Expressions 41
3.4 Change in the Domain 42
3.4.1 Procedures 42
3.5 Temporal Reference 43
3.6 Demons 44
3.7 Constraints and Non-determinism 44
3.7.1 Non-determinism 45
3.7.2 Constraints 46
3.8 Derived Relations 48
3.9 Closed Specification 49
3.10 Total Information 50

Chapter 4: The Development Partnership 53
4.1 The User as Organizer 53
4.1.1 An overview of the package router specification 53
4.1.2 Organization of the router development 55
4.2 The user as consultant/troubleshooter 57
4.3 Summary 68

Chapter 5: A Development Vocabulary 69
5.1 Goal Representation 70
5.2 Characterization of the TI Development Space 73
5.2.1 Mapping Specification Freedoms 73
5.2.1.1 Mapping Information Freedoms 73
5.2.1.2 Mapping Operation Freedoms 74
5.2.1.3 Mapping Efficiency Freedoms 74
5.2.2 Applicability Conditions 75
5.2.3 Jittering 75
5.2.4 Simplification 76
5.2.5 Pragmatics 77
5.3 Coverage 77
5.3.1 Mapping of Specification Freedoms 78
5.3.2 Applicability Conditions 78
5.3.3 Jittering 78
5.3.4 Simplification 79
5.3.5 Pragmatics 80
5.4 Robustness 80
5.5 Extending the Language 81
5.5.1 User Goals 84
5.6 Direct Invocation 85
5.6.1 Applying transformations 86
5.6.2 Manually editing the program 86
5.6.3 Invoking a method 87
5.7 Goal-descriptor Summary 91

Chapter 6: Development Methods 93
6.1 Properties of Method Representation 94

6.1.1 Knowledge accessibility 94
6.1.2 Adding new methods 95
6.1.3 Coverage of development planning space 95
6.1.4 Automatic jittering 96
6.1.4.1 Eagerness 98
6.1.4.2 Prudence 99
6.1.4.3 Level of Effort 99
6.1.5 Representational transparency 100
6.2 Method Template 100
6.2.1 Some examples of method construction 102
6.2.1.1 Construction of the method MergeDemons 102
6.2.1.2 Construction of the method RemoveRelation 106
6.2.1.3 Construction of the method MaintainDerivedRelation 108
6.3 Method Vocabulary 110
6.3.1 Mapping function 110
6.3.2 Predicates, Generators and Accessors 111
6.3.3 General Functions 112
6.3.3.1 pattern-match 112
6.3.3.2 gist-type-of 113
6.3.3.3 domain-type-of 114
6.3.3.4 brother-of 114
6.3.3.5 case-of 114
6.3.3.6 maintenance-location 114
6.3.3.7 reference-location 115
6.3.3.8 trigger-location 116
6.3.4 Access Functions 117
6.3.4.1 component-of 117
6.3.4.2 immediate-component-of 118
6.3.4.3 pairwise-component-of 118
6.3.4.4 component-correspondence 119
6.3.4.5 body-of 120
6.3.4.6 parameter-of 120
6.3.4.7 trigger-of 120
6.3.4.8 local-var-of 120
6.3.4.9 scoped-in 121
6.3.4.10 name-of 121
6.3.4.11 update-relation-of 121
6.3.4.12 new-value-of 121
6.3.4.13 recursive 122
6.4 Direct Method Invocation 122
6.5 Hearsay-III Method Representation 122

Chapter 7: The Selection Process 123
7.1 Selection Criteria 124
7.2 The Glitter Selection Process 125
7.2.1 The initial candidate set 127
7.2.1.1 Inferring that knowledge is missing 127
7.2.1.2 Set size, saturation, etc. 128

7.2.2 The weighted candidate set 128
7.2.2.1 Method-specific rules 130
7.2.2.2 General selection rules 132
7.2.2.3 Problem solving resource rules 133
7.2.3 Final Candidate Set 134
7.2.4 Final Set ordered 136
7.2.5 Method chosen 136
7.2.6 Method actions ordered 137
7.2.7 Critic Mode 138
7.3 Program features used in selection 138
7.4 Problem solving features used in selection 142
7.4.1 Method Inspection 143
7.4.2 The Goal Tree 145
7.4.2.1 Goal-specific knowledge 145
7.4.2.2 Meta-goals 146
7.5 Extending the Rule Catalog 147
7.6 Hearsay-III Rule Representation 149
7.6.1 The Scheduler 149

Chapter 8: Related Work 151
8.1 The PSI system 151
8.2 The CHI system 153
8.3 The Programmer's Apprentice 153
8.4 The FOO system 155
8.5 The IPMS system 156
8.6 The DRACO system 157
8.7 The DEDALUS system 158
8.8 The ZAP system 159
8.9 The PADDLE system 161

Chapter 9: Summary and Future Work 163
9.1 Automation Issues Revisited 163
9.1.1 Formalization of the Development Process 163
9.1.2 Detail Management 166
9.1.3 Glitter As a Development Partner 167
9.1.4 The Development History 169
9.2 Usability 170
9.2.1 The model 170
9.2.2 Tool Usability 172
9.3 Some Maintenance Examples 173
9.3.1 A browsing example 173
9.3.2 A re-implementation example 175
9.4 Final Grades 178

References 181

List of Figures

Figure	Page
1-1. The TI Report Card	9
2-1. TI model	20
2-2. Glitter model	21
2-3. Development tree example	29
3-1. Package Router	39
7-1. Goal context available to selection process	143
9-1. Development exploration tree	169
9-2. The Glitter Report Card	178
9-3. The Space Race	180

Acknowledgments

I gratefully acknowledge my thesis supervisor, Bob Balzer, for his intellectual and moral support. It was Bob who suggested the transformation automation problem as a thesis topic; his keen insight and broad grasp of the field have kept the major points in focus during my research.

I would like to thank the other members of the Transformational Implementation group at ISI: Don Cohen, Martin Feather, Neil Goldman, Jack Mostow, Bill Swartout and Dave Wile. Each brings a unique perspective to the problem of software development. Their expertise in transformational software development is a keystone of the work reported here. It was truly invigorating to have so many sharp minds around; I have benefited from my association with them all.

The other members of my committee -- my chairman Dennis Kibler, Peter Freeman and Dennis Volper -- have read and commented on earlier versions of this thesis; their effort is appreciated. Both Dennis K. and Peter have provided much sound advice during my graduate career. I would also like to thank Dennis K. for the logistic support he has provided for my research.

Lee Erman and Phil London provided able assistance during the construction of the Hearsay III implementation of my system. Without their willing help, I imagine I would still be coding.

Discussions with Jim Neighbors and Gene Fisher have provided enlightening and clarifying views of "the software problem".

The support staff at both UCI and ISI, to an individual, has been incredibly helpful; I thank them all.

Finally, I thank my family and friends who never seemed to doubt that I could bring this work to completion.

Chapter 1
Introduction

This thesis centers on two interconnected propositions: 1) the design process of mapping an abstract program specification into an efficient compilable algorithm can be partially automated, 2) a formal, machine usable history of the design process including goals, methods and selection criteria can be recorded. We will argue not only that both are of value, but that with the high relative cost of maintenance, the latter will become the linchpin of future maintenance efforts.

We will support these propositions by presenting a system that automates part of the design process. The system is based on a model of software development that starts with an abstract, formal specification and *transforms* the specification into a compilable implementation. To handle realistic specifications, the user is expected to play an active part. We will show the documented history of the design process produced by the system is completely captured for the automated portion of the design process and tools are provided to encourage the user to document the goals, methods and decisions of the remaining manual portion.

We will find each of the following items of use in this model:

- A language for stating design goals. The language will become the communication medium between user and machine.
- A catalog of methods for achieving those goals. The method catalog will contain both transformations and planning techniques.
- A catalog of selection rules for choosing among competing methods.
- A well defined partnership[1] with the user which plays off the strengths of both user and machine.

---

[1] We use the word *partner* where others have used *assistant* or *apprentice*. *Coadjutor* may come closest to our intended meaning: equivalent status in all but authority. We will stick with partner.

- ☐ A detailed recording of the design process that includes the goal structure, competing methods and selection rationale used to reach an implementation.

1.1. A Problem Solving Approach to Software Design

A general view of our system is that of a problem solver in the domain of software design. The user issues design related problems (goals) and the system finds means of solving (achieving) them[2].

Designer: *These two parts of the spec are similar; let's consolidate them.*

Designer: *This part of the spec is unneeded (nonessential, redundant); let's remove it.*

Designer: *This part of the spec is complex; let's break it into simpler cases.*

Designer: *This part of the spec is ready to design; let's implement it.*

We would like the system to achieve the user's goals automatically, bothering the user only when interesting (e.g., insightful, organizational, domain specific) information is needed. That is, the system should carry out the mundane detailed steps of the design, allowing the user to concentrate on higher-level development issues.

System: *I'm trying to find an equivalent replacement for this object. I believe that the last element of this sequence is a good candidate for these reasons... Can you verify that they are equivalent?*

System: *In what order should these parts be designed?*

System: *How big do you estimate this* <*fill in domain object*> *will be?*

We would like the system to document its design process in a way that can be used by other tools, e.g., a maintenance tool. The queries below are based on 1) the development history produced by the system and 2) a *hypothetical* maintenance tool which makes use of it; only the former is directly addressed in this thesis.

Maintainer: *How was this portion of the implementation introduced?*

---

[2] We use English here as a reading aid only; the actual language for stating problems is described in chapter 5.

Maintainer: *What role does this design step play?*

Maintainer: *Why wasn't this method chosen at that point in the design?*

Maintainer: *Under what circumstances can I get rid of this portion of the design?*

Maintainer: *Is this portion of the design re-usable if I make the following change to the specification?*

These objectives have largely been achieved by a running system called Glitter[3], upon which the above interactions are based. Glitter is implemented in Hearsay-III, a system for constructing expert systems [Balzer 80, Erman et al. 81]. Glitter has been applied to several development problems, the most interesting of which is presented in Appendix C. Later chapters will discuss the software design model that Glitter embodies in more detail; we provide here a brief description of its features

- *What vs. How*. The model advocates a shift in the way transformational developments[4] are constructed. Instead of the user deciding *how* to achieve some implementation by searching through a transformation catalog for appropriate transformations and then selecting one, the user decides *what* development goal he wishes to achieve by the use of the goal language. The system then makes available all methods which might achieve the goal and all knowledge that might help choose among them. Note that the same what vs. how issues are raised in specifying programming problems (see chapter 3).

- *Automation*. For the package router development presented in Appendices, the system was able to produce 159 planning steps given 13 user goals.

- *Partnership emphasized*. The design process is viewed as a joint activity with the strengths of each partner emphasized. This requires both a mutually understandable form of design knowledge and a control structure that uses both partners in an efficient manner. Note the divergence of approaches between a partnership model and that of automatic programming. In the latter, full automation is achieved by studying constrained examples; research progresses by working on gradually tougher problems. In Glitter, tougher problems can be handled by including the user; research progresses by gradually removing the user from the process.

---

[3] An historically rooted acronym: Goal-directed jitterer. Rather antiquated currently.

[4] Our use of the term *development* in this thesis is limited to the process of transforming a specification into a compilable algorithm. Hence, *design* and *development* will be used interchangeably.

- *Knowledge acquisition facilitated.* As development experience is gained, it is expected that new knowledge, missing from the current system, will come to light. Mechanisms are provided for recognizing and recording holes in the system's knowledge base; such holes are reported at the end of a development for human analysis. The recognition mechanism keys off of actions taken by the user during the development process. From this recording process, new methods and selection rules are formed (by hand; no learning mechanism currently exists).

- *Development history recorded.* The by-product of a Glitter development is the rich planning structure which sits on top of the actual transformation steps. This structure is available to other tools. In particular, we have begun to make use of it in the proposal of a maintenance tool (see Section 9.3).

In the next section, we provide motivation for both a mechanized approach to software development and its automation.

1.2. The Software Problem From a Mechanization Point of View

A major stumbling block in the way of the growing use of computers is the problem of producing high quality, maintainable software. Hammer and Ruth [Wegner 79] clearly state its breadth:

> ... much of the software produced today is either costly or unreliable, and often both; in effect the production of software is a process that is out of control. Software is rarely produced on time or within budget; when delivered, it often fails to meet its specifications or to provide the function for which it was conceived; and all too frequently, it operates incorrectly or not at all. As users address ever more complex applications, and as the price of hardware continues to drop, software costs spiral upwards and dominate other costs of computer-based application systems.

Listed are some of the contributing factors, categorized around major life-cycle processes:

- *Specification.* The construction of software specifications is a difficult task. Any specification is likely to have some combination of missing, imprecise or inconsistent requirements. Because errors of this type may not manifest themselves until deep into the development (imprecision and inconsistency during implementation, omission during testing or after delivery), they can be among the most expensive and difficult to correct.

- *Implementation.* Producing production code from informal specifications is an error-prone and difficult task to control. Resulting software is unlikely to fully meet specifications or be resource optimal.

- *Documentation*. Although standards exist, documentation of large evolving systems is often useless. Frequently this is because of after the fact documentation, often by a third party distinct from developers and maintainers. Even documentation produced as a by-product during development is generally incomplete, hard to understand and difficult to relate to the actual system. As a system is changed, documentation is rarely correspondingly updated. All this makes the already difficult task of maintenance more so.

- *Maintenance*. The majority of system costs and energy over the total life-cycle are devoted to modifying software to meet changing specification requirements. There are several factors making maintenance a generally onerous task. The first is lack of documentation as discussed above. The second is the loss of structure brought about by program optimization steps. A finely tuned piece of software likely has information spread throughout and a high degree of interdependency. Finding all code segments that must be modified and keeping the modifications self consistent is beyond the expertise of most programmers in such a delocalized and documentation-poor environment. The ripple effect of adding k new bugs for every one squashed is now a maintenance cliche.

Software Engineering, a field concerned with addressing these issues, has evolved from an earlier time when machines were expensive and in limited supply. The consequence is that current software development practices are informal and many times undocumented. In the current era of cheap machines and expensive people, it has become apparent that a new approach is necessary, one based on the machine playing an active part in developing programs. The minimal capability would be record-keeping. However, once the machine is involved in the process, one can contemplate various forms of mechanization, including synthesis, analysis, tuning and maintenance.

In mechanizing the production of software, the major life-cycle products and processes must be formalized. There is a dark side to formalization: details that were implicit or ignored in informal models now have to be attended to. For instance, much software today is either specified in English or not at all. Such informal specifications rely on the common sense of the *human* reader to fill in missing detail and disambiguate ambiguous portions (often erroneously). A formal specification for machine consumption must explicate such detail down to the minutest level. The same problem arises in the development process. What previously was accomplished with a favorite text editor now becomes an effort in applying correctness preserving transformations and worrying about the many attendant details that such a formal approach carries as baggage. Our view, one that seems to be supported by empirical evidence, is that the mechanized model of software development is not and will not

be widely used by practitioners of the field until much of the detail is managed by the machine. The model we propose in this thesis addresses one part of this management, the automation of the detailed steps necessary in transforming formal specifications into abstract algorithms. It is a logical extension of earlier work in the area of software development mechanization based on a Transformational Implementation (TI) model. In the next section we examine the TI model in more detail. We will comment on TI's strengths, discuss its weaknesses and show how the Glitter system extends it to meet automation needs.

1.3. The Transformational Implementation Model

Since 1976, a group at Information Sciences Institute has been actively constructing an interactive model of program development based on a Transformational Implementation paradigm [Balzer 76, Balzer 81, Feather 82a, London & Feather 82]. (see [Bauer et al 77, Cheatham 81, Darlington 81] for related models). Glitter is one part of this overall effort.

The components of the TI model include

1. A formal, abstract, operational, specification language called Gist [Balzer et al. 78, Balzer & Goldman 79, Goldman & Wile 79, Swartout 82].

2. An interactive transformation engine called TI, which incrementally maps specifications into implementations [Balzer 76, Balzer 81].

3. A Gist interpreter for symbolically executing Gist programs and explaining their behavior so that specifications rather than implementations can be validated [Balzer et al. 82, Cohen et al 82].

4. A system called PADDLE that allows a developer to record his development as an executable program [Wile 81a]. This program can be run during maintenance to automatically produce portions of the original development.

The Gist language acts as the common interface between model components. We will be concerned with only the first two components above in this thesis: the specification language Gist and the TI transformation engine. When we reference the *TI model of* or *TI approach to* program development, we will be referring to these two unless otherwise noted.

Specification

The TI approach to program development involves mapping a program specification written in a high level specification language into the implementation of an efficient compilable algorithm through a predefined set of correctness preserving transformations. Both specification and implementation are described by Gist. Since Gist contains both a specification subset and an implementation subset, as well as various intermediate subsets, it is viewed as a wide-spectrum language [Bauer et al 78] (Chapter 3 provides an example-based introduction to the Gist language.).

The TI model supports the evolutionary approach to specification construction. Specifications do not spring to life in their full glory, but evolve from incomplete and ambiguous forms into the desired final problem description (see [Swartout & Balzer 82] for a discussion of the intertwining of specification and implementation). Because Gist is an *operational* specification language, specifications can be executed and the results used to validate that the specification meets the user's intentions or point to portions of the spec which require further elaboration. A related effort is the construction of a natural language Gist paraphraser [Swartout 82]. Gist specifications, like any formal specifications, tend to be unreadable[5]. The paraphraser can help a user discover discrepancies between what the specifier thought he said and what he actually wrote by converting the Gist specification into an English description.

Implementation

The target of the TI transformation process is high level in the sense that it allows spontaneous computation (demons) and structures data in a relational data base. To produce the final production code, the implementation produced by TI must be further compiled. There is currently an effort within the TI group to build such a compiler. Other program development systems have also shown the ability to do at least limited compilation at this level (see for instance, [Barstow 79a], [Neighbors 80]).

The effects of transformation application in TI can be classified as mapping *specification freedoms* found in Gist into objects and operations which exist at the implementation level. The mapping process may involve mapping operational freedoms, informational freedoms or

---

[5] We will attempt to overcome the problem in this thesis by an analogous manual paraphrasing process.

efficiency freedoms. Because Gist is wide spectrum, both *refinement* and *optimization* are part of the general mapping process and neither is distinguished. The person involved in a TI development will likewise not be distinguished as the *refiner* or *optimizer*, but simply *user* or *developer*.

The development is carried out by applying catalogued transformations to Gist program fragments. This process is semi-automatic in that a programmer must both choose the transformation to apply and the context in which to apply it; the TI system ensures that the left hand side (LHS) of the transformation is applicable and applies it. The application of a transformation produces a new program state. A final development is a series of transformation applications leading from the initial specification to the desired implementation. The TI model supports the notion of development exploration by allowing a user to revert to some previous program state and explore a new development path by selecting an alternative transformation application.

Maintenance

The development process, whether it be manual or automatic, spreads information throughout a program. What was local and understandable in the spec becomes splintered , smashed and difficult to understand in the final program. This directly affects the ease of modifying a program and leads to much confusion among managers and programmers: what appears to be a trivial change at the specification level generally turns out to be a difficult and error-prone task at the concrete program level. In TI, maintenance is shifted from the final optimized code to the program specification. For each specification change, a new development must be produced (Wile [Wile 81a] suggests ways this effort can be reduced. See also section 9.3).

1.4. The TI Model as a Foundation

We will refer to the TI model as we have described it thus far as the *base-line* model or system. In it, the user is responsible for deciding what transformation to apply and where to apply it. The system is responsible for the faithful application of the transformation. In this section we will argue that this model forms the right foundation for an automation effort. We will first look at its strengths and then some necessary enhancements.

TI Report Card	
1. Ease of Specification	B
2. Efficiency of the Implementation	C
3. Ease of Maintenance	B/D
4. Correctness of the Implementation	A
5. Resources Required	C
6. Type of Problems Handled	B+
Comments: *Does not live up to full potential.*	

Figure 1-1: The TI Report Card

Bob Balzer has suggested six criteria for judging the power of a program development system [Balzer 73]. Figure 1-1 grades the TI model on each of the six. We are grading absolute (straight-line) as opposed to relative (on the curve). **Caveat1**: assigning a single grade to such large categories is less than accurate. The grades are used as a general guide; the discussion following provides the necessary detailed description. **Caveat2**: these six criteria formed the basis of the GIST/TI approach (see also [Balzer 72]). The reader can judge how skewed each is to the TI model. The grading rationale is as follows:

1. *Ease of Specification* (B). The construction of a specification can be measured along several axis:

- Does the system take into account the difficulty of writing correct specifications? Can the specification be, at least initially, incomplete or ambiguous? How are incompleteness and ambiguity discovered? By providing a Gist symbolic interpreter, TI supports an incremental, evolutionary approach to specification construction. Further, a Gist paraphraser exists for providing an English-like description of a subset of Gist.

- How easy is it for the user to translate domain objects and operations into Gist constructs? Gist was not designed around any particular application domain. The question revolves around the general applicability of Gist's model of computation. Of the few domains studied, Gist was capable of handling the corresponding objects and operations. On the negative side, each Gist specification starts from scratch. Thus the considerable domain analysis that must go into producing most specifications must be done anew for each new problem (compare this with Neighbor's Draco system [Neighbors 80] which attempts to reuse domain analysis).

- Is the user forced to make design and implementation decisions during specification? A fundamental tenet of Gist is that the user be able to specify *what* the problem is without specifying *how* it is to be solved.

2. *Efficiency of the Implementation* (C). TI focuses on the mechanical aspects of development rather than the cognitive aspects. As such, it is neutral with respect to the decision making process. Hence, the efficiency of the final implementation rests on the skill of the user. While this may seem the natural way of things, we expect the complexity of a TI development will prohibit all but the most expert user from obtaining a fully optimized implementation.

3. *Ease of Maintenance* (B/D). The B reflects the ease of recording a change: all changes are made at the highest problem description. The D reflects the difficulty of producing an implementation that incorporates the change: a brand new development must be carried out to produce a new implementation. Further, the only guidance the maintainer has is the record of the original transformation steps: there is no indication of the goals the developer was following or what choices were rejected and why. We note that one of the original motivations of the TI model was the lack of documentation provided in informal design processes. Hence, a formal transformation record was produced. We now are complaining that the transformation record leaves the problem solving process undocumented, clearly a case of rising expectations.

4. *Correctness of the Implementation* (A). Given validated transformations and system application, the resulting development guarantees a valid implementation. Note the difference with Program verification: a TI development starts with a specification and *maintains* the behavior specificed during the development process; Program verification attempts to *connect* a final program with its specification in an after the fact manner. Incremental maintenance of the proof is one of the big wins of a transformation system.

5. *Resources Required* (C). While not part of Balzer's original criteria, we consider the user's required participation as a major component of resource measurement. In the TI model, the user is required to carry out all transformation selections.

6. *Type of Problems Handled* (B+). Although our experience base is small, it appears that a large range of problems is specifiable by Gist. We expect the catalog of Gist transformations to grow as new applications are attempted. On the slightly negative side, Gist currently does not address issues of programming-in-the-large, i.e., breaking a problem into separate pieces to be developed by separate groups, defining interfaces, integrating implementations into a final system.

It should be noted that the TI model, and to a lesser extent the model we present in this thesis, trade off items 5 and 6, i.e., resources required and type of problems handled. By incorporating the user in the development process, we increase the types and complexity of problems we can handle. On the negative side, the user may be required to put a significant amount of his time into a development. Systems which offer complete automation ([Barstow 79a, Manna & Waldinger 79]) provide the contrast: they remove reliance on the user at the cost of working on more constrained problems (e.g. smaller problems, more limited domains, lower level specifications).

1.4.1. General Automation Issues

One goal of this thesis is to show that portions of the design process can be automated. There are at least four major issues that must be addressed:

1. *Process formalization*. Current Software Engineering development methodologies focus on the *products* produced in developing software as opposed to the *processes* which produce those p.oducts. Automation demands that we formalize and capture the development process in machine usable form. Once inside the machine, it can be documented, analyzed, understood and modified.

2. *Detail management*. Much of what occupies a developer's time is attending to mundane detail, detail that detracts from the more intellectually challenging problems of design and implementation. This becomes even more so as development processes are formalized. In particular. details that were ignored or dealt with only implicitly now become explicit.

3. *User's role*. Early efforts in programming automation strived for true automation, eliminating the need for the human programmer altogether. However, the complexity of the programming task forced such systems to study small, constrained problems. While several of these systems provided significant results in software automation, they took the sometimes deserved rap of working in toy domains (sorting, list manipulation). One inference from this is that the automation of software production is destined to remain an academic exercise for the foreseeable future. We do not believe that this has to be the case if we lower our sights and allow the user to enter the loop. With the machine taking on a partnership role, the potential exists for tackling much tougher and useful programming problems. As more of the programming process becomes formalized, the less we must rely on the user, a somewhat bottom-up approach to automatic programming.

However, placing the user in the loop presents some corresponding problems:

- An interface or communication line must be established. A mutually understood means of *talking about* program development must be defined.
- A model of the system's knowledge must be available to the user. Further, the user must be able to augment or enhance the system given some perceived missing piece of knowledge.
- A model of the user's knowledge must be available to the system[6]. That is, the system must know what types of knowledge the user can be expected to supply.

4. *Documentation*. The specification, design, implementation and maintenance phases of software production cannot be viewed independently. That they are by a number of models is a reflection of the problems found in the current state of affairs. The entire process history must be available to any particular tool on request. Thus, documentation becomes more than an informal static description of code read by the new guy on the project; it records, in a dynamic fashion, each of the development processes. It is updated as the system and its requirements evolve. We expect it to include, in a machine readable form, the following types of information:

- Specification rationale. What role does each specification modification play. Does it help disambiguate, constrain, enhance. Does it reflect a modification to meet changing requirements.

---

[6] Our use of *user* here is the general as opposed to individual sense.

- Development rationale. What role does each development step play. Why was it chosen. What other alternatives were competing. Why were they rejected. What portions of the spec is the step dependent on, e.g., what performance requirements does it help satisfy.

This information must be gathered anew as a by-product of each new program development.

1.4.2. Advantages of the TI model from an automation perspective

How well does the base-line TI model meet each of the above four problems:

1. *Process formalization*. The model formalizes the development process as a sequence of transformation applications.

2. *Detail management*. Some detail is taken care of by having the system worry about program transformation applications.

3. *User's role*. The user is a major part of the development process. He provides the overall guidance and the sophisticated reasoning necessary to insure that a transformation's applicability conditions are met. A repository of individual development steps is defined in the form of a transformation catalog. Adding new transformations is straightforward. Although it only has to be done once, proving that they are correctness preserving may be not so straightforward (see [Gerhart 75, Broy & Pepper 80] for work in this area). Mitigating the problem, people currently appear to do a good job of informally verifying transformations. While this does not replace the need for formal verification, it allows useful work to continue without it.

4. *Documentation*. The development process is documented in a machine usable form as a sequence of transformation applications.

1.4.3. Needed Enhancements of the TI model

The base-line TI model provides a solid base from which to build. Below, we look at some of the enhancements necessary to meet our thesis goals of automation and documentation.

Process formalization

In the base-line TI model, only part of the development process is formalized. Much of the work in complex problem solving domains such as program development involves 1) formulating the right goals or tasks to pursue and 2) finding the right strategies or plans for refining them into more manageable subgoals, satisfying any pre-conditions and *finally*

manipulating the program through transformation application. In the base-line TI model, only the program manipulation steps are explicated. For example, one high level plan for implementing a spec is "first refine all abstract control structures to operational ones and then work on refining abstract data structures"[7]. Note that plans of this type are far removed from actual manipulations of the program, often passing through many intermediate problem states.

To formalize the problem solving process, we must define both a language for stating particular problems (goals) and building plans for solving those problems (achieving those goals), or as McDermott puts it, a problem vocabulary [McDermott 77]. The base-line system provides nothing in the way of a development problem or goal language[8]. The transformation catalog does provide a limited form of plans, ones that work in the program space. However, it is likely that many of the development goals will be far removed from the actual transformations that finally realize their achievement. Plans that map goals onto goals -- transform or elaborate the problem space -- are missing.

Detail management

Our experience with transformational developments [Balzer 76, Balzer 81, London & Feather 82] has produced an important result: most of the transformation steps are not the interesting and clever optimizations we expect from expert programmers, but the mundane preparatory and clean-up steps which are the filler between them. Often, the attention which must be paid to these steps distracts a user from the more important optimizations that lead to real performance gains.

The base-line TI model takes care of the details of applying a transformation faithfully to a program. However, this is one of the least interesting aspects of detail management. More importantly, we would like the automatic selection and application of entire sequences of low level transformations to meet some higher level development goal.

---

[7] A much simplified example. In reality, such a plan would be too rigid to be useful

[8] In some sense, the left hand side of a transformation can be viewed as a goal to be achieved; its achievement (matching) allows the transformation to take place.

User's role

Although the base-line TI model does provide for a machine/user partnership, it is a rather one-sided effort. The machine's sole responsibilities are transformation application and recording; the user is responsible for all else. Our experience is that this degree of automation is not enough. The machine must help organize the development and automate as much of the development as its knowledge base allows. This has several aspects:

- The catalog of development techniques must not only contain the tactical knowledge embodied in program transformations, but the strategic type of knowledge useful for organizing larger chunks of the development.

- We need the ability to identify and collect the set of tactics or strategies useful in achieving some development goal. This is part of a more general problem: the ability to incorporate and make use of knowledge at the appropriate times. The catalog of development methods can be expected to both grow large and be under a constant state of change as new methods are added, old ones deleted and others modified. In the base-line TI model, the user is responsible for both knowing what is in the catalog and finding it when needed. Even with cleverly constructed names, manually searching a large catalog of transformations for ones that are applicable to the current development task is both tedious and error-prone. Note the irony here: as the system becomes more knowledge rich through the addition of more transformations, the partnership as a whole becomes weaker because of the decreasing likelihood of the user successfully searching the catalog for the set of applicable transformations.

- As a catalog grows, we would expect many candidate methods to be available for achieving a particular goal. Trying each is intractable. Selecting the best one to apply is generally a non-trivial task and one that the machine should participate in. In the base-line TI model, the machine takes no part in the selection process.

Documentation

The record of the development process is expected to be used by other TI tools. For example, a maintenance tool might need to determine the relationship between two steps in a development: is one a preparatory step for the other; are both sub-components of some higher level plan; are both totally independent; can one be replaced by another? (chapter 9 provides examples of these type of questions in a maintenance scenario) The base-line TI model provides a sparse history of development, noting simply the transformation sequence. This type of history cannot be used to answer any of the maintenance questions above. What is needed is the planning structure that sits above the actual program transformation steps.

This includes the goal/sub-goal tree and the selection criteria at each node. The more of the development process that remains in the user's head, the less effective will be other tools relying on the development history. The base-line TI model pries only a small portion from the user.

In the remainder of this thesis, we describe in detail how the Glitter system provides the necessary components to both automate and document the process of developing a Gist specification using the TI model. In particular, we address each of the issues above:

1. The development *process* is formalized and captured by the machine. This can be viewed as an extension of the PADDLE system [Wile 81a], a TI tool for *structuring* a development discussed in chapter 8.

2. Significant amounts of the detailed steps found in a development are automated, freeing the user to concentrate on higher-level development issues.

3. A man/machine partnership is defined. We show how the partnership can play off the strengths of each member to develop non-trivial programs. Further, Glitter is *knowledge-based*: it provides catalogs of development knowledge that can be extended and analyzed by both user and machine.

4. Our documentation ideal is to provide a common development data base that specification, development and maintenance tools will all share. The development history produced by Glitter is the first cut at such a data base.

1.5. Thesis Layout

The general layout of the thesis is as follows:

1) high level introduction -- Chapters 1-4
2) heart of the model -- Chapters 5-7
3) wrap-up -- Chapters 8 and 9
4) package router development details -- Appendixes A-D
5) problem solver components -- Appendixes E-G.

Below we give a brief summary of each individual Chapter/Appendix:

Chapter 1: general introduction of thesis goals and approach.

Chapter 2: overview of the system and its components including user interface.

Chapter 3: an introduction to the software specification language Gist, sufficient to understand the development examples found in this thesis.

Chapter 4: a discussion of the man/machine interaction that occurs during development, illustrated in part by an annotated development transcript.

Chapter 5: a discussion of issues related to development-goal representation.

Chapter 6: a discussion of methods needed to achieve development goals.

Chapter 7: a presentation of the selection process used to choose among competing methods.

Chapter 8: a discussion of related work.

Chapter 9: a summary of a) software development automation issues and how well they have been met, and b) the usability of the model and system. The use of the development history as the input to a future maintenance tool is also discussed and illustrated through several examples.

The appendixes make up a large part of this thesis. This is largely due to our decision to provide an extended development as opposed to several fragments. We believe this gives the "big picture" and a much more continuous view of things. However, there are clearly problems with both presenting and following a *textual* description of a lengthy development[9].

---

[9] As shown in chapter 2, the development history is actually stored in machine usable form. When sitting at a terminal, this allows a more interactive and useful presentation.

To overcome these, we have provided two development overlays. The first provides the planning structure with minimal detail. The second provides the selection knowledge that went into each development step. When both are overlayed with the detailed development, they provide the full planning structure.

Appendix A: the Gist specification of the package router problem. Chapter 3 gives the English problem statement, Chapter 4 a textual description of the key components of the specification.

Appendix B: an overview of the router development is presented that highlights the planning structure. This is extremely brief and hence is useful as a guide to examples scattered throughout the thesis.

Appendix C: the detailed, 100 step router development, minus selection information. Most of the examples in the thesis are taken from here. Chapter 4 provides a high level description of this development.

Appendix D: an overlay of the selection process carried out to produce the development of Appendix C.

Appendix E: an example based description of Glitter's goal language; the detailed counterpart of Chapter 5.

Appendix F: the method catalog; the detailed counterpart of Chapter 6.

Appendix G: the selection rule catalog; the detailed counterpart of Chapter 7.

Chapter 2
The Glitter System

In this chapter we will look in a little more detail at the Glitter development model and the system which implements it. First however, we will revisit the TI model which forms the basis of Glitter.

2.1. Components of the TI model

Figure 2-1 gives a graphical depiction of the baseline TI model of development. The components of the model include the following:

- The initial program state. A Gist specification[10].
- The transformation catalog. An unindexed collection of correctness preserving, program transformations.
- The user. The user is responsible for selecting a transformation to apply and a context from the current program state in which to apply it.
- Transformation applier. Takes the transformation T and context C selected by the user and checks if T is applicable in C. If so, applies it to produce a new program state.
- Final implementation. The model is run iteratively to produce a *path* through the development space. A path consists of a an alternating sequence of program states and transformation applications starting in the initial state and ending in an implementation state. The full output of the model is the path to the implementation along with each alternative path followed, i.e., the tree of alternative developments.

[10] Every Gist specification is a program which can be executed and every program is a specification of some lower level implementation. We will use *program* and *specification* interchangeably.

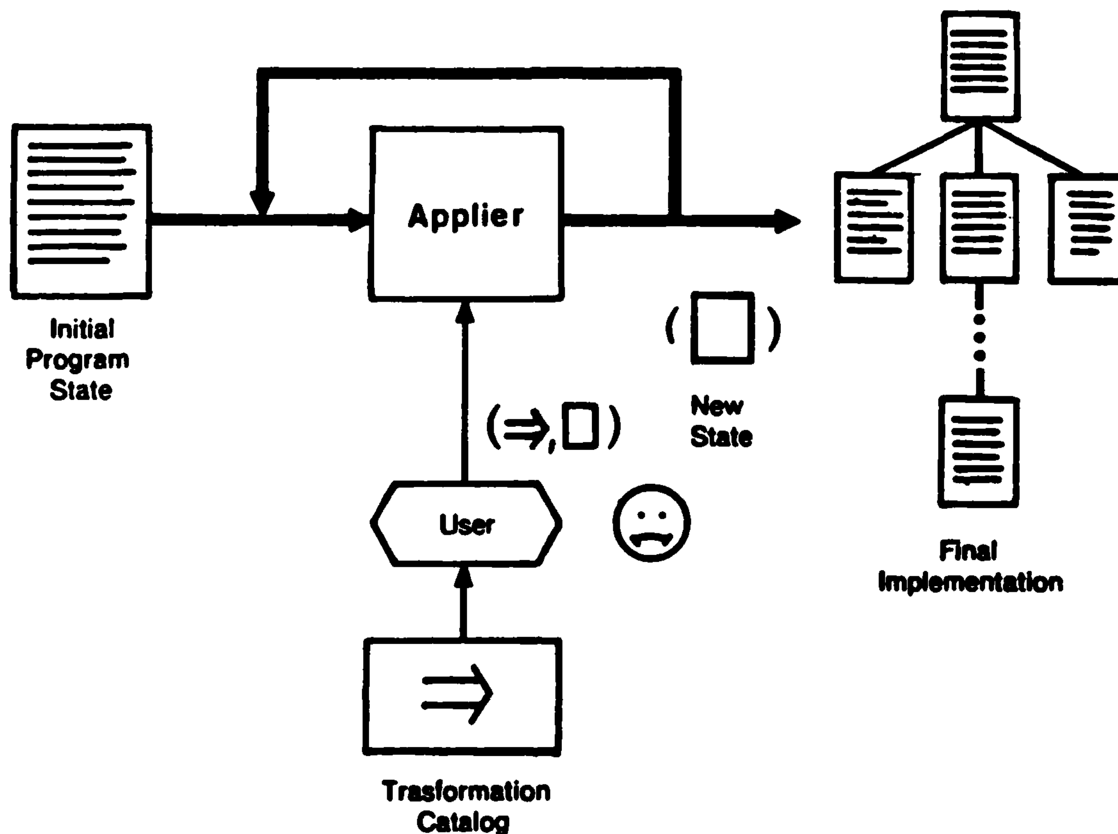


Figure 2-1: TI model

2.2. Components of the Glitter model

As can be seen in figure 2-2. the Glitter and TI models are similar in several ways. Both take an iterative approach to development, both include the user as a part of the development process, both encapsulate development knowledge in terms of catalogs of useful development techniques. However, Glitter attempts to apply the automation lever to a much greater degree than TI. A discussion of the Glitter model components will help illustrate this:

- *Initial Problem Solving state*. A Glitter problem solving state consists of two items:

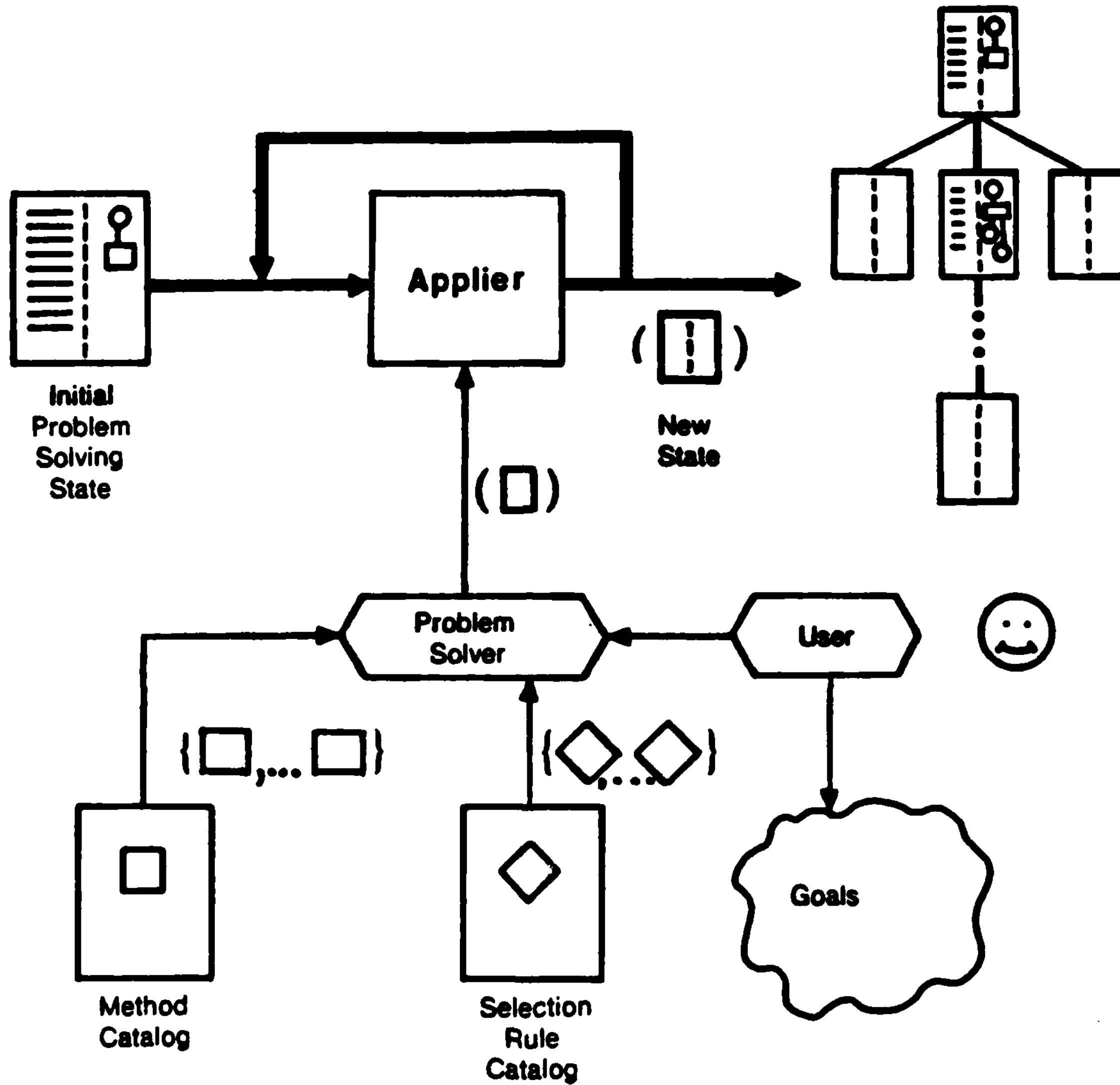
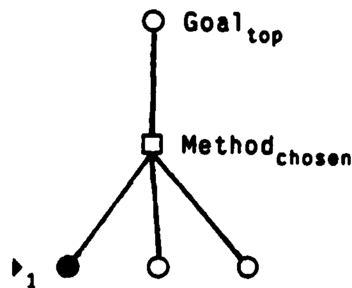


Figure 2-2: Glitter model

a *program* state and a *planning* state. As in the TI model, a program state is simply a Gist program. A planning state consists of a goal/method tree of the following form[11]:

---

[11] We will consistently use ○ to represent a goal and □ to represent a method. Dotted lines represent potential methods. ▸ will be used to point to parts of a diagram or program we wish to highlight.



This state shows the following planning history:

1. $Goal_{top}$ was posted.

2. $Method_{chosen}$ was selected to achieve it.

3. The application of the method generated three new subgoals, one ($▸_1$) of which is currently active.

For the sake of clarity, we have left out the selection criteria from the above diagram, e.g., what other methods were competing with $Method_{chosen}$, why they weren't selected. All of this information is part of the planning state; chapter 7 describes in detail how it is used.

In the initial problem solving state, the planning state consists of a primordial goal "develop program".

- *Development goals*. Glitter provides the user with a development vocabulary in the form of goal descriptors (or simply goals). The user communicates with the problem solver through the goal language. A goal has a life of its own independent of the methods that are indexed to it, i.e., a goal may not be satisfied after the method chosen to satisfy it completes.

- *The user*. The user is responsible for posting development goals.

- *Problem Solver*. The Problem Solver does the grungy work for the user. It takes a user's goal, finds a set of candidate methods for achieving the goal, finds a set of selection rules that help order and refine the candidates and finally chooses a method for application, passing it on to the method applier. Note that a single user goal will likely require the Problem Solver to call on the Applier repeated times, i.e., the Problem Solver is our automation lever.

- *Method catalog*. The method catalog is used by the Problem Solver to achieve development goals. That is, the Problem Solver retrieves from the catalog all methods that claim to achieve the currently active goal. Note that this set of methods (alternatives) becomes a permanent part of the development history; at any time[12], the user may back-up to any node in the development tree and both

[12] In the case of maintenance, the time would be after the initial development was complete.

examine the competing methods for any particular goal and spawn a new branch by selecting an alternative.

In general, a method need not make a *program transformation*, but may instead simply refine the current goal into more manageable subgoals, what we might call a *problem transformation*. This allows the method catalog to grow as new goals are introduced through the use of current capabilities.

- *Selection Rule catalog*. The selection rule catalog is used by the problem solver to help refine and order the methods retrieved from the method catalog. In particular, the selection rule catalog contains rules that examine the current planning state, candidate set and past planning states order competing methods. As with the method candidate set, the information provided by the selection rules is made a permanent part of the development history; at any point in the development, the user may examine why a particular method was chosen or rejected.

- *Method applier*. Takes a method M and applies it, i.e., applies each of M's actions in turn.

- *Final implementation*. The model is run iteratively to produce a path through the development space. Iteration here is in nested loop form: the user repeatedly posts development goals: the Problem Solver repeatedly selects methods to achieve each specific goal. A path consists of a an alternating sequence of problem solving states and method applications starting in the initial state and ending in an implementation state. As in the TI model, the output includes the tree of development alternatives. Further, each node in the tree contains the information which lead to the various branches from it, e.g., the methods that were competing. the selection knowledge used to order those methods.

2.2.1. The User's Role

The user's role in the Glitter model we have presented thus far is predicated on a large catalog of both development methods and selection rules. That is, the user's sole responsibility is to guide the overall development by iteratively posting successive development goals; the Problem Solver is expected to take care of the rest in a non-interactive fashion. Unfortunately, the system's current small experience base leads to holes in its knowledge. As we shall discuss in following chapters, the user may incrementally add new knowledge to the system as experience is gained. However, this does not mitigate the limited knowledge the system has initially. In order to remain a useful partner, the system has expanded the user's role and provided him with a more fine grained control. Besides guiding the overall development, the system relies on the user in the following ways:

1. If the Problem Solver is unable to find any methods to place in the initial candidate set (or if the selection rules rule out all of the methods in the initial set) then the system relies on the user to either a) define a method dynamically (see section 2.3.3.2) or b) choose some previous state to back-up to (see section 2.3.2.2).

2. In computing the applicability of a method or selection rule, the Problem Solver will likely rely on the user to supply formal reasoning steps. Chapter 4 provides a more detailed example and discussion of such steps.

3. The user may place the system in certain *faith modes* which currently include trusting, cautious and critic modes. These modes allow the user to exert various degrees of control over the Problem Solver: in trusting mode the Problem Solver is allowed to choose and apply methods without user approval; in cautious mode the user can examine the result of running the selection rules on the candidate set (see section 2.3.3.3) and is allowed to override the system's choice (see section 2.3.3.5): in critic mode the candidate set is formed but control is returned to the user before the selection rules are run. In the latter case, the user can ask for a critique of a candidate method M, i.e., ask the system to run all of its selection rules pertaining to M (see section 2.3.3.4). In both cautious and critic modes, the user has the option of backing-up to some previous state.

4. Moving around the development tree can be motivated by either the desire to explore various implementations or the need to back out of some dead-end state. In neither case does the system offer any assistance in *selecting* the right state to move to; it does provide him with the means to *examine* the development tree and *move* about it (see section 2.3.2).

As Glitter's knowledge base is augmented through experience, we foresee the user's role moving back towards our idealized model. However, even with a powerful problem solver, it is likely the user can provide other types of guidance or control, e.g., advice on methods to employ, highlighting of critical decisions. High level advice of this type has been either implemented or postulated in other development systems [Feather 82b], [Wile 81a]. We believe it can be useful in future versions of the Glitter system as well.

2.3. Glitter Interface

In this section, we present the user/Glitter interface (The reader may wish to skip this section until reaching the development transcript in Chapter 4 where it is seen in actual use.). The interface is menu driven and currently assumes a CRT display. Because a development is clearly information intensive and often graphical in nature, we plan to re-implement the interface on a bit-mapped display at some future time.

We will organize the discussion around the three menus that the system provides: the top level menu (2.3.1) and its two subordinates DevTree (2.3.2) and PlanSpace (2.3.3). We will first present a menu and then an example of each of its commands. We will try to present the command descriptions in the same order as they appear in the menu, i.e., alphabetically. However in some cases we may change the presentation order to avoid forward reference. All text appearing as ***bold italics*** is input from the user; anything else is printed by the system.

2.3.1. Top Level

```
>Glitter: ?

     one of:
       DevTree -   manipulate program development tree (sub-menu)
       FaithMode - set trusting, cautious or critic mode
       PlanSpace - manipulate planning state (sub-menu)
       PrPlan -    pretty print a portion of the planning state
       PrProg -    pretty print a portion of the program state
       QU -        quit

>Glitter:
```

2.3.1.1. Set faith mode (FaithMode)

```
>Glitter: FaithMode set faith mode

      Mode (trusting, cautious, critic): cautious
```

This command allows the user to provide various levels of control over the selection process. In trusting mode, the system attempts to provide as much of the problem solving as possible. In cautious mode, the system returns control to the user when it is about to make a method selection. In critic mode, the system returns control to the user after the candidate set of methods has been formed bt before any selection rules have been run. Chapter 7 discusses these modes in more detail.

2.3.1.2. Pretty print planning state (PrPlan)

```
>Glitter: PrPlan pretty print planning state

      Print how much of tree (<cr> for concise)? <cr>

  + Map set_switch_on_exit
                          ...using MapByConsolidation

       Consolidate  set_switch_on_exit and package_leaving_switch
                          ...using MergeDemons

         Equivalence trigger and trigger
                          ...using Anchor1

           Reformulate switch_is_empty as expression
                          ...using ReformulateDerivedRelation

            ! Unfold switch_is_empty in trigger
                          ...using ScatterComputationOfDerivedRelation

           =* Reformulate existential as expression

>Glitter:
```

While the user may wish to see the entire goal structure, more often only a local subset will be of interest. The system prompts for the goal which will act as the root of the tree to be printed. A special *concise* mode is made available for printing the portion of the goal tree between the current high level user goal (in this case, the *Map* goal) and the currently active goal. This is useful for establishing a problem solving context without wading through too many details.

The display consists of a set of nested entries. Each entry consists of a goal followed by its arguments and the method chosen for achieving it. The latter is shown as "...using method-name". The system decides how to print a goal's arguments in concise form: names are used when working with defined objects such as demons, procedures, and relations; part names are second best, e.g., trigger, body, argument list; otherwise the type of the argument is printed. Indentation represents sub goal structure, e.g., the Equivalence goal is a sub goal of Consolidate which is a sub goal of Map. A goal may be preceded with one or more special marks: + denotes a goal posted by the user; ! denotes a goal that has been achieved; * denotes the currently active goal; = denotes a goal that is re-posted. The above example is taken from the package router development and represents the planning state at step 6.11 in appendix C.

2.3.1.3. Pretty print program state (PrProg)

```
>Glitter: PrProg pretty print program state

      Print what portion (<cr> for entire program): switch_is_empty


(1:1)
    relation switch_is_empty(switch)
      definition ~exists package || package:located_at = switch;


>Glitter:
```

The program state is stored internally in parse tree form. PrProg produces the text equivalent. The numbers in parentheses, (1:1), give the current development state (see 2.3.2.1).

2.3.2. Manipulating the development state (DevTree)

```
>DevTree: ?

     one of:
       NewState - change problem solving states
       PrPath - print the current development path
       PrTree - print the entire development tree
       QU - quit


>DevTree:
```

2.3.2.1. Print development tree (PrTree)

```
>DevTree: PrTree print the development tree

     1:1 *
        2:1
           3:1
        2:2 *
           3:2 *
              4:1 *
                 5:1 +

>DevTree:
```

The development tree represents the possible alternative paths leading to some implementation. Each node in the tree represents a problem solving state. Each arc represents the choice of one of possibly many competing methods. The tree is used to structure the development exploration space. By moving between nodes, the user can back track from a dead end state, resume a previously abandoned path or choose some new alternative method to employ.

The PrTree command prints the textual form of the development tree; figure 2-3 gives the corresponding graphical equivalent, clearly a better representation. Each pair of numbers is the name of a single problem solving state (node): the first number gives the level of the state in the development tree and the second is a generated uniqueness number. Indentation corresponds to level, visual but redundant information. States at the same level are alternative branches, e.g. 2:1 and 2:2 are branches from state 1:1. The current state is marked with "+". States on the path to the current state are marked with *.

2.3.2.2. Change states (NewState)

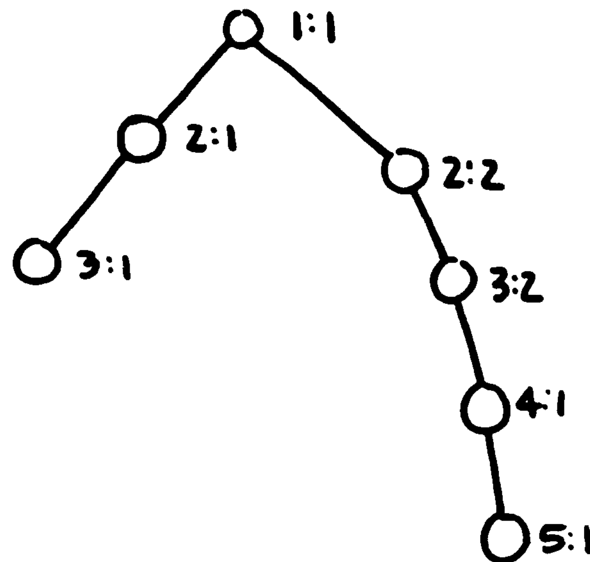


Figure 2-3: Development tree example

```
>DevTree: NewState change current problem solving states

      Change current state 5:1 to what: 3:1


>DevTree: PrTree print the development tree

    1:1 *
      2:1 *
        3:1 +
      2:2
        3:2
          4:1
            5:1


>DevTree:
```

The NewState command allows a user to move to any state within the current development tree. In the above case, state 3:1 is marked as the currently active state and development continues from there.

2.3.2.3. Print current path (PrPath)

```
>DevTree: PrPath print the current development path

  States on path: (1:1, 2:1, 3:1)

  State changes:
       1:1 to 2:1...method MapByConsolidation applied
       2:1 to 3:1...method MergeDemons applied

>DevTree:
```

2.3.3. Manipulating the planning space (PlanSpace)

```
>PlanSpace: ?

     one of:
       Choose -   choose a method from the candidate set to invoke
       Critique - critique a candidate method
       MarkGoal - mark current active goal as achieved
       Post -     post a goal
       PrCSet -   print method candidate set of goal
       QU -       quit

>PlanSpace:
```

2.3.3.1. Mark goal as achieved (MarkGoal)

```
>PlanSpace: MarkGoal mark current active goal as achieved

            New active goal is
               ...

>PlanSpace:
```

An error message is generated if the current active goal is not a user-controlled goal (see section 5.5.1).

2.3.3.2. Post goal (Post)

```
>PlanSpace: Post post a goal

        Goal: Map
           Freedom: set_switch

>PlanSpace:
```

The user enters the goal name and the system prompts for each of the goals arguments (see chapter 5). There are several different effects of this command:

- If there exists a currently active goal G then this command causes 1) an ad hoc method to be created for G, 2) the posted goal to become the action of the method and 3) the method to be chosen and applied, i.e., the posted goal becomes active. An ad hoc method is one that is created on the fly during development, i.e., it is not part of Glitter's general method catalog. Its actions will be filled in as the development progresses.

- If there is no currently active goal then an ad hoc method must have previously been created and still be active[13]. The posted goal is added as a new action to the ad hoc method and made the currently active goal.

---

[13]Note that the initial problem solving state contains a system defined ad hoc method. The high level goals posted by the user are placed under this method (see section 2.2).

This command allows methods to be defined as a development progresses. In this way, it is much like the PADDLE system [Wile 81a]. After a development is complete, such methods can be considered for inclusion in the method catalog, normally after they have been suitably generalized.

2.3.3.3. Print candidates (PrCSet)

```
>PlanSpace: PrCSet print candidate set

   Current active goal is

     • Equivalence expression and object

   Candidate methods are

    -> 1. Anchor2 (+2)
            Action1: Reformulate expression as object

       2. Anchor1 (-)
            Action1: Reformulate object as expression

   Selection information is

       Anchor2 given +2 by rule *Anchor2a

   Current choice of system is Anchor2

>PlanSpace:
```

In this example, taken from step 1.15 in the router development, two methods are competing to achieve the Equivalence goal, Anchor1 and Anchor2. Both are printed with their current weighting (see section 7.2.2) and instantiated action sets. If any of a candidate method's actions will be trivially achieved if the method is selected, it is so noted (by a preceding !). In general, this ability to "look inside" a candidate method is valuable to both user and system (see for instance, *content reference* as it is described in section 7.2.2.2).

Any selection information that has been found is printed. In this case, the selection rule *Anchor2a has been run and has given a weight of +2 to the method Anchor2, the rationale

being that an expression can often be replaced with an equivalent object value. No information has been found regarding Anchor1.

Finally, the system prints the method it would select if asked to choose now.

2.3.3.4. Critique method (Critique)

```
>PlanSpace: Critique critique a candidate method
   Which method (<cr> to print set): Anchor2

   Selection rules firing are
       *Anchor2a: gives +2 to method Anchor2

   Current choice of system is Anchor2

>PlanSpace:
```

When running in critic mode (see section 2.2.1), this command allows the user to selective examine what information the system can provide on one or more methods. Our example above is again taken from step 1.15. Being in critic mode, the user has been given control after the posting of the *Equivalence* goal and construction of the candidate set, but before any selection rules have been run. Here he requests information on a particular candidate method Anchor2. This causes the system to fire the corresponding selection rule *Anchor2a.

2.3.3.5. Choose method (Choose)

```
>PlanSpace: Choose choose a method
            Which method (<cr> to print set): Anchor2

       Method Anchor2 has been invoked.

       Current active goal is
            Reformulate expression as object

>PlanSpace:
```

When running in either cautious or critic mode, the user is responsible for making the method selection. In cases where the user's choice is different than the one preferred by the system, a note is made of the discrepancy, e.g. in what state did it occur, why did the system prefer some other method, what information was known about the user selected method.

2.4. Hearsay-III Implementation

Glitter is implemented in Hearsay-III, a system which provides a framework for constructing knowledge-based expert systems [Balzer 80, Erman et al. 81]. In [Erman et al. 81], a detailed description is given of an earlier version of Glitter called the Jitterer [Fickas 80]. For the most part, the Hearsay-III organizations of both systems are the same. We point the reader to the earlier paper for details.

To avoid forward referencing problems, we will describe the components of the Hearsay-III implementation in a distributed manner. At the end of the relevant chapters, we will provide a summary of how the mechanisms presented in that chapter are represented in Hearsay-III. In particular, here we will describe the state-space representation; in section 6.5, we describe the representation of Glitter's methods; in section 7.6, we describe the representation of Glitter's selection rules and the overall scheduler. Again, these descriptions are brief; the user is urged to look at the previously cited papers for more details.

2.4.1. State/Space Representation

A *problem solving state* consists of a *program state* and a *planning state*. We use the Hearsay *blackboard* to represent a single problem solving state. The program state is represented by a parse tree which is implemented as Hearsay blackboard units and role/component pairs[14]. The planning state is represented as an AND/OR tree which is implemented as role/component pairs and *hypothesis* units.

The development space -- the set of connected problem solving states -- is generated using Hearsay's *context mechanism*. A change in the current problem solving state (blackboard) causes a new state (blackboard) to be spawned. States are encapsulated in Hearsay contexts. Hearsay keeps track of the tree of contexts and provides the necessary inheritance machinery.

---

[14] A project is under way to convert this relational representation into a more economical Lisp record structure. This will allow Glitter to more easily interface with other TI tools.

Chapter 3
The Gist Specification Language

We will attempt to walk a fine line in this chapter. On the one hand, we will need to look at enough of the Gist language to understand and motivate the ideas and examples of this thesis. On the other, a thorough explanation of Gist would be lengthy and somewhat orthogonal. Other sources exist for gaining an understanding of Gist in particular and specification languages in general [Balzer et al. 78], [Balzer & Goldman 79], [Goldman & Wile 79], [Swartout 82], [London & Feather 82].

Gist is a wide spectrum language from which *Programs* are constructed. We further note the existence of a *specification subset* which is used for describing a desired *behavior*, and an *implementation subset* which is used for describing *efficient algorithms*. Put another way, the specification describes *what* is desired and the implementation *how* to achieve it. The Gist foundations, briefly, are

- No valid implementation need be ruled out. The Gist language does not inherently force certain design decisions. However, it does not enforce any notion of appropriate abstraction level, i.e., it is up to the specifier to choose the level of specification abstraction.

- Natural. The Gist language is an outgrowth of the SAFE project, an attempt to accept English specifications from domain experts and translate these into formal specifications. Gist, SAFE's formal specification target language, was designed to handle the type of specification constructs found in Natural Language problem descriptions. In particular, it allows 1) objects and operations to be described at the domain level, 2) process descriptions, 3) description of the environment, and 4) a specification to be incomplete or ambiguous.

- Testable. Specifications can be (symbolically) executed. Feedback can be used to show incomplete or ambiguous portions of the specification.

We will introduce the individual Gist constructs by way of an example. The example, a postal package router, is used as the basis for the detailed development of appendix C and for many of the examples throughout the thesis.

3.1. The Package Router Problem

Suppose we are given the following description of a package router (taken from [Hommel 80]):

Begin English Spec[15]

A package router is a system for distributing *packages* into *bins*. The physical portion of the system consists of a *source station*, (binary) *switches*, bins and *pipes*. Pipes connect the source station to a switch, switches to switches and switches to bins.

Packages enter the router through the source station one at a time and at random intervals. When a package enters the source station, its destination bin can be determined. Once a package leaves the source station, there is no mechanical means of checking its destination.

Switches have *settings* and *sensors*. Two settings are possible corresponding to the two output pipes emanating from each switch. A switch has an input sensor for sensing when a package is entering the switch and an output sensor on each exit pipe for sensing when the package in the switch has exited. A switch can change its setting only if the switch is empty, i.e., no packages are present between the entry sensor and either of the output sensors.

Packages move through the network by gravity (working against friction). Steady movement through the router cannot be guaranteed, hence packages may bunch up and become *misrouted*. A package is misrouted if its current location is not on the path to its destination bin. Once a package becomes misrouted, we are no longer concerned about which bin it is finally routed to (it clearly cannot be its destination bin). Bunched packages entering a switch can, by clever bending of the input pipe, be individually sensed, but the switch is prevented from changing until the last of the bunched packages exits. That is, if we have a "train" of k bunched packages P_1-P_k entering switch S, S can sense that k packages have passed though but cannot change its setting between P_1 entering and P_k exiting.

To diminish misrouting, the destination of a package is checked when it enters the router at the source station. If its destination is the same as that of the previously entering package, it is released immediately (trains with a uniform destination are fine). Else, it is held up some fixed time t. We cannot assume that t is large enough to guarantee that all bunching is eliminated, i.e., misrouting is possible. When a misrouted package reaches a bin, the misrouting should be signalled by the router.

---

[15] [Swartout & Balzer 82] discusses specification abstraction issues in general, and the abstraction level of this specification in particular.

Given that at any instance there may be many packages within the system, the problem is to correctly route packages whenever possible.

End English Spec

Figure 3-1 provides a graphical representation of the router; Appendix A contains the full Gist specification of the package router. In the remainder of this section, we will look at the major specification constructs that Gist provides, grounded in the package router problem.

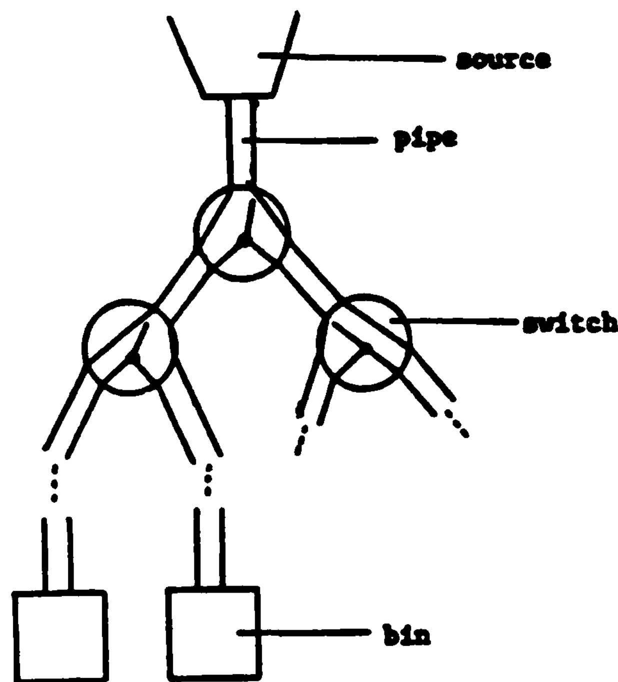


Figure 3-1: Package Router

3.2. Relational Model of Information

Information in Gist is modelled by typed objects and relations among them. The relational model of information permits the specifier to use a descriptive reference to an object:

The pipe that this switch is set to.

The bin that is the destination of this package.

The packages having this bin as their destination.

The relational data model is a very general data representation. For instance, the specifier need not be concerned about data access paths; they are associatively accessible.

The package router domain involves objects of type PACKAGE, SWITCH, BIN, etc.

<u>type</u> PACKAGE;

Type hierarchies are possible; for example, a switch or a bin (or the source or a pipe) is more generally a LOCATION:

<u>type</u> LOCATION() <u>supertype</u> <u>of</u> < SOURCE; PIPE; SWITCH; BIN >

Relations among typed objects model information about a domain:

- The location of a package in the network is modelled by **located_at**, a relation between packages and locations:

  <u>relation</u> **located_at**(LOCATION,PACKAGE)

- The destination of a package is modelled by **destination**, a relation among packages and bins:

  <u>relation</u> **Destination**(BIN,PACKAGE)

- The setting of a switch (i.e., the outlet pipe that the switch is currently set to direct packages into) is modelled by **switch_setting**, a relation between switches and pipes:

  <u>relation</u> **switch_setting**(PIPE,SWITCH)

Binary relations, such as the ones above, are a frequently used form of n-ary relations. Gist provides a syntactic shorthand to more easily declare and access binary relations. This shorthand takes the form of "attributes" associated with types, for example **Destination**, a binary relation between types BIN and PACKAGE, becomes an attribute of type PACKAGE (note that because of the non-directional nature of relations, **Destination** could equally well be made an attribute of type BIN). The simultaneous declaration of types and attributes becomes:

<u>type</u> PACKAGE(**located_at** | LOCATION,**destination** | BIN);

<u>type</u> LOCATION() <u>supertype</u> <u>of</u> < SOURCE(**Source_Outlet** | PIPE);
BIN();
PIPE(**Pipe_Outlet** | SWITCH <u>union</u> BIN);
SWITCH(**switch_outlet** | PIPE, **switch_setting** | PIPE) >

The same effect could have been achieved by defining the attributes **located_at**, **destination**, **Source_Outlet**, **Pipe_Outlet**, **switch_outlet**, **switch_setting** as separate relations.

3.3. Predicates and Expressions

Information about the current state may be retrieved via predicates and expressions denoting objects in the current state.

<u>Expressions</u>:

A package in the domain:	<u>a</u> *package*[16]
A switch in the domain:	<u>a</u> *switch*
The destination of package p:	*p* : **Destination**[17]
The location package p is at:	*p* : **located_at**
A package destined for bin b:	<u>a</u> *package* ‖ (*package* : **Destination** = *b*)[18]

<u>Predicates</u>:

Is package p at its destination?	*p* : **located_at** = *p* : **Destination**

---

[16]Notation. A variable name that is also the name of a type can be used instead of the form <var name>|type. In this example, *package* is shorthand for *package* | PACKAGE.

[17]Notation. <expression> : <attribute name> denotes an object related by <attribute name> to <expression>, in this case a LOCATION.

[18]Notation. The special symbol ‖ should be read as "such that". The construct used here takes the form <u>a</u> <typename> ‖ <predicate> and denotes an object of that type satisfying the predicate.

Existential and universal quantification over objects of a given type is permitted:

Is there a package at every switch?
∀ *switch* || (∃ *package* || (*package*:**located_at** = *switch*))

3.4. Change In the Domain

Change is modelled in the domain by the creation and destruction of objects, and the insertion and deletion of relations among objects. Each such primitive change causes a "transition" to a new state. A Gist specification denotes "behavior" - a sequence of states connected by transitions.

Create a new package: create *package*

Assign bin b as destination of package p: insert *p* : **Destination** = *b*

To include within a single transition several such primitive changes, we embed them inside Gist's "atomic" construct:

Change the location of package p from loc1 to loc2:

```
atomic
   delete p : located_at = loc1;
   insert p : located_at = loc2
end atomic
```

A built-in Gist primitive allows us to state the above change in a more concise form:

update **located_at** of *p* from *loc1* to *loc2*

3.4.1. Procedures

One or more Gist actions can be defined within a *procedure* construct. The procedure construct is parameterized and can be *called* from any number of locations within the spec. Each such call instantiates the procedure's formal parameters with actuals and executes the defined actions accordingly. Note that the Gist view of data is as a global database, and hence procedures are not side-effect-free.

Define a procedure which removes a package from a sequence.

```
procedure TRIM_PACKAGES_DUE_AT_SWITCH(package, switch)
          update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
            to packages_due minus package;
```

3.5. Temporal Reference

The sequence of states connected by transitions leading to (historical) and from (future) the current state is a behavior. *Temporal reference* refers to the ability to extract information from any state in the behavior. Constructs such as asof everbefore, ordered temporally and asof evermore allow the specifier to describe *what* information is needed from earlier and later states without concern for the details of *how* it might be made available. By default, a predicate or expression is evaluated in the current state. Evaluation in some arbitrary state in the behavior is possible. As with reference to objects, specifying the state(s) in which to do the evaluation is done by description - provide a predicate which, when true of a state in the behavior indicates that state is to be used for evaluation.

Following are some examples of temporal reference taken from the package router:

- *Has this package ever been at that switch?*

  ((*package*:**located_at** = *SWITCH*) asof everbefore)[19]

- *The time-ordered sequence of packages ever at the source*:

  ({*package* || (*package*:**located_at** = the *source*) asof everbefore}
  ordered wrt (start(*self*:**located_at** = the *source*)))

[19] Notation. <predicate> asof <state>, or

<expression> asof <state>.

In each case the evaluation takes place in the state(s) in the history (i.e., now or before) designated by <state>. For the predicate, the result will be true if <predicate> held in any of the selected states. For the expression, the result will be the object(s) (non-deterministic if multiple objects) denoted by the expression in any of the selected states. everbefore designates the current state and all past states.

3.6. Demons

Demons are Gist's mechanism for providing data-directed invocation of processes. A demon's trigger is a predicate, which triggers the demon's response whenever a state change induces a change in the value of the trigger predicate from false to true.

Demons are a convenient specification construct for use in situations in which we wish to trigger an activity upon some particular change of state in the modeled environment. They save us from the need to identify the individual portions of the specification where actions might cause such a change and the need to insert into such places the additional code necessary to invoke the response accordingly. The specificational power of the demon construct is enhanced by the power of Gist's other features, since the triggering predicate may make use of derived relationships, historical reference, etc.

Whenever a new package arrives at the source station, do ...

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
   trigger package.new:located_at = the source
   response ...;
```

The trigger of this demon is a predicate that will become true whenever a package becomes located at the source. When the demon is so triggered, occurrences of the variable *package.new* in its response are bound to the instance of the object satisfying that triggering of the demon.

3.7. Constraints and Non-determinism

Constraints within Gist provide a means of stating integrity conditions that must always remain satisfied. Within Gist, constraints are more than merely redundant checks that the specification always generates valid behaviors; constraints serve to rule out those behaviors that would be invalid. The combination of constraints and non-determinism proves to be a powerful specification technique; a specification denotes those and only those behaviors that do not violate constraints. In contrast, an implementation is characterized by the clever encoding of its components to interact in ways guaranteed to result in only valid behaviors.

3.7.1. Non-determinism

Because of its "descriptive" nature, Gist's form of reference to objects may result in several objects satisfying the description. In such a case we say the the expression is non-deterministic.

Any package whatsoever: a *package*

Any package at a bin: a *package* || (*package* : **located_at** = (a *bin*))

An outlet pipe of switch s: a *pipe* || (*s* : **switch_outlet** = *pipe*)
(more concisely) *s* : **switch_outlet**

Non-deterministic behavior results when such a non-deterministic reference is used in a transition. The alternative transitions give rise to distinct continuations (branches) of the behavior. Hence a Gist specification denotes a set of behaviors.

```
demon SET_SWITCH(switch)
   trigger RANDOM()
   response
      begin
         if SWITCH_IS_EMPTY(switch)
          then update :switch_setting of switch to switch:switch_outlet
      end;
```

The SET_SWITCH demon is a non-deterministic expression of behaviors. It presents several non-deterministic choice points. First, it triggers at non-deterministic times. Second, the binding of the variable *switch* is non-deterministic. Third, the update of the switch setting within the response is to a non-deterministic outlet pipe. Given no further constraints, the picture would be of a package router mechanism with switches flapping at random.

Non-determinism may also be introduced through the use of non-deterministic control constructs.

Set each switch to its second outlet.

```
loop S|SWITCH do update switch_setting of s to pipe.2
```
[20]

3.7.2. Constraints

Constraints are used to describe the limitations of the domain and of the desired behaviors. Several constraints on the package router domain are given below[21].

Packages cannot pass through each other.

```
always prohibited PACKAGES_OVERTAKING_ONE_ANOTHER
  ∃ package.1, package.2, location
   || start ( package.1:located_at = location ) earlier than
        start ( package.2:located_at = location ) and

      finish ( package.2:located_at = location ) earlier than
        finish ( package.1:located_at = location ) ;
```

There cannot exist more than one source:

```
   always prohibited  ∃ s1 | SOURCE, s2 | SOURCE || ( s1 = s2 )
```

Switches and bins have a single input pipe

```
always prohibited
        ∃ switch_or_bin | (SWITCH union BIN).
      pipe.1, pipe.2 || pipe.1 ≠ pipe.2 and
          pipe.1 : connection_to_switch_or_bin = switch_or_bin ) ) and
           pipe.2 : connection_to_switch_or_bin = switch_or_bin))
```

Constraints of the above form (on the cardinality of attributes and relations) are common, hence we apply a notational shorthand and declare them at the same time as we make type and attribute declarations. For example, we have defined a package to have two attributes: a destination and a location (a.k.a. **located_at**). For each, the value is unique for a given

---

[20] Notation: loop <expression> do <statement> does the statement for each object denoted by the expression - the non-determinism arises from the non-specified ordering in which to consider the objects.

[21] It is interesting to note that in practice, constraints of this type are often forgotten in the initial spec, perhaps because they so trivial. In any case, it is expected that a spec will be elaborated over time to include the necessary domain constraints (see [Swartout & Balzer 82]). Packaging domain constraints to form an *essential domain spec* could alleviate the problem of every specification starting from scratch (Neighbors refers to this as the problem of reuse of domain analysis [Neighbors 80]).

package, but any number of different packages may share the same value (may have the same destination or location). In Gist, we may specify this as

```
type PACKAGE(Destination | BIN : unique :: any[22],
             located_at | BIN)[23]
```

In the package router, constraints such as MORE_THAN_ONE_SOURCE define the nature of the world in which the specified system will exist. It does not directly constrain the part of the specification to be implemented. Constraints may also be used to rule out those behaviors that would be invalid. In conjunction with non-determinism, they permit us to describe an activity in a non-deterministic fashion; those behaviors of the activity leading to states that violate the constraint are "pruned away". This provides the ability to express our intents more directly (in the form of constraints), rather than encoding all the processes of the specification so as to interact in only those ways that prevent arriving at an undesirable state. The constraint DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE is a good example.

```
always prohibited DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE
 ∃ package,switch ||
    package:LOCATED_AT = switch
      and
    SWITCH_SET_WRONG_FOR_PACKAGE(switch,package)
      and
    ((package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
        and
     SWITCH_IS_EMPTY(switch)) asof everbefore));
```

The system behavior we desire to specify is to route packages correctly whenever possible - given the limitation of not being able to change a switch's setting unless that switch is empty. The constraints that state this desired behavior are the require statement within the body of the SET_SWITCH demon, and DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE. The

[22] Notation: the keyword following ":", in this case unique, constrains how many objects of the attribute type (BIN) may be attributed to the type being defined (PACKAGE); the keyword following "::", in this case any, constrains how many objects of the defined type (PACKAGE) may have as their attribute an object of the attribute type (BIN).

[23] Furthermore, since the unique/any case is typical of many attributes, the default is to assume : unique and :: any

former rules out those behaviors that involve setting a switch when it is not empty[24]. The latter defines a predicate that defines when a switch has not been correctly set. This condition occurs when

- a package is in a switch,
- the switch is set incorrectly for that package, and
- at some time in the past there was a chance to set the switch for that package, i.e., at some time when the switch was empty, the package was the first of those not-misrouted packages due to go through the switch (relation PACKAGES_DUE_AT_SWITCH is defined to hold between each switch and the sequence of not-misrouted packages due to go through that switch).

By putting this predicate into an always prohibited, behaviors which lead to such states are ruled out. Picture a package router mechanism with switches still flapping, but in ways that lead to only desirable behaviors. Note that we have not ruled out the non-deterministic setting of switches, but just constrained it to a desired subset.

The conjunction of non-determinism and constraints serves as a powerful specification technique; non-determinism denotes a set of behaviors, constraints rule out those behaviors containing anomalous states. Hence a Gist specification denotes only the set of valid behaviors.

3.8. Derived Relations

Often it is convenient to make use of a relationship that is derived from others. Its derivation is declared once and for all. The specificational power of this construct comes from being able to state a derivation (that is, an invariant among several relations) in a single place, and make use of the derived information throughout the specification.

Derived relations may be accessed within expressions and predicates in just the same way as any standard relation. They may not, however, be explicitly inserted or deleted - their definitions serve to denote precisely when they hold.

---

[24] Since this is the unique place in the specification where the setting of switches is modelled, we have chosen to use a require statement rather than a global constraint (a stylistic choice).

```
relation PACKAGES_DUE_AT_switch(PACKAGES_DUE | sequence of PACKAGE, switch)
  definition
    PACKAGES_DUE =
      { a package ||
        LOCATION_ON_ROUTE_TO_BIN(switch,package:destination) and
        ~((package:located_at = switch) asof ever) and
        ~MISROUTED(package)
      } ordered wrt (start (package:located_at = the source);
```

This is a derived relation between sequences of packages and a switch. The derivation is a predicate, which defines the sequence of packages (called *PACKAGES_DUE*) in terms of the other argument (the *switch*). This definition is expressed by means of a set of packages upon which an ordering is imposed. For any particular *switch*, the set of packages consists of those for which:

- the *switch* lies on the route to the package's destination,
  LOCATION_ON_ROUTE_TO_BIN(*switch*,*package*:destination)

- the package has not already reached the *switch*,
  ~((*package*:located_at = *switch*) asof everbefore)

- and it is not misrouted,
  ~MISROUTED(*package*)

The ordering puts packages in sequence by the time at which they were located at the source.[25]

3.9. Closed Specification

Gist specifications are closed, in the sense that in addition to describing the portion to be implemented, they also describe (in as much detail as necessary) the environment in which that portion is to operate. Thus the behaviors specified are those required of the entire world. The portion to be implemented must act in such a manner as to interact with its environment to produce a non-empty subset of those behaviors.

The package router is described in a closed world in which packages are created at the

[25] Observe that the structure of the network (a tree with the source at the root) and the property that packages cannot overtake one another combine to ensure that packages will arrive at switches in the same order in which they were located at the source.

source, and are caused to move down through the network. The portions to be implemented (and hence over which we have control) are the source, where packages are released into the network, and the switches. These must perform in such a manner as to cause the correct routing of packages, whatever their destination, and however they might move through the network.

Movement of packages through the router is not within the control of the portion we are to implement, yet must be described in sufficient detail to express the behaviors required of that portion. Movement is modelled by a (non-deterministic) demon that at random causes a random package to move (if possible) to the next location in the router.

```
demon MOVE_PACKAGE(package)
  trigger RANDOM()
  response if ∃ location.next ||
            MOVEMENT←CONNECTION(package:LOCATED←AT, location.next)
        then update located_at of package
             to MOVEMENT_CONNECTION(package:located_at, *);
```

3.10. Total Information

In specifying the behaviors required of the system, it is convenient to make arbitrary retrievals from relations, quantification over all objects of a given type, etc., in order to achieve a straightforward specification. Typically, however, the portion to be implemented will be restricted in the queries it may make of its surrounding environment.

To describe the desired behavior of the package router the constraints, demons, derived relations, etc., make use of knowledge about the destinations and locations of packages anywhere within the routing network.

```
relation SWITCH_SET_WRONG_FOR_package(switch, package)
  definition
    LOCATION_ON_ROUTE_TO_BIN(switch,package:destination) and
   ~LOCATION_ON_ROUTE_TO_BIN(switch:switch_setting,package:destination) ;
```

The mechanical nature of the environment limits observation of the destination of a package to the time at which the package is at the source, and hence an implementation must explicitly read each package's intended destination while it is at the source, and explicitly remember that information in order to control the switches, perform signalling, etc. The *implementation specification* on page 198 defines what information is available to the developer:

```
implement PACKAGE_ROUTER
  observing
    attributes
      package:Destination when package:located_at = the source
...
```

Total information provides the freedom to use any and all information about the system and its environment to specify desired behaviors. It is left to the development of the implementation to determine just what information is useful or necessary and derive it from what is available.

Chapter 4
The Development Partnership

In this chapter, we will take a closer look at the role both user and machine play in the Glitter model. The overall objective is to illustrate the type of interaction that occurs during a Glitter development: what is the user responsible for?; what is the system responsible for?. The package router development (see Appendixes A-D) is used as an explanation vehicle. We will first look at the user's role of development organizer, and then present a small, annotated portion of the router development transcript to illustrate other types of user/machine interaction.

4.1. The User as Organizer

In the Glitter model, the user is responsible for guiding the overall development. In practice, this means he or she must produce the high level goals that drive the problem solving engine and organize the development. In this section we will look at an example of development organization taken from the package router development. Before getting into details, we provide an overview of the package router development as background.

4.1.1. An overview of the package router specification

The Gist specification of the router problem (see section 3.1 for the English statement of the problem) is given in Appendix A. It uses most of the specification freedoms offered by Gist including temporal reference, derived relations, constrained non-determinism, demons and total information. The general task of the developer is to map these freedoms into forms computable in the target language. A general discussion of the mapping of specification freedoms can be found in section 5.2.1. The key components of the router specification that must be addressed in the development include the following[26]:

---

[26] We have excluded from the list the portions of the specification which model the router environment, e.g., creation of packages, their movement by gravity feed.

- RELEASE_PACKAGE_INTO_NETWORK - a demon that triggers when a new package arrives for routing. The portion of the Gist spec that models the environment is responsible for creating packages for routing. Checks if the package has the same destination as the last package. If not, delays its entry into the router for some amount of time. Places the package in the pipe leading to the first switch.

  Specification freedom: data-directed invocation of processes eliminates the need to identify individual portions of the specification where the processes must be invoked. Further, demons can trigger on various events including past events and unobservable events. In this case the event is an observable one produced by the environment.

  Mapping concern: since the target language supports demons and this demon triggers on an observable event, no mapping is necessary.

- SET_SWITCH - a demon that triggers at random times. Selects (binds) a random switch. If that switch is empty, it sets the switch at random to one of the two output pipes.

  Specification freedom: in conjunction with appropriate constraints (see below), allows a specifier to describe the set of all acceptable behaviors without choosing a particular one.

  Mapping concern: non-determinism and constraints *must be* combined to produce only acceptable behaviors.

- DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE - a constraint. Effect is to constrain the demon SET_SWITCH to act in an acceptable fashion. In general, it <u>disallows</u> the following situation: 1) a package P is located at a switch S, 2) the current setting of S will cause P to become misrouted, 3) there was a time when P was the next package due at S and S was empty, i.e., S could have been set so that P would not become misrouted.

  Specification freedom: allows a specifier to limit behaviors without explicating control (see above). The third term of the constraint references a past event (a.k.a. historical reference).

  Mapping concern: constraints (and non-determinism) are not present in the target language, hence they must be mapped. Past states must be remembered.

- PACKAGES_EVER_AT_SOURCE - a derived relation that defines the sequence of packages that have arrived at the source as of ever (a monotonically increasing sequence). The sequence is ordered by arrival time: package1 precedes package2 if package1 arrived at the source before package2 (packages cannot arrive simultaneously).

  Specification freedom: defines information that is useful in the specification (in

this case a sequence of packages) by deriving it from other information (in this case a past event). The specifier does not need to state how this information can be obtained or maintained, simply the invariant relation among packages and their entry into the router.

Mapping concern: Since derived relations are not supported in the target language they must be mapped to a computable form.

- **PACKAGES_DUE_AT_SWITCH** - a separate instantiation of this derived relation exists for each switch. It defines a sequence of packages that are due to arrive at a switch at some later time (size of sequence depends upon capacity of router). As with **PACKAGES_EVER_AT_SOURCE**, the sequence is ordered by arrival time. Because packages cannot overtake one another, it is also ordered by the time packages are due to arrive at the switch: package1 precedes package2 if package1 will arrive at the switch before package2.

  Specification freedom: same as **PACKAGES_EVER_AT_SOURCE**.

  Mapping concern: same as **PACKAGES_EVER_AT_SOURCE**.

- **LOCATION_ON_ROUTE_TO_BIN** - a derived relation that defines a connection matrix between locations and bins within the router.

  Specification freedom: same as **PACKAGES_EVER_AT_SOURCE**, except defines a static relation.

  Mapping concern: same as **PACKAGES_EVER_AT_SOURCE**.

4.1.2. Organization of the router development

The package router development is organized around six high level goals provided by the user. Below we list each and discuss its motivation.

1. *Remove* relation **PACKAGES_EVER_AT_SOURCE**. This relation defines the sequence of all packages that have ever entered the router. Normally, this derived relation would have to be mapped to an explicitly maintained relation with a process to add new packages as they enter the router. However, this relation also is the product of an *efficiency freedom*: the specifier knew that he or she would need to reference previous packages, hence the entire sequence was defined without regard to whether all of it was truly needed. The user (developer) notices this and posts a goal to get rid of the unneeded sequence.

2. *Remove* relation **PREVIOUS_PACKAGE**. In the process of removing **PACKAGES_EVER_AT_SOURCE** in the previous step, this relation is introduced along with the relation **LAST_PACKAGE**. Both relations represent the same basic information, making one of them superfluous. As a clean-up step, the user

chooses to remove PREVIOUS_PACKAGE, leaving LAST_PACKAGE to record the necessary information. Again this can be viewed as the mapping of an efficiency freedom: at any point the specification may contain much redundant or unneeded information. It may be introduced by the specifier because he or she thought it would be necessary, convenient, or made the specification more understandable, or by the development process as in this case. It is the developer's task to remove it before the final implementation is reached.

3. *Remove* relation LAST_PACKAGE. This relation is part of the residue of removing PACKAGES_EVER_AT_SOURCE. It represents the only part of the package history we really need, i.e., the last package to enter the router. Once again, however, this information is overkill: the only information that need be remembered about the last package is its destination. This step is then in the same vein as the last two: further optimize the storage of information, i.e., map an efficiency freedom. The outcome is that a new relation is defined that records just the destination.

4. *Map* constraint DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE. The task here is to decide a switch setting strategy. In general, this constraint must be combined with the non-deterministic behavior of SET_SWITCH to get necessary deterministic switch setting action. This is clearly the most difficult step in the development.

5. *Map* relation PACKAGES_DUE_AT_SWITCH. This derived relation defines a dynamic sequence of packages for each switch. It provides a type of information freedom: assume that the location of all packages can be determined at all times and that they can be ordered by the time they are due to arrive at switches along the route to their destination. Since in the router the only time a package and its destination are identified is when a package enters the network, PACKAGES_DUE_AT_SWITCH must be explicitly maintained by the system. That is, packages must be added to the end of the sequence as they enter the router, taken off the front when they enter the switch, and taken out of the middle when they become misrouted.

6. *Map* demons. One specification freedom is that demons may trigger on unobservable events and/or non-deterministically. For instance, a demon may trigger when a package "bumps" another package. In the router specification, this is an unobservable event, i.e., there is no *mechanical* means provided for sensing it. It would be up to the developer to map it into some form that relied on or could be derived from observable events. There are several demons at this point in such form; the developer marks each for mapping.

Some of these steps are specific to the router development while others are more general. For instance, mapping of constraints, demons and derived relations are all steps likely to be found in any development. Hence, why not define a method which triggers itself at the start of the development and simply cycles through each of the Gist constructs placing a mapping goal

on each? During a development, the *recognition* of the constructs that have to be mapped is only part of the development problem. Just as important is the *order* in which general and domain specific mapping goals are attempted. For instance, if we tried to map the **SET_SWITCH** demon (step 6) before constraining its random trigger (step 4), we would have to unfold it at *every state transition point* within the program. Currently we rely on the user both to recognize specification freedoms that must be mapped and to order the mappings in an effective manner. In later chapters we discuss how this might be done by the machine.

4.2. The user as consultant/troubleshooter

Besides providing development organization, the user will likely be called on to supply information unavailable to Glitter and fill in missing portions of Glitter's catalogs. In this section, we will look at how this consultant/troubleshooter role manifests itself in a portion of the package router development. In particular, we will look in detail at the first few steps of the development presented in the appendixes, and discuss the user's and system's actions in generating them.

The first organizational step of the router development, as seen in the last section, is the removal of the derived relation **PACKAGES_EVER_AT_SOURCE**, or PEAS for short. In the transcript to follow, we will be dealing with this relation along with the demon **RELEASE_PACKAGE_INTO_NETWORK**. A description and pretty printed version of these two Gist constructs is given below.

When a new package arrives at the source of the package router[27] the demon RELEASE_PACKAGE_INTO_NETWORK is triggered (▸$_1$). The demon first checks to see if the previous package's destination was different than the current package's destination (▸$_2$). The previous package is computed (▸$_3$) by using the sequence of packages that has ever arrived at the source, PACKAGES_EVER_AT_SOURCE. The previous package is the package immediately before the current package in that sequence. If the destinations are different then the new package is delayed for some time before being released (▸$_4$). The final action of the demon is to transfer the package to the pipe leading to the first switch by updating its location property (▸$_5$).

The relation PACKAGES_EVER_AT_SOURCE specifies an ordered set (i.e., sequence) of packages. The set is the packages that have ever arrived at the source. The ordering is by their time of arrival.

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
▸1  trigger package.new:LOCATED_AT = the source
     response
      begin
▸2        if
▸3           (the package.previous ||
                 package.previous immediately before package.new
                   wrt PACKAGES_EVER_AT_SOURCE(*)
             ):DESTINATION ≠ package.new:DESTINATION
▸4         then invoke WAIT[];

▸5        update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
      end;

relation PACKAGES_EVER_AT_SOURCE(package_seq | sequence of package)
   definition package_seq =
     ({package || (package:LOCATED_AT = the source) asof everbefore}
        ordered temporally by start (package:LOCATED_AT = the source));
```

[27] The actual creation and placement of packages at the source is modeled by the spec but is part of the environment rather than the portion to be implementable.

Following is the annotated transcript of the first several steps of the router development dealing with the removal of PACKAGES_EVER_AT_SOURCE. For readability, all user input appears as ***bold italics***. While a *detailed* understanding of the transcript relies on information presented in subsequent chapters, we attempt here to provide an overall impression of the partnership roles.

```
>Glitter: FaithMode set faith mode

   Mode (trusting, cautious, critic): cautious
```

By placing the system in cautious mode, the user will have final say over the method selected to achieve a goal (see sections 2.2.1, 7).

```
>Glitter:  Post post a goal

   Goal: Remove
     What: PACKAGES_EVER_AT_SOURCE
     From context: defaulting to entire specification
```

Since the relation definition is global, the system fills in the context slot automatically.

```
 Initial candidate set formed:

    RemoveRelation
```

A single method, RemoveRelation, triggers on the remove goal. The method can be paraphrased as "if you want to remove a relation then remove all reference to it". Note that since only one method has triggered, it seems pointless to run any selection rules. However, as discussed in Chapter 7, the selection process not only orders competing methods, but may reject them as well. If this method was rejected, it would mean that either a method was missing from the catalog or that the goal was unachievable. .

```
Running selection rules...

     BurnedOutHulk fires

     *RemoveRelation1 being considered

        Is only one element of packages_ever_at_source
         being used? yes

     *RemoveRelation1 fires

     *RemoveRelation2 being considered

        Is packages_ever_at_source acting
         as a "temporary variable"? no

     *RemoveRelation2 rejected

Final candidate set formed:

   RemoveRelation(+4)
```

*Notation: rules prefixed by a * are tied to specific methods. Others are method independent and compute a weight by both examining the effects a method will have if chosen and how those effects will impact the current planning state as a whole, e.g., !will the method make it easier to achieve higher level pending goals?*

*The selection rules have given RemoveRelation a combined weight of 4 (well above the selection threshold) on the following grounds: in general, a good way to get rid of a defined object is to remove all reliance on it (BurnedOutHulk); if only one element (first, last, nth) of a sequence is ever needed then it is likely that the sequence can be replaced with a single value (*RemoveRelation1).*

Once the competing methods have been ordered, a selection is ready to be made. If in trusting mode, the system would choose the highest ranked method for invocation. If in cautious mode, the system returns control to the user for his ok (the case here). At this point the user may examine the individual selection rules that fired and their effect on the method ordering as a whole. With this information he or she may choose to either invoke another competing method, step in and define a method on the fly, or backtrack to some previous planning state. In each of these cases, the system will record the context for later (human) analysis: it is likely that they are symptoms of a missing piece of development knowledge. In this case the user chooses to confirm the system's choice.

```
>PlanSpace:  Choose choose a method
       Which method (<cr> for system's choice): <cr> RemoveRelation

RemoveRelation invoked

Goal posted: Remove reference to packages_ever_at_source
               from the specification
```

The method chosen, RemoveRelation, attempts to remove a relation by removing all references to it. In this case, there is only one reference to packages_ever_at_source, the one found in the derived object package.previous (▸$_3$). The corresponding goal is posted.

```
 Initial candidate set formed:

   BabyWithBathWater-1
   BabyWithBathWater-2
   BabyWithBathWater-3
   BabyWithBathWater-4
   MegaMove-1
   MegaMove-2
   PositionalMegaMove-1
   PositionalMegaMove-2
   RemoveByObjectizingContext-1
   RemoveByObjectizingContext-2
```

Hyphenated names indicate separate instantiations of the same method. In all cases above, each instantiation represents a different context from which to view the problem. For instance, the method BabyWithBathWater attempts to remove X by removing some context Y that contains X; instantiations 1-4 above represent different bindings of Y, i.e., the predicate of the derived object, the derived object itself, the conditional that uses the derived object, and the demon that contains the conditional.

```
Running selection rules...

   Fillin fires
   *BabyWithBathWater1 fires
   *BabyWithBathWater2 fires
   *BabyWithBathWater3 fires
   *MegaMove1 fires
   *PositionalMegaMove fires
   RemoveRef1 fires
   RemoveRef2 fires
   RemoveRef3 fires

Final candidate set formed:

   1. MegaMove-2(+3)
   2. MegaMove-1(+3)
   3. PositionalMegaMove-2(+2)
   4. PositionalMegaMove-1(+2)
   5. {RemoveByObjectizingContext-1(-),
       RemoveByObjectizingContext-2(-)}
```

All four instantiations of BabyWithBathWater have been rejected by the selection rules. The two RemoveByObjectizingContext methods have no selection information and so are placed last (see section 7.2). If in trusting mode, the system would choose MegaMove-2: although it has the same weight as MegaMove-1, it has been explicitly ordered as first by RemoveRef1. In cautious mode, control is returned to the user.

```
>PlanSpace:  Choose choose a method
       Which method (<cr> for system's choice): <cr>

 MegaMove-2 invoked
   ...
```

We will summarize the next two steps of the development without providing transcript details. The planning structure that follows this paragraph is used as reference. The MegaMove method posts two subgoals: ▸$_1$ fold (isolate) the derived object into a derived relation R; ▸$_3$ then unfold (maintain) R at locations where it might more easily be replaced. Before folding, an effort is made to remove any references to locally scoped variables (▸$_2$), in this case the variable *package.new* declared by the demon. We will resume the development at this point. Note that a) the user has had to post only a single goal up to this point (user posted goals are marked with a " + "), and b) no changes have yet been made to the program; the action has all

taken place in the planning space. The planning structure at this point is as follows (see the PrPlan command, 2.3.1.2):

```
+ Remove packages_ever_at_source from spec
                          ...using RemoveRelation

     Remove reference to packages_ever_at_source from spec
                          ...using MegaMove

▸1      Isolate derived-object package.previous into R
                          ...using FoldGenericIntoRelation

▸2        Globalize derived-object package.previous
                          ...using GlobalizeDerivedObject

               * Reformulate package.new as global expression
                          ...to be selected

          Apply fold-into-relation(package.previous, R)
                          ...primitive transformation

▸3      MaintainIncrementally R
                          ...to be selected
```

We resume the development at the currently active goal (marked with a "*"):

```
...
Goal posted: Reformulate package.new as global expression

Initial candidate set formed:

  ReformulateLocalAsFirst
  ReformulateLocalAsLast
```

The two competing methods capture the following development technique: "if you are trying to reformulate a local variable as a global expression then look for a defined sequence of the same type (i.e., package) as the variable; it may be the case that the object (package) referenced by the local variable is also contained within the sequence." The two methods find such a sequence (PACKAGES_EVER_AT_SOURCE) and suggest trying the first and last element respectively.

```
Running selection rules...

   ReformLoc1 fires
```

ReformLoc1 orders the two competing methods. It can be paraphrased as "if you are trying to decide between reformulating a local reference as the first element of a sequence or the last element of a sequence AND the sequence is ordered temporally then try reformulating it as the last element".

```
Final candidate set formed:

   1. ReformulateLocalAsLast(+2)
   2. ReformulateLocalAsFirst(+2)


>PlanSpace:  Choose choose a method
        Which method (<cr> for system's choice): <cr>

  ReformulateLocalAsLast invoked

  Goal posted: Reformulate package.new
                                 as last(PACKAGES_EVER_AT_SOURCE(*))

  Empty initial candidate set
```

Assume that there is currently no method in the catalog that can reformulate the reference to a local variable into the retrieval of the last element of a sequence. At this point the user may either a) give up on this line of pursuit and backup to some previous state to try an alternative path or b) continue this path and supply the next step himself. In either case, the system will record the event as a possible symptom of missing knowledge: in the former case a bad selection was made; in the latter a method is missing. The user chooses to continue by reasoning as follows:

1) every package shows up in PACKAGES_EVER_AT_SOURCE,
2) because the ordering is oldest first, the newest package is last,
3) package.new is the newest package,
4) therefore package.new and the last element of packages_ever_at_source reference the same object.

The user uses a manual editing step to change the program. There are two points to make here. First, even though the system was unable to carry through on its general strategy of removing package.new, it did get to a low enough level to allow the user to supply the last necessary reasoning step. Second, the need for the user to intervene, as well as the needed verification of the manual step,

has been recorded for post-development analysis. If a new method can be added (by a human analyst) that generalizes the manual step then the mistake won't happen again, i.e., the system can supply the replacement step in the future[28].

```
>Glitter: Post post a goal

     Goal: Manual Replace
        Replace what: package.new
        With what: last(PACKAGES_EVER_AT_SOURCE(*))

   Recording step in development log as manual edit to be verified

   Recording step in development log as possible missing method
```

Note that this is the first operation actually carried out on the program. Without motivation, i.e., the planning structure sitting above it, it appears as a somewhat lucky or Eureka step: replace one expression with another. With the subgoal hierarchy intact, its true purpose is illuminated: prepare the derived-object for folding (so that it can be maintained so that the reference can be removed ...).

[28] The needed method is one that reformulates an arbitrary expression as another, with an applicability condition that the two expressions can be shown to be equivalent.

```
   Goal achieved: Reformulate package.new
                              as last(PACKAGES_EVER_AT_SOURCE(*))
   Goal achieved: Reformulate package.new as global expression
   Goal achieved: Globalize derived-object
   Goal posted: Apply fold_into_relation(derived-object, R)

New relation being defined:


relation  ?relname (?argname | package)
 definition ?argname =
  (a package.previous ||
   package.previous immediately< last(PACKAGES_EVER_AT_SOURCE(*))
          wrt PACKAGES_EVER_AT_SOURCE(*));


    What do you wish to call
      the new relation (?relname): previous_package

    What do you wish to call the argument (?argname): prev_package
```

When the system needs a name for a new item, it asks the user to supply it; user supplied names lead to much more readable programs.

```
   Goal achieved: Apply fold_into_relation(derived_object,
                                           previous_package)
   Goal posted: MaintainIncrementally previous_package

    Initial candidate set formed
     ...
```

The next step in the development would be the incremental maintenance of the newly defined relation PREVIOUS_PACKAGE. The overall planning state is as follows:

```
    + Remove packages_ever_at_source from spec
                              ...using RemoveRelation

        Remove reference to packages_ever_at_source from spec
                              ...using MegaMove

         ! Isolate derived-object package.previous
                              ...using FoldGenericIntoRelation

           !  Globalize derived-object package.previous
                              ...using GlobalizeDerivedObject

             ! Reformulate package.new as global expression
                              ...using manual replace

           !  Apply fold-into-relation(package.previous, R)
                              ...primitive transformation

         * MaintainIncrementally derived relation previous_package
```

Notation: achieved goals are marked with a "!". The currently active goal has become the maintenance of previous_package. The program state is now as follows:

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      if PREVIOUS_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
       then invoke WAIT[];
      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;

relation PACKAGES_EVER_AT_SOURCE(package_seq | sequence of package)
   definition package_seq =
     ({package || (package:LOCATED_AT = the source) asof everbefore}
        ordered temporally by start (package:LOCATED_AT = the source));

relation PREVIOUS_PACKAGE(prev_package | package)
   definition prev_package =
    (a package.previous ||
     package.previous immediately < last(PACKAGES_EVER_AT_SOURCE(*))
           wrt PACKAGES_EVER_AT_SOURCE(*));
```

The remainder of the removal of PACKAGES_EVER_AT_SOURCE along with the remainder of the package router development is presented in Appendix C.

4.3. Summary

In this chapter, we have provided a general view of the role of both user and system during development. Regarding development automation, the system performed 5 out of the 7 planning steps in the transcript presented above. The continuation of the removal of the relation PACKAGES_EVER_AT_SOURCE results in 34 planning steps out of which 12 are actual program transformations. The user was required to provide 2 of these 34 steps: the posting of the Remove goal, which triggered the problem solving; the manual step, which filled in a missing method. On the whole, the system was able to perform 146 out of the 159 steps in the router development, i.e., the user was required to post 13 goals during the development. The user was also called on to supply formal reasoning necessary in both method selection and method application. While part of this can be automated in the near term by incorporating sophisticated state of the art flow analyzers, much of it remains in the realm of program verification. This is particularly messy when constrainted non-determinism, demons. parallelism and inference must be considered, all freedoms provided by Gist.

In following chapters, we discuss in detail the underlying knowledge necessary to produce the development transcript above, and the partnership that underlies it.

Chapter 5
A Development Vocabulary

In the Glitter model, the user is responsible for guiding a development by providing development goals and the system is responsible for providing the steps which achieve those goals. In this chapter we will look at the Glitter representation of development goals, what development concepts they must represent, how well they do and how they may be extended. At the end of the chapter we present a brief summary of the Glitter goal descriptors (Appendix E provides the detailed description), and below a summary of the important points made in the remainder of the chapter:

Separation of what from how.

Glitter provides a general goal language for stating domain independent development problems. We have chosen to separate the description of a particular development problem -- the *goal* -- from the particular techniques that can be used to solve it -- the *methods*. As we shall see in later sections, this gives us several useful capabilities including goal monitoring, development structuring and knowledge additivity.

Goal representation as a parameterized structure with an explicit achievement checker.

To make goals useful over a broad range of problems, a typed set of context setting parameters is defined for each goal. In systems where a goal is tied directly to methods for achieving it, the goal derives its semantics from the connected methods. We have unlinked goals and methods, and hence must define some alternate form of explicit goal semantics. We have chosen to attach to each goal a Lisp function which monitors the goal's achievement as the development progresses.

Our major development concern is mapping Gist specification freedoms.

Gist provides information, operation and efficiency freedoms to the specification writer. The concern is mapping these freedoms into a form directly computable in a target language.

Other development concerns include showing applicability conditions hold, jittering, simplification and organization.

Each of these must be addressed in the problem solving approach we have taken to development.

The development vocabulary should provide a certain amount of robustness to the user.

Portions of the development that are independent should not be constrained to an arbitrary order of attempt. Arbitrary constraints on the way a problem can be stated should be avoided. The user may wish to sometimes work at the highest problem description level and sometimes at the lowest. In the latter case in particular, the user may wish to bypass the explicit statement of a goal and directly name a method or transformation to apply (While the system allows this, it bypasses the planning documentation needed for later maintenance.).

Goals specific to the particular specification under development will likely crop up.

While providing a general development vocabulary tied to no particular domain or spec, Glitter has a mechanism for defining *user goals* on the fly as a development progresses.

5.1. Goal Representation

To completely remove the user from the development process, Glitter would have to able to achieve a goal such as "reach a state where only target language constructs remain". A more complete form of this goal would be "reach a state where only target language constructs remain and the code is optimal". While a Glitter development implicitly embodies these goals, their achievement in a totally automatic way is beyond the reach of the system (see Chapter 4 for a discussion of the user and system roles in the Glitter model). More typical in the partnership model is the user's recognition that certain distinguishable states must be achieved on the way to a complete development. These may be viewed as *island states* along the development path which the user and system must link together. Examples include reaching states where all derived relations are explicitly computed or all non-determinism has been removed. The user's role is to guide the development by his choice of development goals. We can classify these into several types:

- The achievement of some goal state. Here, the user wants to reach a state where some possibly abstract feature or pattern is present (Feather [Feather 82b] notes a similar need for stating goal patterns in a fold/unfold transformation system.).

Reach a state where these two expressions are equivalent.

Reach a state where this expression matches that pattern.

Note that a goal state may be described in varying degrees of abstractness; we will have more to say on this below.

- The completion of some abstract action. This differs from a goal state in that the process as opposed to the result is specified.

  Break this expression into simpler cases.

  Map this construct.

  Swap these two statements.

  Any one of several methods may produce the necessary case break-out, mapping or swapping.

- The request for a specific technique to be employed. This differs from an abstract action in that a single method (out of possibly many) is selected to achieve the goal.

  Use incremental maintenance to map this relation.

  Use unfolding to map this demon.

We define a *goal descriptor* as our formal notation for stating development goals of the above type. A goal descriptor consists of a unique name[29], a set of typed slots and a predicate which tests whether the goal has been achieved. The user states a particular goal descriptor by use of the Post command (see section 2.3.3.2). For instance, there exists a goal descriptor *Map* which takes a single argument, a Gist specification freedom. The user would post the Map goal in the following manner:

[29] We have chosen names which connote action for stylistic reasons.

```
>PlanSpace: Post post a goal

        Goal: Map
           Freedom: <Gist construct>

        ...
>PlanSpace:
```

The system prompts for each argument of the specified goal and does type checking on the value supplied by the user.

For purposes of presentation, we will use a more concise and complete form of goal notation:

GoalName(Arg_1|***argtype***, ..., Arg_n|***argtype***)

Achievement condition: Predicate stating goal achievement.

Thus our Map goal becomes

Map(Arg_1|***freedom***)

Achievement condition: Arg_1 has been either removed or operationalized

The semantics of a goal descriptor are given by its achievement condition. Because we have chosen to represent all user development goals through the goal descriptor notation, the predicate defining achievement may be called on to monitor either a pattern or feature becoming manifest, or some action completing. Note that in the former case a development goal has a life of its own, independent of any method application. That is, the completion of a method indexed to a goal does not automatically mark the goal achieved; a goal is achieved only when its achievement condition becomes true. This allows us the flexibility of incremental achievement by the combination of several method applications.

When the current goal is marked as achieved, the system does one of the following:

- If the goal has a brother which has not been achieved and is next in line to be posted, then it is posted.

- If the goal has no brother waiting then the method it is part of is marked as complete. As noted above, this does not necessarily mean that the supergoal that the method was attached to has been achieved. If it has then it is in turn marked as achieved and the process repeats. If it hasn't then it is reposted as if it were a waiting brother goal.

- If the achieved goal is a user posted top level goal then control reverts to the user.

Section 6.2 provides a more detailed discussion of goal/method control issues.

Given a goal representation, we next look at the type of Gist development problems that need to be represented.

5.2. Characterization of the TI Development Space

The set of goal descriptors defines the type of development problems that the system can work on (Appendix E lists the current descriptors). In our development of Gist specifications, we are interested in the initial or high-level development of the specification. In this section we will characterize Glitter's development concerns.

5.2.1. Mapping Specification Freedoms

Gist provides a certain set of *specification freedoms*: constructs that allow behavior to be described without referencing implementation details (Chapter 3 explains Gist freedoms in more detail). The major concern of a Glitter development is the *mapping* of these freedoms into implementations. The development in Appendix B contains examples of most of the necessary mappings, including those on demons, derived relations, temporal reference, constraints and non-determinism. In general, there are three freedoms that we must deal with, *operation*, *information*, and *efficiency* (see [Balzer et al. 82] for related discussion).

5.2.1.1. Mapping Information Freedoms

In a Gist specification, what information is necessary may be specified without describing how it is to be computed. The mapping choice can be one of two general strategies:

- *Maintain* the necessary information explicitly. That is, store its initial value and incrementally maintain it as the program executes (see [Paige & Koenig 82] for related discussion).

- *Unfold* the necessary code to rederive the information at each place that makes use of the information.

Section 7.3 discusses the criteria used for choosing among these two.

5.2.1.2. Mapping Operation Freedoms

A Gist specification may contain non-deterministic choice points, which allow the specifier to indicate equally acceptable alternatives in a straightforward fashion. The integral partner of non-determinism is constraints, which allow the specifier to declaritively state the limitations of the system. A specification denotes a set of behaviors governed by its constrained non-determinism. There are two basic strategies for dealing with constrained non-determinism:

- Backtrack by *Unfolding* a constraint at each place in the program where it might be violated. If the constraint is violated then control backs up to the most recent choice point and a new choice is made.

- Predict which choices will lead to violation and don't choose them, i.e. generate only ones that satisfy all constraints. A general technique is to *Map* the constraint into a demon which watches for potential violations and takes appropriate action to insure they don't occur. We use this type of control in the package router development. A related technique is to change a backtracking control into a predictive control by moving constraints into choice points. The development in [Balzer 81] uses this strategy (see also [Tappel 80]).

Section 7.3 discusses the criteria needed for choosing among the two.

5.2.1.3. Mapping Efficiency Freedoms

A Gist specification need not and should not contain efficiency concerns. The efficient ordering of operations, the elimination of unneeded (e.g., redundant, unused) information or operations, the sharing of information or computations among program parts, is not the concern of the specifier. Mapping these freedoms is the concern of the developer. Glitter supports three basic efficiency mappings:

- Efficient ordering of operations by making non-deterministic control sequential and resequencing already sequential actions.

- Removal of unneeded information or operation structures.

- Sharing of like parts among compound structures by consolidation and factoring.

These clearly do not cover all efficiency goals. However, we are only interested in the type of

optimizations that can be made during the initial mapping from spec to algorithm. While we believe ordering, removing and sharing are all efficiency goals useful for indexing optimization methods at the algorithmic level as well, the type of optimizations that deal with making algorithms more efficient ([Standish et al 76, Kibler 78, Bentley 81, Rutter 77] lie,for the most part, out of our area of interest.

5.2.2. Applicability Conditions

Most Glitter methods rely on some program or domain property to hold before they can successfully complete. A large portion of the development may be committed to showing these applicability conditions hold currently or making them hold if they don't. The DEDALUS system [Manna & Waldinger 79] automates this process in its limited problem domain through the use of an automatic subgoaling mechanism. Barstow [Barstow 79b] further speculates on the automation of condition proving in a knowledge-based system. Our view is that the freedoms afforded by Gist make the construction of a general purpose property prover an unlikely prospect in the foreseeable future. In any case, Glitter currently has no automatic means of proving the applicability conditions of methods, hence it becomes the purview of the partnership.

5.2.3. Jittering

We have defined jittering to be the process of getting the current program state to match the state required by some development method. Let's look at some ways we might get around jittering. First, we could attempt to define a Gist canonical form. It is clear that some jittering will be required even when the program and pattern are in this form. For instance, commutative and associative operators will always be necessary. In the example below, we are given a canonical form of a conjunction with terms alphabetized. The canonical pattern that we wish to match requires that the current expression be rewritten in a non-canonical form to match the pattern.

pattern: (a and c)

current: (a and b and c)

The simple jittering necessary above could be handled by an automatic mechanism (see for

instance PADDLE's table driven mechanism, [Wile 81a]). However, the inclusion of defined and derived constructs in Gist makes any general automatic canonicalization process infeasible. For instance, the canonical form would require that all functional structure be flattened. Because the user is in the loop, such destruction of the basic form of the program is unacceptable.

A second approach would be to anticipate all of the contextual forms. That is, each transformation would be broken out into k new transformations where k represents the number of different ways the transformation can match. The magnitude of k makes this intractable in general.

In Glitter, jittering is made part of the overall problem solving process. The automation of jittering allows the program (parse tree) to remain in a non-canonical (but normalized) form and the method writer to be unconcerned with specializing his methods. The general jittering goals follow.

- *Reformulation*. In some sense, reformulation is a form of local canonicalization. We choose some syntactic pattern as our canonical form and attempt to get the program to match it.

- *Equivalency*. Many times, sharing of structures requires that one or more parts be equivalent.

- *Positioning*. A method may require that one statement be in a certain relational position with regards to one or more other statements.

- *Information Movement*. Much of the mapping process involves the movement of information around the specification. One part of this process is pulling an expression out of a local context and making it a global structure (*isolation*). This may require that steps be taken to trim any ties the expression has to the local context. Another part of the movement process is moving a global structure to a local context where it can be further optimized.

5.2.4. Simplification

As a Glitter development progresses, the intermediate forms of the program tend to become messy and hard to read. Part of this problem can be handled by the reorganization of the program (see the next section). A major part, however, has to do with the movement of code from one context to another. This may be a global to local movement or a local to local

movement. In any case, the newly introduced code, in combination with its surrounding environment, can often be simplified using low level rewrite rules. Typical rules include

(and ... ***false*** ...) ⇒ ***false***
(or ... ***true*** ...) ⇒ ***true***
(not (not P)) ⇒ P
(if P then A else A) ⇒ A

There are two things worth noting about the simplification process in Glitter:

1. The simplification process is below the planning level. That is, the individual steps involved in simplification are not made part of the development history.

2. The firing of the simplification rules is carried out by the system in a non-supervised fashion. Hence, simplification rules should not a) call on the user for assistance, or b) rely on costly reasoning. For example, removing an unneeded relation from the specification (see step 2.1 in Appendix B) could be viewed as a simplification step. However, the resulting steps necessary to carry out this task both rely on the user and are costly.

Only the cheapest and simplest of rules should be used in simplification: rules that involve planning will be not show up in the final history; rules that involve costly reasoning will run independent of the selection process and will not be under user control.

5.2.5. Pragmatics

Part of the development process involves practical organizational issues. For instance, breaking a complex expression into a number of simpler cases may facilitate further development. Regrouping a set of scattered objects may make the specification easier to read. Ordering certain mappings may have a profound effect on the ease of development. Generally, each of these steps address not program efficiency but problem solving efficiency, whether it be by human or machine.

5.3. Coverage

In the previous section we looked at the type of development problems that arise in developing a Gist specification. The coverage of the development process by the current Glitter goals is a function of both our experience in developing Gist programs and our ability to generalize from that experience. Because our experience base is small, it relies more strongly on the latter.

5.3.1. Mapping of Specification Freedoms

Our results show that only a small number of development strategies exist at the Gist mapping level (see also [London & Feather 82]), strategies that can be indexed by a correspondingly small number of goals. The difficult part of the mapping process is one of recognizing the interdependencies among mapping decisions and organizing the development accordingly. Elaine Kant addresses some of these issues in her LIBRA system [Kant 79]; much work remains in modeling mapping organizations in Glitter. Currently, the user is responsible for mapping organization, and little help is provided in the way of goals. We believe progress will come in this area by studying a much wider range of developments.

5.3.2. Applicability Conditions

As discussed previously, Glitter provides no general purpose property prover. This means that showing that a property holds becomes just another type of interactive problem solving. Unfortunately, the current Glitter vocabulary for dealing with applicability conditions is weak. The single descriptor *Show* is used to handle all property proving tasks. Bulnes-Rozas [Bulnes-Rozas 79] demonstrates that a richer set of goal descriptors is possible in his interactive GOAL system. Further work is needed along the GOAL lines to provide a better vocabulary for cooperative property proving in Glitter.

5.3.3. Jittering

The set of Glitter goal descriptors contains a subset of what we might label *jittering goals*, goals that are not achieved for their ends but for as a means of achieving other goals. In some sense, the jittering goals are used to paraphrase the left hand side of the transformations (see section 6.2.1). In a GPS system, they would act as the difference description produced by comparing the left hand side against program code. The definition of the current jittering goals was in fact influenced by some earlier work in jittering using a GPS approach [Fickas 80]. Section 6.1.4 in the next chapter discusses the evolution of jittering in Glitter in more detail.

5.3.4. Simplification

A number of researchers have looked at means of incorporating simplification rules into the programming process: [Standish et al 76] provides a general catalog; Kibler [Kibler et al. 77] and Neighbors [Neighbors 80] discuss ways to make their execution more efficient. In Glitter, the simplification process is invoked by the posting of the *Simplify* goal, which causes the rules in the *simplification subcatalog* (see F.16) to be run until a quiescent state is reached, i.e., none of the rules fire. The argument of *Simplify* is the context in which simplification is to be carried out.

Glitter's current view of simplification is that of a *user invoked* process that will likely be needed at frequent intervals during development. Their are clearly other views. For instance, PADDLE [Wile 81a] automatically invokes its simplification rules after <u>every</u> *non-jittering* change to the program. Simplification cannot be done directly after a jittering step for this will likely undo the effects of jittering. For instance, one of PADDLE's simplification rules is

TrueCond: <u>if</u> ***true*** <u>then</u> ***action*** ⇒ ***action***

However, another one of PADDLE's rules is

EmbedInCond: ***action*** ⇒ <u>if</u> ***true*** <u>then</u> ***action***

We clearly do not want to simplify after applying EmbedInCond, assuming that its application is part of some larger problem solving context that relies on its effect. PADDLE's solution is to provide a special catalog of jittering rules which turn off simplification after their application. EmbedInCond is one of these rules. In Glitter, there arise questions on how long the simplification should be shut off (or equivalently, what event signals its reactivation) and whether, in general, all development methods can be split cleanly into jittering and non-jittering classes.

Another approach would be to rely on the method writer to include *Simplify* goals in methods where they would likely have a payoff and would not interfere with the surrounding problem solving context (this has somewhat the same flavor of Kibler's and Neighbor's work). This provides some degree of simplification automation, and we are considering it as an improvement over the current manual process.

5.3.5. Pragmatics

The system supports two basic program arrangement goals: 1) split a compound or defined structure into sub-pieces, 2) group sub-pieces into a compound or defined structure. The first indexes divide-and-conquer strategies, the second is a simplification ends in itself. Note that commutative, distributive and associative operations achieve these goals at the lowest level.

5.4. Robustness

We view the robustness of the system as the freedom given the user in carrying out the development task. There are several aspects to developmental freedom:

1. A user may choose to work at somewhere below the highest level goals provided by the system. For instance, the router development in [London & Feather 82] was organized around isolation and incremental maintenance, two levels below the top goal in our development (see appendix B). In general. the user should be able to move among all descriptive levels, from stating abstract mapping goals to naming particular transformations to be applied.

2. A user can choose one of possibly many orderings of mapping steps. For instance, in mapping away the demonic structure in section 6 of Appendix C, the user was free to choose the order in which to map each demon.

3. There may be two or more equivalent ways of viewing a problem. Given that the goal language supports multiple descriptions, we need a means of mapping each onto our known development techniques. Both Mark [Mark 80] and Mostow [Mostow 81] discuss related problems. We will look at the problem in Glitter in more detail below.

The first two items are provided by the system: the user is given the freedom to choose the problem solving level he wishes to work at, and is allowed (relied on) to organize the development as seen fit. The last item presents more of a problem. We can characterize the problem as such: we have a technique for solving a particular development goal and k ways of stating the goal. We must find a way to map each onto the technique. A concrete example may be helpful here. In section B.4 of the router development, the user turns his attention towards implementing a switch setting policy. There are two constructs involved: a demon, SET_SWITCH, which triggers at random times and sets a switch to one of its output ports, non-deterministically; a constraint, DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE, which limits the non-determinism of SET_SWITCH. As described in Chapter 3, constraints and non-determinism go hand in hand. Hence, there are two different ways the user can describe

the switch setting problem: 1) a mapping of the constraint **DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE** (the goal used in the development in B.4), or 2) a mapping of the non-determinism in the demon **SET_SWITCH**. Either goal should eventually lead to the application of a demon consolidation method.

It is important to separate implicit and explicit descriptive power here. The goal language provides the explicit robustness of the system. It determines what problems can be stated. In our example above, the user was able to state both types of mapping goals. Contrast this with a language which allowed only one to be used. This forced viewpoint could be carried out by limiting the arguments of the map goal to certain constructs, e.g., constraints but not non-determinism.

Given that a problem can be described in different ways, it is up to the system to find methods for mapping those descriptions onto techniques for achieving them. The language may supply us with full descriptive power, but without this implicit mapping capability it is rather hollow. However, it is difficult to define *a priori* all of the ways a particular problem can be stated. We have relied on experience with different developers and developments to build the implicit mapping base of methods. Because this experience has been small, we view the current mapping base as incomplete. This is offset by 1) the ability of Glitter to highlight potentially missing knowledge, and 2) the ease with which such knowledge can be added to the system. Both of these capabilities are discussed in more detail in Chapter 6.

5.5. Extending the Language

Goals and the methods for achieving them are explicitly coupled. This makes the addition of a new goal to the language a potentially difficult task[30] requiring its integration into the method catalog. An example may help to illustrate. Suppose that we wish to add the goal *Extract* to the language:

Extract(Inner|***construct***, Outer|***construct***)

Achievement Condition: Inner, a subcomponent of Outer, replaces Outer.

Extract is achieved by destructively replacing a compound structure with one of its

[30] We mean here the *useful* addition of a goal, i.e., the integration of the goal into the rest of the problem solving system. Simply defining a new goal is straightforward.

components. For example, we can extract a statement $\blacktriangleright_2$ from a begin/end block $\blacktriangleright_1$ by replacing $\blacktriangleright_1$ with $\blacktriangleright_2$:

$\blacktriangleright_1$ begin
  update x to 0;
$\blacktriangleright_2$ invoke A(x)
end

Stated in goal form:

Extract $\blacktriangleright_2$ from $\blacktriangleright_1$

To add the *Extract* goal, we must do the following:

1. Find how *Extract* interacts with existing goals. Is it a specialization of some other goal? In this case the component to be extracted can be viewed as the pattern argument of a *Reformulate* goal and the compound structure as the current expression argument (see section E.11 for a more detailed description of this goal).

   Reformulate $\blacktriangleright_1$ as $\blacktriangleright_2$

   We must define a method for mapping specific types of *Reformulate* goals into *Extract* goals. A method of the following type will suffice (Glitter method notation is introduced in chapter 6; this method can be paraphrased as "if you are trying to reformulate X as Y, and the pattern Y is found somewhere within X then call the match M and try extracting it from X".):

```
| Method  ReformulateByExtracting                                  |

          Goal:   Reformulate X as Y
          Filter:   a) pattern-match[Y, M, X]
          Action:   1) Extract M from X

        [If Y is found within X then try extracting it.]
| End Method                                                        |
```

2. Find any existing methods triggered by the more general goal *Reformulate* which achieve the more specific goal *Extract*. Specialize their triggers accordingly. Their are none such in the current catalog.

3. Analyze the various types of extraction that can occur and define corresponding methods. A good choice is to start with syntactic types: a method for extracting from begin/end blocks, a method for extracting from conditionals, etc. For instance, we might define the following methods:

   If you want to extract the action portion of a conditional then show that the predicate portion is always true.

   If you want to extract a statement from a begin/end block then get rid of all the other statements and simplify.

The actual Glitter form of the above two methods is given below.

```
| Method  ExtractConditionalAction                                  |

          Goal: Extract A|action-statement from C|conditional
          Action:  1) Reformulate condition-predicate as true
                   2) Simplify C

      [if true then A ⇒ A]
| End Method                                                        |
```

```
| Method  ExtractStatementFromBlock                                 |

          Goal: Extract A|action-statement from B|begin/end
          Action:  1) forall immediate-component-of[S, B]
                        suchthat S ≠ A
                          do Remove S from B
                   2) Simplify B

      [begin S end ⇒ S]
| End Method                                                        |
```

5.5.1. User Goals

The addition of the *Extract* goal is an example of the addition of a general, development-independent descriptor. We would expect this goal to be useful in many problem domains. However, it is likely that any particular development will involve goals specific to that development or application. The Glitter system provides the user with a facility for defining development-dependent goals which remain defined throughout a particular development. User defined goals take the same form as Glitter goals:

- *Goal name*. Must not conflict with existing goal name.
- *Typed slots*. Zero or more.
- *Achievement Condition*. Either user provided function or user controlled (default). Primitive AC-building Glitter functions are available to the user.

Because user goals are not indexed into the method catalog, they serve only as a development structuring and documentation aid. They allow the user to tie application-related steps together under ad hoc methods (see [Wile 81a] for a similar capability). We will look at an example taken from the router development.

Development context: although broken out separately, sections B.1, B.2 and B.3 of the router development in Appendix B can be viewed as sub-goals of a single higher level goal: *optimize the use of package history*. Currently this must be viewed as a development-dependent goal; further experience may lead us to define a general, development-independent Glitter descriptor for optimizing historical reference of this type.

To post a development-dependent goal, the user employs the same mechanism as posting a built-in goal (see section 2.3.3.2):

```
>PlanSpace: Post post a goal
          Goal: OptimizePackageHistory     user goal? yes
             Arg1: <cr> ...no arguments defined
             AC: <cr> ...user determined
>PlanSpace:
```

The user enters the goal name and the system, unable to recognize the goal name, asks the user to confirm that it is a user goal (it could also be a typing error). The system then prompts for each of the goals arguments. Here, the user has chosen to forego any arguments. Finally, the system prompts for the name of a function which will check the achievement of the goal. The default, and in this case the user's choice, is to allow the goal to remain active until the user explicitly marks it as achieved (see section 2.3.3.1).

We would expect this goal to be posted as the first step of the development and marked as achieved at the completion of the steps in section 3.

5.6. Direct Invocation

The goal descriptors form the link between problem and methods. Glitter advocates their use as a means of separating the concerns of how to solve a development problem with that of stating what the problem is. However it has become clear that there will arise cases where the user wishes to name the method to employ directly, foregoing the explicit statement of the corresponding goal. This can be viewed as part of the robustness supplied by the system. There are three mechanisms that deal with direct invocation: a descriptor, *Apply*, for invoking a transformation; a descriptor, *Manual*, for doing unsupervised editing of the Gist program; a descriptor, *Use*, for invoking a method. We will look at each in turn.

5.6.1. Applying transformations

Apply(T|***transformation-procedure***)

Achievement Condition: Procedure T is invoked and completes.

Program transformations are represented as procedures in Glitter. They are invoked using the Apply goal. Each procedure incorporates the necessary applicability conditions to guarantee a correctness preserving change. Compare this with manual changes described next.

5.6.2. Manually editing the program

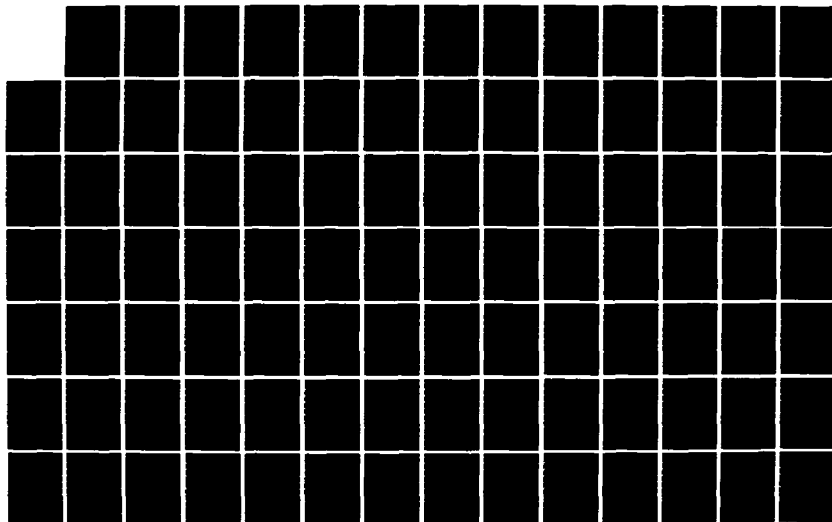
Manual(M|***primitive-edit-operation***)

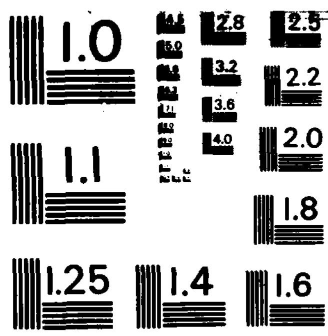
Achievement Condition: Completion of M.

There are several reasons why a development system might provide the user with primitive program-editing operations which bypass the normal method catalog: 1) given a posted goal G, no methods exist for achieving G, or 2) the user is unable to find the right method in the catalog. The first is especially likely during initial catalog construction and as long as the system lacks a powerful theorem prover. The second is especially likely if the catalog is large and unindexed. Since the Glitter system provides a solution to the second problem, we will be concerned only with the first.

Given no method for achieving a posted goal, we allow the user to edit the program directly, using a set of built-in primitives. While the validity of this editing process is assumed to be correct, there is no check made during development. It is left to some other analyst to verify the editing steps (and possibly suggest the construction of new methods) after the development is complete (see section 7.2.1.1). We will look at an example from the router development.

Development context: step 1.6 posts a goal of reformulating the variable *package.new*, an object defined locally, as a retrieval of the last element of the sequence PACKAGES_EVER_AT_SOURCE. No methods exist for achieving this reformulation. In step 1.7 the user manually replaces *package.new* with last(PACKAGES_EVER_AT_SOURCE(*)), thus satisfying the reformulation goal. The reasoning necessary to make the replacement is non-trivial: since 1) all packages are


AD-A139 860 AUTOMATING THE TRANSFORMATIONAL DEVELOPMENT OF SOFTWARE 2/3
VOLUME 1(U) UNIVERSITY OF SOUTHERN CALIFORNIA MARINA
DEL REY INFORMATION S.. S F FICKAS MAR 83
UNCLASSIFIED ISI/RR-83-108 NSF-MCS79-18792 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

being "kept" in the sequence PACKAGES_EVER_AT_SOURCE, and 2) the ordering is temporally descending (oldest first), and 3) the newest (last in sequence) package is the last one to be located at the source station, and 4) *package.new* is the last package to be located at the source station then 5) *package.new* and the last package of the sequence are equivalent objects.

The complexity of the reasoning process above is not uncommon in a development. Without a certain level of specificity, we cannot be certain that the user will continually be able to generate this type of analysis in its entirety. In particular, the system's role must be to generate enough of the motivation to allow the user to finish up the task. In this case, that meant 1) pointing out that the variable *package.new* should be reformulated to avoid carrying it along as baggage in the forthcoming isolate step, and 2) finding a likely candidate expression to replace it with. It is left to the user to confirm that the new expression on PACKAGES_EVER_AT_SOURCE is equivalent.

5.6.3. Invoking a method

Use(M|***method-name***)

Achievement Condition: M's triggering goal is achieved

This descriptor allows the user to request that a method be invoked directly, bypassing the normal Glitter goal-posting/problem-solving process. When a user employs this goal, several interesting things happen:

- The normal process of forming the candidate set and selecting among the members (see 7.2) is eliminated. This càn produce a substantial speed-up.
- The development documentation is weakened. The normal documentation provides a) goals, b) the competing set of methods for achieving individual goals, and c) the rationale for selecting one method over the others. Both b and c are lost.
- A disregard for the system's development knowledge is shown. The system's knowledge is contained in the method and selection rule catalog. The user is saying that he knows what methods are available and which one is best and he is choosing it explicitly. During the initial construction of the system's knowledge-base, many situations will arise when this is exactly the case. However, as the system becomes more powerful and the knowledge more complex, it is less likely the user can choose so precisely.

The posting of the *Use* goal has an important side-effect: the triggering goal associated with the selected method is posted as if done by the user. The user will be required to fill-in the necessary slots of the goal which also act as the needed context for the method. We will look at a hypothetical example taken from the router development.

Development context: in step 1.1 of the development, the user posts the goal of removing the relation **PACKAGES_EVER_AT_SOURCE** from the specification. Suppose instead that he decided to bypass the problem solving in 1.1 (and 1.2, 1.3) and choose a method to use directly, in this case **FoldGenericIntoRelation**.

Given the following state

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      if
▸1        (the package.previous ||
              package.previous immediately before package.new
                wrt PACKAGES_EVER_AT_SOURCE(*)
          ):DESTINATION ≠ package.new:DESTINATION
       then WAIT[];

      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;
```

the user would post the following goal

```
▶PlanSpace: Post post a goal
          Goal: Use
             Method: FoldGenericIntoRelation
                Splicing implicit goal Isolate
          Goal: Isolate
             Expr: ▶1
          Method FoldGenericIntoRelation invoked ...
```

The triggering goal of the selected method is *Isolate* which has a single argument, an expression Expr. The *Use* goal is replaced with an *Isolate* goal and the user is asked to supply a value for Expr, in this case the derived object $▶_1$. Once the goal has been filled-in, the method is invoked.

This example was chosen to make several points. A previous development produced by Feather and London [London & Feather 82] skips directly to the application of the fold method, thus obviating the problem solving necessary in the Glitter development of Appendix B. While a speed-up in development time is likely to be gained, there remain several drawbacks:

- Two levels of the planning space have been eliminated: 1) remove the relation **PACKAGES_EVER_AT_SOURCE** by 2) removing all references to it. Although the system will post an *Isolate* goal automatically, the motivation for isolating the derived object has been lost. Without the two higher level goals, much of section 1 of the development appears unmotivated as well.

  This flattening of the planning space is directly proportional to the level of method the user chooses to invoke. In the TI model, the user always must choose at the program manipulation level, eliminating all of the planning space. This could be simulated in Glitter by exclusive use of the *Apply* descriptor.

- An interesting competing development strategy is missed. The isolate/maintain strategy employed in the Feather and London development seems a perfectly reasonable one. The Glitter development relies on the same strategy. However, during Glitter's problem solving process, a competing strategy (**BabyWithBathWater**) is suggested: remove the reference $▶_1$ by removing the conditional that it is embedded in. That is, when a package enters the router, wait

unconditionally (see snapshot below). The appropriateness of this strategy rests on showing that the conditional predicate is always true, i.e., no consecutive packages have the same destination[31]. We believe that strategies such as this are frequently overlooked when a user moves directly to a method invocation.

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      invoke WAIT[];
      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;
```

[31] If we can't show this, we may consider the case where it is rare that any two consecutive destinations are the same, i.e., the resources necessary to always check the conditional are not worth it. To get rid of the conditional using this motivation cannot be done with a transformation, but requires a spec change. In section 9.3 we look at exactly this change and its consequences to the original development.

5.7. Goal-descriptor Summary

Appendix E provides an example driven presentation of the current Glitter goal set. Here we summarize each descriptor.

Casify takes as an argument any construct capable of being broken into separate cases, e.g., actions, constraints, expressions; achieved when the original construct is replaced with two or more cases.

ComputeSequentially takes two actions as arguments; achieved when the first computationally precedes the second.

Equivalence takes two constructs; achieved when both are structurally equivalent.

Factor takes a template and a context; achieved when all constructs which match the template within the context have been replaced with references to a newly constructed global definition.

Flatten takes a context; achieved when no reference to defined objects (e.g., procedures, derived relations) exist in the context.

Globalize takes a construct; achieved when the construct does not rely on the local context.

Isolate takes an expression; achieved when the expression is replaced with a reference to a derived relation.

MaintainIncrementally takes as an argument an information freedom (e.g., derived relation, temporal reference); achieved when the necessary information is explicitly stored and maintained.

Map takes as an argument a freedom construct. Map is achieved when the construct has either been eliminated or transformed into some operational form.

Purify takes an action; achieved when the action does not appear inside an unimplementable portion of the specification.

Reformulate takes a construct and a pattern; achieved when the construct matches the pattern.

Remove takes a construct and a context; achieved when the construct has been removed from the context.

Show	takes a development property; achieved when the property has been asserted as true.
Simplify	takes a context; achieved when all simplification transformations have completed.
Swap	takes two actions within a begin/end block; achieved when their placement within the block has been interchanged.
Unfold	takes as an argument a global defined construct (e.g., demon, constraint, derived relation). Unfold is achieved when the global construct is replaced by local computation.

Chapter 6
Development Methods

In chapter 5 we defined one component of our model, a vocabulary for stating development goals. In this chapter we look at another model component, the methods necessary for achieving those goals. We will first look at the important properties of representing development knowledge and how Glitter addresses each. Next, we will present Glitter's method representation and demonstrate how it can be used to capture various types of development knowledge. Finally we define a set of pre-defined method building material useful for method construction. Below is a summary of the major points made in this chapter.

Once a goal is posted, all methods relevant to achieving the goal should be locatable.

Given a large catalog of development methods, finding all methods which might be useful in achieving a goal becomes a major problem. Searching such a catalog by hand is both tedious and error-prone. Glitter provides an automatic retrieval system based on goal indexing.

New methods should be readily addable.

Research into the mechanization of development knowledge is just beginning. Each new example unearths new methods. The ability to add these new methods to the system is crucial to its evolution. Issues related to both the construction and addition of new knowledge are discussed.

The entire planning space must be covered.

Traditional transformation systems deal solely with manipulating the *program* space. In Glitter, methods must address the manipulation of the *problem* space as well, e.g., goal reduction, subgoaling.

In many cases, a chosen method is not directly applicable in the current state, i.e., jittering is necessary.

Glitter methods provide explicit subgoaling to reach a matching state.

The method representation should be analyzable by other components of the system.

In particular, the selection mechanism (see Chapter 7) requires information about

competing methods including the actions they propose to take and their instantiation of local variables.

6.1. Properties of Method Representation

Glitter is a knowledge-based system. One component of that knowledge is the development methods that are useful in achieving development goals. We assume that such methods will be stored in a method catalog. Here we will look at some of the desired properties of the catalog and the methods in it.

6.1.1. Knowledge accessibility

Given a problem description (development goal), we would like to find *all* relevant methods for solving the problem. Our experience is that a manual search of a method catalog is both wasteful of the user's time and error-prone, i.e. relevant methods are often overlooked.

In Glitter, each method is indexed to a particular development goal. When a goal is *activated* (posted and shown not to hold in the current state), all methods indexed to it are are formed into a candidate set (see section 7.2.1). Note that this type of indexing is geared towards problem solving; it may be inadequate for other browsing type of activities:

- A catalog maintainer may wish to peruse the method catalog for methods that have a certain applicability condition or employ some technique. The CHI system [Green et al. 81] allows a user to retrieve methods by content, e.g., "Find all transformations which contain X in their left hand side", "Find all transformations which rely on property P".

- A developer may be interested in all of the methods which became applicable after a certain program change was made. The DRACO system [Neighbors 80] uses *meta-rules* to derive information about which new transformations will be applicable after a particular transformation has fired.

We view each of these capabilities as useful extensions of the current Glitter system. Note that both rely on a form of representational transparency discussed below.

6.1.2. Adding new methods

As our experience base grows, new development knowledge will need to be added to the catalog. There are several aspects to this. First, there is the problem of *constructing* a method to capture a needed piece of development knowledge. This is a problem of a) providing the necessary representational power, and b) defining method-building materials, which allow for quick construction. Both of these are discussed in more detail in sections 6.2 and 6.3.

The second aspect is what McDermott refers to as *additivity* [McDermott 78]: the ability to incrementally add new knowledge to the system and show that the new knowledge will be used at the appropriate times. In systems like TI, where the user is responsible for searching the method catalog, the addition of each new method slightly *reduces* the likelihood of the user collecting all methods applicable to a given goal. That is, as the catalog increases, the additivity property decreases.

In Glitter, a method is defined as an independent piece of development knowledge and interfaces with the system as a whole through its goal index. Hence, once a method is added, it is immediately usable by the system. However, additivity based on knowledge independence comes at the price of problem solving efficiency. Other systems use a more tightly coupled form of knowledge in an attempt to cut down on search [Kibler 78], [Neighbors 80], [Terry 82]. They pay the price in additivity: the addition of new knowledge to these systems requires a re-organization of the knowledge base.

6.1.3. Coverage of development planning space.

The methods must represent both knowledge about ways of manipulating the *program* space and ways of manipulating the *problem* space. An example of the latter is the following:

> **DivideAndConquerDemons:** *If you want to map a complex demon then try splitting it into several simpler demons and mapping each individually (divide-and-conquer).*

Map D|***demon***
➡
Split D into D_1|***demon***...D_k|***demon***
Map D_1...D_k

An example of a splitting method is:

SplitConjunctiveTrigger: *If you want to split a demon into simpler cases and the demon has a conjunctive trigger then apply transformation SplitConjunctiveTrigger.*

```
demon
 trigger P and Q;
 response R;
⇒
demon
 trigger P
 response R when Q;

demon
 trigger Q
 response R when P;
```

The first method reduces a difficult goal into several simpler goals, i.e., it transforms the problem space. The second method replaces a demon with two or more new demons, i.e., it transforms the program space. In practice, the second implements the split goal of the first. A Glitter method should be able to represent either type of transformation. We discuss the necessary representation in more detail in section 6.2.

6.1.4. Automatic jittering

A problem with the base-line TI model is the limited applicability of the transformations: once a user finds a transformation he would like to apply, the system will be unable to apply it if it cannot match the transformation's left hand side against the current state, i.e., sub-goaling is not supported. Given that a particular method has been judged *appropriate* for achieving the current development goal (i.e., selected directly by the user or indirectly by the system's selection process), some preliminary jittering steps may be necessary before the method can be applied. Our results show that jittering makes up a significant portion of a development (see section 5.2.3 for a general discussion of jittering). Hence, we would like a means of carrying out the jittering automatically.

A predecessor of Glitter called the Jitterer [Fickas 80] used a GPS control structure [Newell 72] to automate jittering. In this system, the user was responsible for choosing a transformation to apply. If the transformation did not apply, the system passed the transformation's left hand side pattern and the current state to a system called the Differencer

[Chiu 80], which produced a set of difference descriptions which could be viewed as low level editing commands, e.g., "delete these three constructs and add this one", "commute these two statements". These commands, when applied to the current state, would edit the program into a state that matched the left hand side pattern. i.e., a state from which the transformation could be applied. The problem was how to map the Differencer's output onto a sequence of transformations which would actually produce the necessary correctness-preserving changes. The Jitterer's approach was to attempt to translate the description produced by the Differencer into higher level development goals (the forerunners of the goals of Appendix E). Each transformation was augmented with one or more goals which provided the necessary indexing. Hence, once the translation process was complete, the relevant transformations could be gathered.

There were two major problems with this approach:

1. Of practical importance Chiu's Differencer was not yet implemented. It was necessary to hand simulate it in the Jitterer.

2. More importantly, the translation of the Differencer's output into higher level development goals was not practical as a *general* approach in Gist developments. That is, the language necessary to describe the changes produced by a TI transformation in a Gist development (e.g., mapping, casifying, information movement) was at a much higher level than the Differencer's description. We stress the word *general* here. There are many cases of jittering during a development where the necessary changes are of a mundane low level variety. For example, jittering logical or arithmetic expressions often involves changes closely matched to the Differencer's description. Since the Differencer is now implemented, our future plans include exploring ways that in can be incorporated into Glitter's problem solver as a useful jittering tool.

Because of the above problems, a way to eliminate the need for a differencing engine in the jitterering process was needed. The solution, as embodied in Glitter, was to make each individual method responsible for the jittering necessary to apply it. In the Jitterer, the Differencer's role was to produce a set of "goals", which when achieved would leave the program in a state where the method transformation was applicable. The problem was that the goal language was not at the level of the transformations which must achieve them. Glitter's approach is to have each method produce the goals needed to produce its pattern *in a high level development language* (see chapter 5) and *independent of the current state*. Thus, each method is responsible for posting a set of goals which will change the current state into the necessary form. The method must be prepared for the worst case where all of

the subgoals may be necessary; often one or more of the goals will be achieved trivially in the current state, i.e., the current state will partially match the pattern. In some sense, each method can be viewed as having its own built-in differencer. Using this approach generally results in run-of-the-mill backward-chaining control. However, as described in Chapter 5, Glitter goals are independent of methods and allow a more general GPS type of control if necessary.

Below we list some further aspects of Glitter's approach to jittering.

6.1.4.1. Eagerness

Given that method M has been selected as the method to employ in achieving goal G, then M should be *eager* to apply itself. If the program is not in the right state, then part of M's actions will be to remedy the situation by calling for the necessary jittering steps (posting the necessary sub-goals). As an example, suppose we are given a method MergeDemons for consolidating two demons with the same trigger into a single new demon:

> **MergeDemons**: *Given two constructs D1 and D2, if D1 and D2 are both demons and have the same trigger and the same local variables then under certain conditions they can be consolidated into a single demon.*

Suppose that this method has been selected to consolidate two constructs S1 and S2. An eager MergeDemons would do the following: 1) if S1 or S2 are not demons then change them into demons, 2) if the two triggers are not equivalent then make them equivalent, 3) if the local variables are not equivalent then make them equivalent, and finally 4) replace the two demons with a consolidated third.

Note the importance of the method selection process here: the philosophy is that if M is selected then M is the best candidate for the job and is set free to change the program in arbitrarily complex ways to reach a desired state. The selection process becomes an important filter in weeding out unlikely methods, and hence, potentially costly excursions down wrong paths. In effect, we have moved the burden of determining method applicability from the methods themselves to the selection engine. Chapter 7 discusses the system's selection knowledge in detail.

6.1.4.2. Prudence

A consequence of the eagerness property is a robust collection of methods with wide applicability. Another less desirous property is that when a development goal is posted, the set of methods competing for attention will generally include ones that are *unfeasible* or *unlikely* to achieve the particular development goal, i.e., overeager methods. We have mentioned the need for a strong *general* selection mechanism to combat this problem. We may also be able to add *local* knowledge which will filter out a method under certain conditions. Such filtering knowledge often has a subjective flavor since the conditions *unfeasible* and *unlikely* are currently subjective. Using the MergeDemons example, it is unlikely that the method should be attempted if D1 and D2 are not initially bound to demons: reformulation of non-demonic constructs into demonic ones is a dubious undertaking[32].

If a method is erroneously filtered out, the consequence is that the user will be responsible for supplying enough of the jittering steps to pass the filter test. Note that we can simulate the non-subgoaling TI model by adding to each method M a pattern P which represents a left-hand-side pattern. We require that M be considered only if P matches exactly against the appropriate portion of code.

6.1.4.3. Level of Effort

We have described eagerness and prudence as binary choices: a method may either elect or reject to pursue a particular subgoal. In some cases, the method may wish to attempt to achieve a subgoal to some level of effort. For instance, MergeDemons may wish to try reformulating one construct as a demon if the other is already a demon, *but only to a limited extent*. After a certain amount of problem solving resources have been expended, the method will signal abandonment. Glitter currently provides no hooks for attaching this type of resource utilization knowledge to a method, i.e., the choice remains binary. We view incorporating this type of knowledge into jittering in particular, and the Glitter problem solving engine in general as a significant research effort. Section 7.2.5 discusses more sophisticated search control from a broader perspective.

---

[32] This is an overgeneralization. It is feasible to reformulate certain structures (e.g., constraints) into demon form. Also, if only one construct is non-demonic then we may want to compare the method with its competitors before rejecting it. Currently, the MergeDemons method requires both D1 and D2 to be bound to demons, and hence considers neither of these factors.

6.1.5. Representational transparency

Our development approach is based on the user playing an active role in planning. Such collaboration requires that the human be able to follow the planning process in general and the effects of individual methods in particular. Further, if the system is to reason about the best method to apply in a given situation, the ability to examine the effects of each competing method becomes crucial. As Davis [Davis 80] points out, one powerful means of determining this is to directly analyze the content of each method.

The internals of a Glitter method are transparent down to the transformation application level. That is, the fillers of each field of a method consist of components with analyzable semantics. This is true for all but the *Apply* goal. While we can reason that the posting of an *Apply* goal will lead to a change in the program state, we cannot analyze what the change will be. The method writer defines his own procedure for carrying out the application of a program transformation. The analysis of the procedure code is beyond the capabilities of the system. Section 7.2.2.2 discusses possible solutions to this problem.

In the next section we define the method notation used in Glitter

6.2. Method Template

In this section we present the representation of development knowledge in the Glitter model, i.e., the *method*. A method template takes the following form:

```
Method <unique name>
    Goal: <development goal>
    Filter: [<boolean expression>]^0
    Action: [<development actions>]^1
          [ Short description of method. ]
End method
```

A 0 superscript indicates zero or more items, a 1 superscript indicates one or more items. In general, a method can be read as "if the *goal* is G and the following conditions hold (*filtering* properties are met) then try the following *actions* to achieve G". Below is a further description of each of the method's fields:

Goal field: filled with a Glitter development goal as defined in chapter 5. When a goal becomes *active* (is posted and not trivially achieved) and matches the filler of this field, the method is *triggered*.

Filter field: filled with zero or more boolean expressions. Multiple expressions are assumed to be conjunctive. All expressions must evaluate to true if the method is to be added to the *candidate set* (see section 7.2.1). The filter provides a hook for non-subgoalable pre-conditions on a method (see *prudence* under property 4). In a following section we define a set of functions which are helpful in building filters.

Action field: filled with an ordered sequence of one or more *development actions*. A development action is either a subgoal to be posted or the action mapping function *forall*. The latter maps one or development actions onto one or more components of a structure. Mapping functions and generators are discussed in more detail in section 6.3.

Actions are initiated after the method is a) triggered, b) filtered, c) added to the candidate set and d) chosen by the *selection engine*. The latter is discussed in detail in chapter 7. When all actions have been successfully completed, the method is marked as completed. A method completing and the triggering goal being achieved are independent events. Thus, a method is not guaranteed to achieve its triggering goal. Sometimes it may just move goal achievement closer, although no guarantee is made of this either. If the triggering goal is not achieved when a method finishes, the goal is re-posted and the method triggering process starts anew.

By default, actions are attempted as ordered. That is, action A_i must complete before A_{i+1} begins. An override of the default ordering can be specified using a selection rule as described in chapter 7. Briefly, a selection rule can re-order two or more actions depending on development specific information.

Next we will look at the construction of several development methods using the above notation.

6.2.1. Some examples of method construction

In this section we will follow the construction of several methods taken from the method catalog of Appendix F. We will first describe the method informally and then use the process below to build the corresponding Glitter method. Note that this process must be currently carried out by hand; future work includes studying ways that it may be partially automated.

1. Translate implicit intent into an explicit development goal. Make this goal the trigger of the method. Sometimes a method may have multiple effects, and correspondingly, multiple intents. Each intent is broken out into a separate method (Disjunctive goals are not currently allowed in a method's *Goal* field.).

2. If the method requires that the program be in a certain state before it can be applied, define the subgoals necessary to bring that state about. Make these part of the action portion of the method (see *eagerness*, section 6.1.4).

3. If the method has applicability conditions attached to it then define a *Show* goal for each and make them part of the action portion of the method.

4. Translate the modification carried out by the method as one or more goals. Add each to the action portion of the method.

5. Incorporate any *local* constraints that are possible (see *prudence*, section 6.1.4). If certain instantiations of the method are unlikely to lead to an achievement of the goal, rule them out by using the *Filter* field.

We next look at three concrete examples. First, the method MergeDemons introduced in section 6.1.

6.2.1.1. Construction of the method MergeDemons

MergeDemons: *Given two constructs D1 and D2, if D1 and D2 are both demons and have the same trigger and the same local variables then under certain conditions they can be consolidated into a single demon.*

Below is the 5 step construction process applied to the above informal description (we will use a slightly sugared notation when presenting the fillers of the various method fields; section 6.3 defines the actual notation):

1. *Goal definition*. The effect of this method is to *Consolidate* two demons; this is made the goal of the method.

```
| Method  MergeDemons                                   |

        Goal: Consolidate D1 and D2
        Filter:  a) ...
        Action:  1) ...

     [Consolidate two demons into one.]
| End Method                                            |
```

2. *Define jittering steps.* To carry out the consolidation, several things must be present in the current state: 1) two demons with 2) equivalent triggers and 3) equivalent local variables. Syntactically, we can represent this as (all underlined items are quoted, others are pattern variables):

<u>demon</u> D1 (vars)
  <u>trigger</u> t
  <u>response</u> r1
...
<u>demon</u> D2 (vars)
  <u>trigger</u> t
  <u>response</u> r2

Represented as subgoals, we get the following:

- ☐ *Reformulate* D1 as ***demon***
- ☐ *Reformulate* D2 as ***demon***
- ☐ *Equivalence* triggers (t) of D1 and D2
- ☐ *Equivalence* declared variables (vars) of D1 and D2

We now have

```
| Method  MergeDemons                                          |

          Goal: Consolidate D1 and D2
          Filter:  a) ...
          Action:  1) Reformulate D1 as demon
                   2) Reformulate D2 as demon
                   3) Equivalence triggers of D1 and D2
                   4) Equivalence declared variables
                        of D1 and D2
                   5) ...

        [Consolidate two demons into one.]
| End Method                                                   |
```

3. *Subgoal on applicability condition*. The applicability condition of MergeDemons is MERGEABLE_DEMONS, which requires showing that the two responses r1 and r2 can be interleaved to form the new response. After defining the corresponding *Show* goal we have:

```
| Method  MergeDemons                                          |

          Goal: Consolidate D1 and D2
          Filter:  a) ...
          Action:  1) Reformulate D1 as demon
                   2) Reformulate D2 as demon
                   3) Equivalence triggers of D1 and D2
                   4) Equivalence declared variables
                        of D1 and D2
                   5) Show MERGEABLE_DEMONS(D1, D2, I|interleaving)
                   6) ...

        [Consolidate two demons into one.]
| End Method                                                   |
```

4. *Define effect*. The program transformation we want to carry out is the construction of a new demon out of the old two. We define a Lisp procedure that takes as arguments the demons bound to D1 and D2, checks to make sure that the triggers and declared variables are equivalent, builds a new demon using shared parts and the interleaving supplied by the *Show* goal, and finally deletes the two old demons and inserts the new. The procedure is made the argument of an *Apply* goal. Note that the procedure is self-contained: it is a transformation that can be applied directly and that can be guaranteed to produce a correctness

preserving change in the program. Thus if for some reason one of the previous method subgoals becomes "unachieved", the transformation will not fire. The user will likely be required to step in at this point and fix things up.

```
| Method  MergeDemons                                        |

        Goal: Consolidate D1 and D2
        Filter:  a) ...
        Action:  1) Reformulate D1 as demon
                 2) Reformulate D2 as demon
                 3) Equivalence triggers of D1 and D2
                 4) Equivalence declared variables
                      of D1 and D2
                 5) Show MERGEABLE_DEMONS(D1, D2, I|interleaving)
                 6) Apply DEMON_MERGE(D1, D2, I)

     [Consolidate two demons into one.]
| End Method                                                 |
```

5. *Define local constraints*. It is improbable that two non-demon structures marked for consolidation will need to be reformulated into demons. That is, we can view the reformulation of a structure into a demon as a major step and one beyond simply jittering. Therefore, we add a filter that restricts our demon merge method to work on only demons, removing the first two reformulation goals. In effect we have decided against subgoaling in certain situations. Note the negative consequences of this decision: any consolidation requiring that the constructs bound to D1 and D2 be reformulated as demons will not trigger this method; the reformulation goal(s) will have to be supplied by the user.

```
| Method  MergeDemons                                        |

        Goal: Consolidate D1 and D2
        Filter:  a) D1 isa demon
                 b) D2 isa demon
        Action:  1) Equivalence triggers of D1 and D2
                 2) Equivalence declared variables
                      of D1 and D2
                 3) Show MERGEABLE_DEMONS(D1, D2, I|interleaving)
                 4) Apply DEMON_MERGE(D1, D2, I)

     [Consolidate two demons into one.]
| End Method                                                 |
```

The actual MergeDemonS method is given below. Note that the two filters have been moved to the goal statement, and that accessor functions (trigger-of, declaration-of) replace the more informal descriptions.

```
| Method  MergeDemons                                                  |

        Goal: Consolidate D1|demon and D2|demon
        Action:  1) Equivalence trigger-of[D1] and trigger-of[D2]
                 2) Equivalence declaration-of[D1] and
                        declaration-of[D2]
                 3) Show MERGEABLE_DEMONS(D1, D2, I|interleaving)
                 4) Apply DEMON_MERGE(D1, D2, I)

     [You can consolidate two demons if you can show that they have the same
     local variables, the same triggering pattern and that they meet certain
     merging conditions.]
| End Method                                                           |
```

Before leaving this example, we should note that the next step would be to define any method-specific knowledge on when this method is a good choice and when, if ever, the default action ordering should be overridden. In particular, MergeDemons is the only method which is able to consolidate two demons and this fact is recorded in a selection rule (*MergeDemons). It is sometimes easier to attempt action 2 before action 1 and this is also recorded in a selection rule (TriggersAlmostEquiv). The next chapter discusses in detail the representation of this type of knowledge.

6.2.1.2. Construction of the method RemoveRelation

Our second example introduces the mapping of a goal onto one or more components. The method, called RemoveRelation, can be stated as follows:

> **RemoveRelation**: *If the goal is to remove a relation from the spec then try removing all references to it. Once this is accomplished the definition can be removed.*

The construction of the corresponding Glitter method follows (again we will use a slightly sugared notation):

1. *Goal definition*. The effect of this method is to *Remove* a relation from the specification.

```
| Method  RemoveRelation                                              |

            Goal: Remove R|relation from spec
            Filter:  a) ...
            Action:  1) ...

        [Remove a relation.]
| End Method                                                          |
```

2. *Define jittering steps.* To remove the definition we must first get rid of all references to the definition. In this case, there is no corresponding syntactic representation. To define the necessary sub-goals, we must map the *Remove* goal onto each reference of R. We use the forall mapping function:

```
| Method  RemoveRelation                                              |

            Goal: Remove R|relation from spec
            Filter:  a) ...
            Action:  1) forall references of R called RR
                          do Remove RR from spec
                     2) ...

        [Remove a relation.]
| End Method                                                          |
```

Note that if no references of R exist in the current state then the forall function produces no subgoals.

3. *Subgoal on applicability condition.* This method has no applicability conditions.

4. *Define effect.* The program transformation we want to carry out is the removal of a relation definition from the spec. We define a Lisp procedure which takes as an argument the relation bound to R, checks to make sure that no references to it are found, and removes it from the spec. The procedure is made the argument of an *Apply* goal.

```
| Method  RemoveRelation                                         |

          Goal: Remove R|relation from spec
          Filter:  a) ...
          Action:  1) forall references of R called RR
                        do Remove RR from spec
                   2) Apply REMOVE_UNREFERENCED_RELATION(R)

       [Remove a relation.]
| End Method                                                     |
```

5. *Define local constraints*. There are several heuristics for estimating the likelihood of the RemoveRelation method succeeding. Each looks for particular features of the program that signify that the relation is likely unneeded and so can be removed. However, the absence of these features is not enough to rule out the method. Hence, these heuristics are made a part of the method selection process discussed in Chapter 7 as opposed to the triggering process.

The actual RemoveRelation method is given below.

```
| Method  RemoveRelation                                         |

          Goal: Remove R|relation from spec
          Action:  1) forall reference-location[R,RR,spec]
                        do Remove RR from spec
                   2) Apply REMOVE_UNREFERENCED_RELATION(R)

       [You can remove a relation if you can remove all references to it.]
| End Method                                                     |
```

6.2.1.3. Construction of the method MaintainDerivedRelation

Our last example shows a transformation carried out solely in the *problem space*. The method, called MaintainDerivedRelation, can be stated as follows:

MaintaintDerivedRelation: *If the goal is to map a derived-relation then try maintaining it incrementally.*

The construction of the corresponding Glitter method follows:

1. *Goal definition*. The effect of this method is to *Map* a derived-relation.

```
| Method  MaintainDerivedRelation                            |

          Goal:  Map DR|derived-relation
          Filter:   a) ...
          Action:   1) ...

       [Map a derived-relation by incremental maintenance]
| End Method                                                 |
```

2. *Define jittering steps.* This is a straight goal reduction; there are no jittering steps.

3. *Subgoal on applicability condition.* This method has no applicability conditions.

4. *Define effect.* The effect is the transformation of the mapping goal into a more concrete goal, i.e., incrementally maintain the relation.

```
| Method  MaintainDerivedRelation                            |

          Goal:  Map DR|derived-relation
          Filter:   a) ...
          Action:   1) MaintainIncrementally DR

       [Map a derived-relation by incremental maintenance]
| End Method                                                 |
```

5. *Define local constraints.* There are several pieces of selection knowledge which pertain to this method. The first involves comparing it with other competing mapping methods, clearly not a local constraint. This knowledge is defined in terms of selection rules and will be applied during method selection as described in Chapter 7. The second notes that it is not useful to attempt to incrementally maintain a relation which is unchanging, i.e. static. This knowledge is placed in the filter. As with all filtering knowledge, it could alternatively have been made a selection rule, and hence part of the method selection process. We have chosen to place it in the filter because of its clear discriminatory power. There is a drawback to this placement: staticness can be moderately costly to check for and out of the system's control as a filter.

```
| Method  MaintainDerivedRelation                                   |

            Goal:  Map DR|derived-relation
            Filter:   a) DR is not static
            Action:   1) MaintainIncrementally DR

      [Map a derived-relation by incremental maintenance]
| End Method                                                        |
```

The actual MaintainDerivedRelation method is given below.

```
| Method  MaintainDerivedRelation                                        |

            Goal:  Map DR|derived-relation
            Filter:   a) ~static[DR]
            Action:   1) MaintainIncrementally DR

      [One way of mapping a derived relation is to maintain it explicitly.]
| End Method                                                             |
```

6.3. Method Vocabulary

In this section, we will look at the predefined functions (or what McDermott [McDermott 77] refers to as the *problem vocabulary*) which are available in method construction. These deal with Gist syntax, iteration, pattern-matching, etc. The use of such a vocabulary serves several purposes: 1) it buffers the user from the internal representation of the program, 2) it facilitates the type of *content reference* needed for method selection (see section 7.2.2.2) and 3) it provides convenient building blocks to the method writer.

6.3.1. Mapping function

We frequently want to map an action (or actions) onto several constructs. For instance, in section 6.2.1.2 we needed to map the remove goal onto each reference of a relation. The forall function gives us the necessary capability:

```
forall <generator>
       [suchthat <predicate>]^0
         do [<development action>]^1
```

It is important to note that forall simply enumerates the set of development actions to be carried out; all actions are created before any are considered for activation. The ordering of the actions depends on the generator. In any case, the ordering can be overridden by action ordering rules as discussed in Chapter 7. If no mappings are produced -- the generator is empty or the suchthat filter eliminates them all -- the function is removed from the method's action sequence.

6.3.2. Predicates, Generators and Accessors

Glitter provides a method writer with a predefined set of functions to aid in method construction. These functions may appear in either a filter or iterator. They can be used in the following ways:

- *Predicate*. Each function always returns a nil or non-nil value.
- *Generator*. Some functions (i.e., ones marked as **generators**) can be used to generate all possible combinations of bindings when one or more of their arguments are left unbound. This is a powerful mechanism which can be used in the following ways:
    1. In a filter: will instantiate a *separate* method for each binding. The set will be *disjunctive* and *competing* to achieve the triggering goal.
    2. In a forall: will generate all bindings producing a *conjunction* of actions *co-operating* to achieve the triggering goal.
- *Accessor*. A subset of the functions deal with *walking* the specification parse tree[33]. We call these access functions. The function F(S, C) used in the item above (known as *component-of* in the system) is an access function. Section 6.3.4 lists the access functions used in our examples.

There are several things worth further discussion in the case of generators. First, note the

---

[33] Up to this point, we have been viewing the program under development as a pretty-printed textual form. For general understandability, this seems the right choice. However, the actual object under development is a program parse tree. The semantics of several of the functions we will be discussing can be described more concisely in terms of the parse tree internal form.

difference between the use of a generator in a filter and in an action: one produces an alternative set, the other a cooperating set. For example, a general strategy for removing a compound structure S is to remove each of its components (divide-and-conquer). Suppose we had a function F(S|***structure***, C|***component***) for generating all components of S. If we placed F(S, C) in the filter of a method where S was bound and C was not, we would produce k method instantiations were k is the number of components of S. The choice of one of these methods would allow us to remove only one component. On the other hand, using F(S, C) as the generator in a forall loop produces a set of k subgoals which all must be achieved before the method completes, i.e., the method removes all components.

Second, a word about generation order. Currently, all Glitter generators generate items in an arbitrary order. This requires that any action ordering knowledge be represented as action ordering rules, which analyze the order of actions after all have been generated. A more efficient approach would be to build some of the ordering smarts into the generators themselves. We have refrained from doing so because such knowledge would necessarily be of a procedural form unanalyzable by the system as a whole.

Notation: the type *construct* represents (can be bound or point to) some syntactic portion of the current Gist program under development (a node in the parse tree). The type *pattern* can represent one of several things:

- Gist syntactic type. For example, ***demon, loop, action*** are all valid patterns.
- Template. A partially to fully instantiated Gist construct. Portions of the template may consist of wild-cards.
- Named template. Certain templates occur frequently as patterns. Any template can be given a name which is then used as a reference.

6.3.3. General Functions

6.3.3.1. pattern-match

pattern-match[*pattern*, *construct.loc*, *construct.context*] **generator**

The general pattern matching capability of the system.

- If *pattern* and *construct.loc* are bound, returns true if they match (*construct.context* is ignored).

- If only *pattern* and *construct.context* are bound, binds *construct.loc* to (and returns as value) any program fragment within *construct.context* which matches *pattern*, nil if no match. As a generator, returns all fragments that match *pattern* within *construct.context*.
- Any other binding combinations return nil.

As an example, the following method attempts to find a pattern of the Gist syntactic type ***demon*** and bind it to D2. To consider all cases, we need a separate method instantiation for each consolidation partner, i.e. each demon other than the one bound to D. We will let the selection process sort out which is the best choice. As discussed in section 6.3.2, we can use a generator in a filter to give multiple instances. Here we use pattern-match as the necessary generator by leaving D2 unbound.

```
| Method  MapByConsolidation                                  |

        Goal:   Map D|demon
        Filter:  a) pattern-match[demon, D2, spec]
                 b) D ≠ D2
        Action:  1) Consolidate D and D2

      [To map D, find some other demon D2 and consolidate.]
| End Method                                                  |
```

6.3.3.2. gist-type-of

gist-type-of[*construct*, ***Gist syntactic type***][34]

This is a special case of the pattern-match function: the first argument must be bound and the pattern must be a Gist syntactic type. It is included as a special case.

- If both arguments are bound, returns true if *construct* is of the right ***Gist syntactic type***.
- If only *construct* is bound, returns its ***Gist syntactic type***, nil otherwise.

gist-type-of[X, ***relation-reference***] (or X|***relation-reference***)
gist-type-of[X, ***action***] (or X|***action***)

---

[34] A more concise form of this function is frequently used: *construct*|***Gist syntactic type***

6.3.3.3. domain-type-of

domain-type-of[*object expression*, DOMAIN TYPE]

- If both arguments bound, returns true if *object expression* is of the right DOMAIN TYPE.
- If DOMAIN TYPE is unbound, returns the correct type for *object expression* else nil.

domain-type-of[*S*, SWITCH] (or *S*|SWITCH)
domain-type-of[*L*, LOCATION] (or *L*|LOCATION)

6.3.3.4. brother-of

brother-of[*construct.1*, *construct.2*] **generator**

- If both arguments are bound, returns true if both constructs have the same father.
- If one is unbound, binds and returns a brother, else nil. When used as a generator, returns all brothers.
- If both are unbound, returns nil.

6.3.3.5. case-of

case-of[*construct.case*, *case-list*] **generator**

case-list is a list of case constructs.

- If both arguments are bound, returns true if *construct.case* is an element of *case-list*.
- If *construct.case* is unbound, binds and returns one case from *case-list*. When used as a generator, returns all cases.
- If *case-list* is unbound, binds and returns the list of brother cases of *construct.case*.

6.3.3.6. maintenance-location

maintenance-location[*data-structure*, *construct.maint*, *construct.context*] **generator**

It is often necessary to generate all locations within a certain context that modify a data-structure.

- If *data-structure* and *construct.maint* are bound, returns true if the execution of *construct.maint* changes the value of *data-structure*.

- If only *construct.maint* is unbound, will bind and return a construct which changes the value of *data-structure* (e.g., insert, update, delete, assign) in *construct.context*. When used as a generator, will return all constructs modifying *data-structure* in *construct.context*.

- If only *data-structure* is unbound, will bind and return the data-structure that *construct.maint* changes.

As an example, suppose we are given the following:

```
begin
  update R($) to y;
  insert S(y);
  delete R(z)
end
```

Used as a generator,

```
forall maintenance-location[R1, Construct, SB] do ...
```

where R1 is bound to the relation R, SB to the begin-end block and Construct is free, we get the following two bindings to Construct generated by the forall:

```
1)  update R($) to y;
2)  delete R(z)
```

Currently, maintenance-location generates items in an arbitrary order. Used for a slightly different purpose,

```
forall maintenance-location[DS, Construct, SB] do ...
```

where Construct is bound to the update statement and DS is free we get the following binding to DS:

```
1)  R
```

With Construct bound to the begin-end block, we get bindings of DS as

```
1)  R
2)  S
```

6.3.3.7. reference-location

reference-location[*data-structure*, *construct.ref*, *construct.context*] **generator**

Similar to maintenance-location, but deals with *references* to (retrievals of) *data-structure* as opposed to modifications.

- If *data-structure* and *construct.ref* are bound, returns true if the computation of *construct.ref* references the value of *data-structure*.

- ☐ If only *construct.ref* is unbound, will bind and return a reference to *data-structure* in *construct.context*, nil if none exist. When used as a generator, returns all references.

- ☐ If only *data-structure* is unbound, will bind and return a data-structure referenced in *context.ref*, nil if none referenced. When used as a generator, returns all data-structures referenced.

6.3.3.8. trigger-location

trigger-location[*demon*, *construct.trigger*, *construct.context*] **generator**

It is often necessary to generate all locations within a certain context that potentially cause a demon to trigger.

- ☐ If *demon* and *construct.trigger* are bound, returns true if the execution of *construct.trigger* potentially triggers *demon*.

- ☐ If only *construct.trigger* is unbound, will bind and return a construct which potentially triggers *demon*. When used as a generator, will return all constructs potentially triggering *demon* in *construct.context*.

- ☐ If only *demon* is unbound, will bind and return a demon which *construct.trigger* potentially triggers. When used as a generator, returns all demons that potentially trigger.

As an example, suppose we are given the following:

```
begin
   demon Example1 (x)
     trigger P(x);
     response A(x);

   demon Example2 (x)
     trigger P(x) and Q(x);
     response B(x);
...
    insert P(a);
    insert Q(b);
...
```

Used as a generator,

```
forall trigger-location[Example2, Construct, spec] do ...
```

where Construct is unbound, we get the following binding to Construct on iterating through the loop:

```
1)  insert P(a)
2)  insert Q(b)
```

Used to find the demons which a construct might trigger,

```
forall trigger-location[D, IP, spec] do ...
```

where IP is bound to the insert of P(a) and D is unbound, we get the following bindings to D:

```
1)  demon Example1 (x) ...
1)  demon Example2 (x) ...
```

6.3.4. Access Functions

The following functions are useful for accessing various syntactic portions of the program.

6.3.4.1. component-of

component-of[*construct.1*, *construct.2*] **generator**

- As a predicate (both arguments bound), returns true if *construct.1* is a non-reflexive transitive sub-component of *construct.2* (*construct.1* is a descendant of *construct.2*).
- If *construct.1* is unbound, will bind and return a transitive sub-component of *construct.2* (a descendant of *construct.2*), returning nil if none exist. As a generator, will return all transitive sub-components.
- If *construct.2* is unbound, will bind and return a transitive super-component of *construct.1* (a ancestor of *construct.1*), returning nil if non exist. As a generator, will return all transitive super-components.
- If both are unbound, returns nil.

Suppose we take the following example:

```
demon Example (x)
  trigger P(x);
  response: if Q(x) and R(x) then A(x);
```

Now, given

```
forall component-of[R(x), Super] do ...
```

where R(x) is the reference in the response and Super is free, we would get the following bindings to Super on iteration of the loop

```
1)  Q(x) and R(x)
```

```
2)  if Q(x) and R(x) then A(x)
3)  demon Example (x) ...
```

Given

```
forall component-of[Sub, Conditional] do ...
```

where Conditional is bound to the demon response and Sub is free, we would get the following bindings to Sub on iteration of the loop

```
1)  Q(x) and R(x)
2)  Q(x)
3)  R(x)
4)  A(x)
```

6.3.4.2. immediate-component-of

immediate-component-of[*construct.1*, *construct.2*] **generator**

Same as component-of except that *construct.1* must be an immediate or direct component of *construct.2*.

Using the examples above with immediate-component-of

```
forall immediate-component-of[R(x), Super] do ...
```

where R(x) is the reference in the response and Super is free,

```
1)  Q(x) and R(x)
```

Given

```
forall immediate-component-of[Sub, Conditional] do ...
```

where Conditional is bound to the demon response and Sub is free,

```
1)  Q(x) and R(x)
2)  A(x)
```

6.3.4.3. pairwise-component-of

pairwise-component-of[*construct.sub1*, *construct.sub2*, *construct.sup1*, *construct.sup2*] **generator**

Used to compare *ordered* components of two structures in a pairwise fashion. The two structures *construct.sup1*, *construct.sup2* must be of the same Gist type.

- If one of the two components is unbound, will bind and return the corresponding component.

- If one of the two structures is unbound, will bind and return the corresponding structure.

- If both components are unbound, will bind and return two corresponding components of the two structures. When used as a generator, will return all corresponding components.

- Any other combinations return nil.

Suppose we are given two structures

```
S1: update x of R($) to y
S2: update w of S($) to z
```

The loop

```
forall pairwise-component-of[C1, C2, S1, S2] do ...
```

would bind C1,C2 to the following on iteration

```
1) x,w
2) R($),S($)
3) y,z
```

6.3.4.4. component-correspondence

component-correspondence[*construct.sup1, construct.sup2, C***/correspondence**]

generator

Used to compute a pairwise correspondence between *unordered* components of two structures of the same Gist type.

- If all arguments are bound, will return true if C is a valid correspondence of the components of the two structures.

- If C is unbound, will compute a correspondence between the components of the two structures.

- Any other combinations return nil.

Suppose we are given two local variable declaration lists (A B) and (C) as bindings to the first two arguments. component-correspondence will generate the following correspondences: <A,C>, <B,C>.

6.3.4.5. body-of

body-of[*construct.body*, *construct.definition*]

- If both arguments bound, returns true if *construct.body* is the definition body of some defined construct (e.g., demon, constraint, derived-relation) *construct.definition*.
- If only *construct.body* is unbound, returns and binds the body of *construct.definition*.

6.3.4.6. parameter-of

parameter-of[*parameter*, *construct*] **generator**

construct may either be a parameterized definition (e.g., relation, action) or a parameterized reference (e.g., relation reference, action call).

- If both arguments are bound, returns true if *parameter* is one of the parameters of *construct*.
- If *parameter* is unbound, binds and returns a parameter of *construct*. When used as a generator, returns all parameters of *construct*.

6.3.4.7. trigger-of

trigger-of[*construct.trigger*, *demon*]

- If both arguments bound, returns true if *construct.trigger* is the trigger of *demon*.
- If *construct.trigger* is unbound, binds and returns the trigger of *demon*.

6.3.4.8. local-var-of

local-var-of[*variable-name*, *construct.declarative*] **generator**

construct.declarative must allow the declaration of local variables (e.g., scoping-blocks, demons, quantifiers).

- If both arguments are bound, returns true if *variable-name* is declared by *construct.declarative*.
- If *variable-name* is unbound, binds and returns a variable declared in *construct.declarative*, nil if none declared. As a generator, returns all variables declared by *construct.declarative*

6.3.4.9. scoped-in

scoped-in[*variable-reference, construct.declarative*] **generator**

construct.declarative must allow the declaration of local variables (e.g., scoping-blocks, demons, quantifiers).

- ☐ If both arguments are bound, returns true if *variable-reference* uses a variable declared by *construct.declarative*.
- ☐ If *construct.declarative* is unbound, binds and returns the construct which declares (scopes) the variable used in *variable-reference*.
- ☐ If *variable-reference* is unbound, binds and returns a variable reference that uses a variable declared in *construct.declarative*. When used as a generator, returns all references.

6.3.4.10. name-of

name-of[*name, construct.named*]

construct.name must be a construct with a name associated with it (e.g., variable, relation, constraint, demon, action).

- ☐ If both arguments bound, returns true if *name* is the name associated with *construct.named*.
- ☐ If *name* is unbound, binds and returns the name of *construct.named*.

6.3.4.11. update-relation-of

update-relation-of[*relation-reference, update*]

- ☐ If both arguments bound, returns true if *relation-reference* is used as the object being updated in *update*.
- ☐ If *relation-reference* is unbound, binds and returns the object being updated in *update*.

6.3.4.12. new-value-of

new-value-of[*object, update*]

- ☐ If both arguments bound, returns true if *object* is the new value of *update*.
- ☐ If *object* is unbound, binds and returns the new value of *update*.

6.3.4.13. recursive

recursive[*construct.definition*]

A predicate which is true if *construct.definition* is either directly or indirectly recursively defined.

6.4. Direct Method Invocation

A method can be invoked directly, bypassing the normal problem solving process. This is accomplished through the *Use* goal as described in section 5.6.3. When invoking a method this way, the triggering goal associated with the selected method is posted as if done by the user. The user will be required to fill-in the necessary slots of the goal which also act as the needed context for the method.

The use of direct method invocation, while saving problem solving time, has some disadvantages: development documentation is weakened; alternatives are disregarded; possible deleterious side-effects are ignored. Section 5.6.3 discusses each of these in more detail.

6.5. Hearsay-III Method Representation

Methods are implemented as *domain knowledge sources*, or simply KS. A KS consists of a *trigger*, *immediate code* and a *body*. The trigger reacts to changes on the blackboard (a.k.a. problem solving state). A method's *goal* and *filters* are implemented as a KS trigger. The immediate code is executed at triggering time. It is used to set up the actions of the triggering method and add the method to the candidate set. After the immediate code has been executed, Hearsay creates an *activation record* for the method and places it on the *scheduling blackboard* (see section 2.4 for details). The activation record notes the triggering context and points to the method's body. The body is executed only after the activation record has been selected by the scheduling process. In Glitter, a method's body contains code which marks the method as the one chosen and spawns a new context.

Chapter 7
The Selection Process

During a Glitter development, there arise various points where selections must be made:

1. Given one or more competing methods, we must decide which if any should be selected.

2. Given a selected method, we must decide if the method's default action ordering should be used; we may choose a reordering of independent actions based on specialized development knowledge.

3. Given an unachievable goal, we must decide what previous problem solving state the development be should backed up to.

4. Given an overall development strategy, we must choose the high level goals which will implement it.

We consider the definition, representation and use of *selection knowledge* -- knowledge useful in making the right choice in each of the above areas -- a necessary component of our model. Glitter's current selection knowledge lies in areas 1 and 2; in this chapter we will describe how this knowledge is represented and used. At the end of the chapter (section 7.5) we will discuss ways to incorporate selection knowledge for areas 3 and 4.

Before getting into the details of Glitter's selection process, we will summarize the important points made in this chapter:

In making a selection, both implementation and problem solving efficiency must be considered.

Part of the development process involves making design decisions which will affect the efficiency of the final implementation, e.g., use method A instead of method B if you want to conserve space (time). Also important is the amount of time necessary for the selection engine to make the optimal choice. The Glitter search space is large. An exhaustive search is out of the question. With a user in the loop, some estimate of how much time a particular method will take to complete must be considered.

Partnership paradigm shines in all its glory.

The machine provides a repository of accumulated selection knowledge and is able to call it forth in the right situations. Further, the machine will perform detailed and tedious analysis uncomplainingly. However, its selection knowledge is incomplete, a situation we expect to exist for a long time to come. The user fills in the missing pieces.

The system must make allowances for its incomplete knowledge.

Because the system's knowledge base is incomplete, we expect a user to view the system's analysis with some skepticism. The user is provided with several selection modes with which to gain various degrees of control of the selection process.

The system facilitates growth.

Experience breeds knowledge. The catalog of selection knowledge is made up of independent selection rules. This gives the system the important *additivity* property discussed in chapter 6. Further, the system monitors the actions of the user to detect its own selection mistakes; when found, it records the necessary context to allow future knowledge maintenance.

Problem solving structure is accessible.

Local selection knowledge is not enough. The selection engine requires access to 1) a method's internals, 2) the current active goal, and 3) the goal superstructure. The notion of meta-goal and meta-plan (see [Wilensky 80]) are introduced here as useful concepts.

7.1. Selection Criteria

There are several different criteria on which to base selection. Traditionally, the development of an algorithm has been based on how efficiently the final implementation runs, i.e., how much space does it use, how much time does it take on best, average and worst cases. Kant [Kant 79] describes a rule based form of such rules for developing text manipulation programs. This type of knowledge is directed at the *product* of the development process, the final implementation program. This is one type of knowledge contained in Glitter's selection rule catalog.

More recently, the development process itself has been the focus of selection knowledge (see for instance [Kant 79]). This has been brought about largely by 1) the study of non-trivial specifications and 2) the attempt to automate the development process. Current state-of-the-art machines do not have the capacity to exhaustively search all possible implementations of a given specification. Systems that use a partnership paradigm have particular problems in judging the amount of time to devote to a specific selection problem: the selection process must be geared towards returning a solution to the waiting user in a reasonable amount of time. This forces us to examine the *problem solving* efficiency associated with various development strategies. Glitter's selection rule catalog contains knowledge which attempts to estimate the effort involved in applying a method. In the next section we will see that such an estimate requires examining 1) the actions a method takes and 2) its compatibility with the overall goal structure.

7.2. The Glitter Selection Process

In this section we present first a summary and then a detailed description of one stroke of the Glitter selection engine. We will use this as the organizational basis for introducing each type of selection knowledge found within the Glitter selection rule catalog.

Selection Process Summary

1. Goal G posted. If G is satisfied in the posting state then it is marked as trivially achieved.

2. Initial method candidate set formed. Given that G is not trivially achieved, then G is activated and all methods that are indexed to G and whose filters evaluate to true are placed in an initial candidate set.

3. Weighted method candidate set formed. Weight assigning rules are run.

    a. General selection rules run. These rules use the goal context tree to assign weights to methods within the initial candidate set.

    b. Method specific rules run. These rules are indexed directly to specific methods within the initial set.

    c. Resource rules run. These rules inspect methods within the initial set for their use of problem solving resources.

    d. Weights summed. The weights provided by the general, method specific and resource rules are summed for each method.

4. Final candidate set formed.

    a. Ordering rules run. These rules provide a partial ordering on the methods of the initial candidate set. They may or may not take into account the weighted sum of a method.

    b. Weak methods culled. Any methods whose weighted sum is below a given threshold (currently 1) are removed from the final set.

    c. Any methods that are ordered after a culled method are also removed regardless of weighted sum.

    d. Methods that are unordered and have no weight (i.e., are not referenced in any of the selection rules run in step 3) are not removed.

5. Final candidate set ordered. The remaining methods are ordered by 1) explicit ordering provided in step 4a, and 2) by sum of weights. Methods about which no opinions are expressed are ordered last in the set.

6. Method chosen from final set

    a. If the system is in *cautious mode* then the user is called on at this point to select a method.

    b. If the system is in *trusting mode* then a selection will be made as follows:

        - If there is a clear winner in the final set then the system will choose it.

        - If there is no clear winner then the system will ask the user to arbitrate.

7. Actions reordered. Any action ordering rules associated with the selected method are run.

8. Method applied.

We will follow the selection process in more detail. The references to Davis are from [Davis 80].

7.2.1. The initial candidate set

The activation of a goal G causes several things to happen. First, a check is made on the achievement of the goal within the current state. If G is achieved then it is marked as such and a new goal is selected for activation. If it is not achieved then the method catalog is searched for methods that are indexed to G. If the filter of a matching method evaluates to true then it is added to the initial method candidate set (what Davis refers to as the set of plausibly useful Knowledge Sources). If this set turns out to be empty the user is informed and control reverts to him. At the same time, the context is recorded for future reference as described in the next section.

7.2.1.1. Inferring that knowledge is missing

There are two different causes for an empty initial candidate set: 1) a piece of development knowledge is missing from the method catalog, or 2) the goal G is unachievable. While the system cannot determine which is the case at this point, the user's next action provides a big clue:

- If the user next does a manual operation on the program we can infer 1, i.e., a program transformation is missing.
- If the user next posts a subgoal of G we can infer 1, i.e., a problem reduction is missing.
- On the other hand, if he next switches to an alternative problem solving context (backtracks) we can infer 2, i.e., this is a dead-end. The system's default assumption is that some missing piece of selection knowledge led to a wrong choice at some earlier point in the path.

The system helps facilitate the discovery of missing knowledge by monitoring the above events. First, the system records each occurrence of a change to some previous development state (see section 2.3.2.2). At the end of development, any states that were abandoned and not later resumed as part of the final implementation path are marked as backtracking points. States which a) were backed-up to and b) are on the final path are flagged as places where new selection knowledge may be needed.

Second, the system records any ad hoc methods created by direct user intervention, i.e., goal posting or manual transformation (see section 2.3.3.2). Such methods, after a generalization process carried out by a human analyst, become likely candidates for inclusion into the

method catalog. Note that the ad hoc method defined by the system under the primordial goal (see *initial problem solving state* in section 2.2) is included in this set. Thus, the user's high level organization of the development is always considered for inclusion as a generalizable method.

7.2.1.2. Set size, saturation, etc.

Once the initial candidate set is formed, why bother defining a further selection process? Why not simply try all methods in a breadth-first manner (see for instance, unadorned PECOS [Barstow 79a]). Davis gives one answer:

> Almost all traditional problem-solving structures are susceptible to *saturation*, the situation in which so many applicable knowledge sources are retrieved that it is unrealistic to consider exhaustive, unguided invocation.

Depending on the eagerness of the methods (see chapter 6), we can assume that the initial set will often be saturated. However, even in cases where only a few methods are competing, their individual resource costs may be large. For the same reason that we don't want to try each door exhaustively in the lady and the tiger problem, we want to avoid getting eaten up following non-optimal methods. This is particularly important given the assistant role that the system plays, i.e., where development is interactive. With a human user in the loop, the system must become resource conscience in several ways: 1) once the user passes off a task to the Glitter assistant, he must wait for Glitter to come back before moving to the next task, i.e., Glitter cannot be "gone" for arbitrary lengths of time, and 2) the user may need to get involved with lower-level problem solving in much the same way the user/physician was needed in MYCIN [Davis 77] -- to supply information unavailable or uncomputable by the system. The latter case is particularly troublesome since, with an exhaustive search, it requires the user to answer a set of questions, most of which are likely irrelevant to the final choice.

7.2.2. The weighted candidate set

Given the need for a selection or refinement process, Glitter's next step is to apply its knowledge about the applicability of certain methods in a given development situation. The general form this knowledge takes is as follows:

```
Selection Rule <unique name>
        IF: [<selection expression>]1
        THEN: <weight>
            [optional comments]
End Selection Rule
```

The fields of a selection rule are broken out as follows:

<unique name> - provides a unique textual handle and is intended to give a short description as well.

<selection expression> - in general, some problem solving event such as a method joining the initial candidate set or a goal becoming active. For details, see the particular classes of rules that follow.

<weight> - a weight in the set {-5, -4, -3, -2, -1, 1, 2, 3, 4, 5} to be attached to a specified set of methods.

We further divide selection rules into three classes:

1. *Method Specific Rules*. These rules have as a <selection expression> the inclusion of a particular method within the initial set. Each rule captures some piece of knowledge about the usefulness of a specific method in the method catalog.

2. *General Rules*. These rules have as a <selection expression> some situation involving the wider problem solving context. In particular the rule will access a portion of the goal tree beyond the current initial set, e.g., super-goals activated, super-methods applied. Further, they generally reference methods within the candidate set by content as opposed to name. More on this later.

3. *Problem Solving Resource Rules*. These rules reward candidate methods that avoid what are known to be costly actions. Again, reference is on method content as opposed to name.

We will look at examples of each type.

7.2.2.1. Method-specific rules

The selection rule *Anchor1b[35] is a method-specific rule:

```
| SelectionRule  *Anchor1b                          |
      IF  a) Anchor1 is a candidate
          b) Y|RANDOM
      THEN   +5
| End Selection Rule                                |
```

It is keyed to method Anchor1:

```
| Method  Anchor1                                   |

          Goal: Equivalence X and Y
          Action:  1) Reformulate Y as X

        [Try changing the second construct into something that matches the first.]
| End Method                                        |
```

The first <selection clause> of *Anchor1b is the inclusion of a specific method (Anchor1) in the initial set. The second <selection clause> checks on further details of the method, namely whether Y is bound to RANDOM, a particular type of event in Gist. If so then the <selection action> is to give a large weight to the method: it is always possible to reformulate a random event into a more specific event.

Let's look at another method-specific rule keyed to Anchor1.

[35] Method-specific rules are given names starting with * to differentiate them from the corresponding method name.

```
| SelectionRule  *Anchor1c                              |
      IF  a) Anchor1 is a candidate
          b) Y|derived-relation-reference
          c) Definition of Y reformulatable as X
      THEN    +2
| End Selection Rule                                    |
```

The first two clauses are similar to those of *Anchor1b. However, the third clause introduces a new wrinkle. Since Y is bound to a reference of a defined object, and it is known that defined objects can often be unfolded in place of their reference, this clause asks whether the defined object associated with the reference Y looks like it can be reformulated as the expression bound to X. In other words, the rule hypothesizes that the body will be unfolded and attempts to look ahead and see how successful that unfold will ultimately be. This is the focus of the rule. However, carrying out the necessary analysis presents a problem: the system currently has no general means to analyze the likelihood of reformulating an expression E1 into an equivalent expression E2[36]. We must rely on the strength of the partnership here, *i.e.*, call on the user to supply missing information. The following question is asked of the user:

Can <definition of Y> *be reformulated as* X?

where <definition of Y> and X are printed as their bound values. Note that the system carries out as much of the computation of the rule as possible. That is, given that the focus of the rule is to check whether Y can be unfolded into something like X, the system 1) checks to see if Y is unfoldable, 2) gets Y's definition and 3) calls on the user for the reformulation information. By gathering together the information in 1 and 2, the system avoids questions such as

Can RANDOM *be unfolded as* X?

Can the body of the definition associated with Y *be unfolded as* X?

The first is plain silly, the second simply bothersome (the user must search for the definition associated with the reference Y).

---

[36] Rule *Anchor1b above embodies knowledge on reformulating a special expression, a random event, into some other expression.

7.2.2.2. General selection rules

DemonsAreGood is a general selection rule:

```
| SelectionRule  DemonsAreGood                         |
      IF  a) Goal/Supergoal is Map X
          b) Method M reformulates X as a demon
      THEN   +1
      [Demons are generally easy to work with.]
| End Selection Rule                                   |
```

This rule captures empirical knowledge built up from our development of the package router program presented in appendix C. In particular, we found that during this development, the reformulation of various Gist constructs (e.g., constraints, derived relations) into demon form facilitated future optimizations. However, we give it a relatively small weight because of the small experience base it is derived from; we are not convinced that the catalyzing effect of demons will carry out of the package router domain or domains like it. What might be necessary to give the rule a stronger weight is another antecedent clause of the form

c) the problem domain has features F1...Fn

where the features F1...Fn may or may not be computable by the system.

An important point to notice about the rule DemonsAreGood is its indirect reference to methods by the actions they take. That is, instead of naming methods to reward as in the method-specific rules, the actions of methods within the candidate set are analyzed for the actions they propose. Hence, if a new method is added to the catalog that reformulates some construct as a demon, it will automatically be rewarded whenever competing. Davis refers to this ability of looking inside knowledge sources (methods) to glean control (selection) information as *content reference*.

In regards to content reference in Glitter, the system can analyze both the individual filters and actions of a method. This includes examining the arguments of individual goals. However, program transformations, as found in *Apply* actions, reflect calls on Lisp code whose effects are unanalyzable by the system. For example, the general selection rule SubComponent,

```
| SelectionRule  SubComponent                      |
      IF  a) Goal is Reformulate X as P
          b) pattern-match[Y, P, X]
          c) Method M extracts Y from X
      THEN   +2
| End Selection Rule                               |
```

asks whether a method M extracts[37] a component Y of a compound structure X. Since there exists no corresponding goal for the extraction task (see however section 5.5), we can assume that M's action is to apply an extraction transformation (as opposed to an *Extract* goal). There are several ways the system can get at the necessary information about this transformation: 1) define an *effects language* of transformations and require the transformation writer to augment his code with the relevant descriptors from the language, 2) better yet, interpret the effects language directly, doing away with the Lisp code, or 3) ask the user. The first two remain in the domain of future work, the third is current practice.

7.2.2.3. Problem solving resource rules

ReadyToGo is an example of a problem solving resource rule:

```
| SelectionRule  ReadyToGo                         |
      IF  a) M|method is a candidate
          b) forall actions A of M either
                     1) A is an Apply or
                     2) A is achieved trivially
      THEN   +1
      [if only apply goals left then cheap choice]
| End Selection Rule                               |
```

This rule notes that if the only actions a method will take if chosen will be to apply transformations then it may be worthwhile in terms of problem solving costs to give it a try. Note that Glitter is able to compute the antecedent clauses of this rule without help from the user.

---

[37] The meaning of extract here is that of destructive replacement: overwrite a compound structure with one of its components.

After the method-specific, general and resource rules have been run each candidate method will be augmented with weighted opinions of the form

(Weight RuleName)

At this point, a method's weights are summed and the total is attached to the method.

7.2.3. Final Candidate Set

We are now ready to apply ordering knowledge to the candidate methods. Selection ordering rules, found at the end of each section in the Selection rule catalog (appendix G), have the following form:

```
Selection Rule <unique name>.
        IF: [<ordering expression>]^1
        THEN: [<ordering action>]^1
            [optional comments]
End Selection Rule
```

The fields of an ordering rule are broken out as follows:

<ordering expression> - the inclusion of one or more specific methods within the candidate set. May be modified to include the goodness of a method as provided by the weighting rules, e.g., "if method Foo is a *good* candidate ..." where *good* is defined as a function of total weight[38]. Besides the reference to specific competing methods, other clauses may reference properties of the program or goal tree.

<ordering action> - the selection ordering of a set of candidate methods before or after another set of candidate methods.

We will look at some examples of ordering rules.

---

[38]The system currently defines *good* as a total weight greater than 1.

```
| SelectionRule  MapDR1a                                              |
      IF  a) StoreExplicitly is a good candidate
          b) (number of refs * recompute cost) is more costly than
                 number of explicit insertions
      THEN   StoreExplicitly > UnfoldDerivedRelation
| End Selection Rule                                                  |
```

Glitter computes the number of references to the relation and relies on the user to supply estimates on 1) the cost of computing the relation on demand, and 2) the number of explicit insertions necessary if storing the information explicitly. The power of the partnership approach is illustrated well here: the system focuses on the right questions; the human provides domain specific knowledge about branching frequencies and an estimation of computation costs. Each piece of information is unavailable to the system, the former because it is domain specific and the latter because the system lacks a sophisticated analysis model.

The action of the rule asserts that StoreExplicitly should be chosen before UnfoldDerivedRelation.

The results of running the ordering rules (and taking the transitive closure) is a partial selection ordering on the weighted candidate set[39]:

```
M1 > {M2,M3} >...Mn
```

The next step is to cull the candidate set of unpromising methods (what Davis refers to as knowledge source *refinement*). Any method whose total weight is below the goodness value is removed from the set. Any method that is ordered after a removed method is in turn removed. All methods remaining form the Final Candidate Set. If this set is empty, one of several things may be the cause: 1) some piece of development knowledge is missing, 2) some piece of selection knowledge is missing, 3) the goal is unachievable. As with an empty initial candidate set, the system examines the user's next action: 1) if the next step is manual the system records the step and notes a potentially missing method, 2) if the next step is to select a method from the initial candidate set the system records the method chosen and notes a potentially missing piece of selection knowledge, 3) if the next step is to switch to another

[39] The system flags inconsistent orderings.

problem solving context the system notes the change and, at the end of development, records locations among which a wrong branch was taken.

7.2.4. Final Set ordered

Any methods not ordered explicitly by selection ordering rules in the previous step are ordered by relative weight: if method M1 has a higher total weight than method M2 then M1 is ordered before M2. Unordered methods with equal weights remain unordered. Un-imputed methods, i.e., methods that have no weight associated with them, are ordered as a set, last.

7.2.5. Method chosen

The user can place the system in several selection modes (see also critic mode, section 7.2.7):

- Trusting mode. The system selects the highest ranked method for application. If no one method emerges as a best choice, the user is called on to make the selection. The context and the user's selection is recorded as a point where possible further discrimination knowledge is needed.
- Cautious mode. The system presents the final ordered candidate set to the user. The user selects the method to apply. If this method is different than the one that the system has ranked first, a record is made of the discrepancy.

Note that there are cases where forcing the user to arbitrate ties is a very conservative choice, i.e., when a subset of equally weighted methods are all relatively good. In cases such as this, the system could instead try an automatic approach by either carrying each method in the subset along for some specified time to pick up more information or picking one and backtracking to this selection point on failure and choosing another. There are two reasons that techniques like these have not been incorporated into the selection process. The first is quite pragmatic: they have yet to be needed. That is, there generally is a marginal winner which also turns out to be an adequate choice. We view this not as an insightful find that in Gist developments there are always clear cut best methods, but that our current selection rules are over-tailored to the small set of developments that we have tried.

Secondly, handling conditional or backtracking control intelligently requires a new problem solving vocabulary and knowledge base, an effort, we argue in section 7.5, best left as future work.

7.2.6. Method actions ordered

The independent actions of a method are given a default ordering by the method writer. For example, the first two actions of method MergeDemons

```
| Method  MergeDemons                                                  |

        Goal: Consolidate D1|demon and D2|demon
        Action:  1) Equivalence trigger-of[D1] and
                                trigger-of[D2]
                 2) Equivalence var-declaration-of[D1] and
                                var-declaration-of[D2]
                 3) Show MERGEABLE_DEMONS(D1, D2, I|ordering)
                 4) Apply DEMON_MERGE(D1, D2, I)

      [You can consolidate two demons if you can show that they have the same
      local variables, the same triggering pattern and that they meet certain
      merging conditions.]
| End Method                                                           |
```

can be achieved in either order. However, it is usually easier in terms of problem solving effort to achieve the equivalencing of the two triggers first. Hence, the method writer orders this goal first. In cases where it is easier to achieve the equivalencing of the demon's variables first, action ordering rules can be used to reorder the actions. In this particular example, it is easier to achieve the second action first when the triggers differ only in variable naming. The action ordering rule TriggersAlmostEquiv represents this information:

```
| SelectionRule  TriggersAlmostEquiv                                   |
      IF   a) MergeDemons is selected
           b) Triggers differ only in variable renaming
      THEN   action-2 > action-1
      [The first goal will fall-out as side-effect of second.]
| End Selection Rule                                                   |
```

While action ordering rules are local to a method, we place them as part of the selection process to provide control over when they are computed, i.e., instead of being computed on triggering as a method's filter is, they are computed only on need, when the method is selected.

After all action ordering rules are run, the method is applied.

7.2.7. Critic Mode

Besides placing the system in trusting or cautious mode, the user can choose to go into *critic mode*. In this mode, once the initial candidate set is formed, control is passed back to the user. From this point the user can either choose a method from the initial set or request information on a specific member M. From the system's point of view, this is the same as an initial set containing the single member M. The system runs selection and ordering rules on M and reports the results back to the user.

The purpose of critic mode is to allow the user to take over the selection process, calling on the system to critique selected methods in cases where the computation is complex or detailed. While the critic mode is likely to provide a significant speed-up in response, it has the same disadvantages as calling a method directly (see section 5.6.3): documentation is lost; there is a chance the user will be unaware of or forget to apply some piece of selection knowledge cataloged by the system.

7.3. Program features used in selection

In this section we present the program features which have been useful in making selection decisions in the package router development. Note that none of this knowledge is particular to the router domain. This is not to say that such domain specific knowledge has no place in the selection catalog, only that more general selection knowledge was found sufficient in our particular development.

Non-deterministic Control

There are two basic strategies for mapping constrained non-deterministic control: 1) at each location where a constraint may be violated insert conditional *backtracking* code to undo the current choice and generate a new one or 2) at each non-deterministic choice point, *predict* what choices will lead to constraint violation and avoid choosing them (or vice versa, choose one that meets all constraints). Infrequently, both strategies may be possible in a given domain; more often only one is applicable. The following knowledge is useful in choosing between the two:

- It may be (often is) the case that one of the two strategies is impossible or at least intractable in the problem domain. As an example, in step 4.1 of the router development two methods are competing: UnfoldConstraint, corresponding to the backtracking strategy; MapConstraintAsDemon, corresponding to the predictive strategy. The related selection rules, *UnfoldConstraint and *MapConstraintAsDemon, ask the user the following questions:

  Is it possible to backtrack on violation of
  DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE?

  Is it possible to predict violation of
  DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE?

  We rely on the user to notice that 1) once a package is at a switch and the switch is set wrong there is no means to move the package back out of the switch and reset it (backtrack) and 2) it is possible to avoid violating the constraint by choosing the right switch setting before the package reaches the switch (predict). The consequence of this is that the backtracking strategy is ruled out while the prediction strategy is rated high.

- Given that a predictive strategy has been chosen to map a particular constraint C, we must next choose an event E suchthat whenever event E occurs, we make sure that the constraint C will not be violated. If the constraint is conjunctive, the event E can often be most easily chosen as a conjunct arm. That is, whenever one arm of the constraint becomes true, guarantee that the other arms don't. Choosing which arm to select is left to the user:

  Which arm of <conjunction> *is a useful predictor*?

  In step 4.2 of the router development, the question becomes

  Which arm of

```
▶1  (package:LOCATED_AT = switch
     and
▶2   SWITCH_SET_WRONG_FOR_PACKAGE(switch,package)
     and
▶3   ((package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
       and
      SWITCH_IS_EMPTY(switch)) asof everbefore));
```

  is a useful predictor?

  Both arms $\blacktriangleright_1$ and $\blacktriangleright_2$ are of the idiot light variety: when they are on (true) it's too late. The third arm $\blacktriangleright_3$ is the right choice: when a package becomes the first one due at a switch and the switch is empty, set the switch correctly.

Demons

Demons provide a powerful specification abstraction. Moreover, they have been found to be a equally powerful development aid, as recorded in the selection rule DemonsAreGood. The use of demons as a type of intermediate mapping form allows other demonic development strategies to be employed. For instance, the demon NOTICE_NEW_PACKAGE_AT_SOURCE is an intermediate mapping of the derived relation PACKAGES_EVER_AT_SOURCE (see step 1.11), the demon SET_SWITCH_WHEN_HAVE_CHANCE an intermediate mapping of the constraint DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE (see step 4.1). In both cases, a strategy of consolidating the intermediate form with an existing demon leads to further optimizations. Opposite of consolidation, splitting a demon into separate cases can lead to divide-and-conquer strategies. Below is the knowledge found useful in dealing with demon-manipulating methods:

- Knowing whether a demon is within the implementable portion of the spec or triggers on an observable event can be useful in determining its utility as a consolidation partner (see selection rules *MapByConsolidation1, *MapByConsolidation4)
- Recognizing complex or abstract triggering events can lead to the selection of divide-and-conquer methods (see selection rules *CasifyDemon, CasifyComplexConstruct).
- Recognizing non-deterministic triggering events can be useful when attempting to map away constraints (see selection rule *MapByConsolidation2).

Relations

There are several actions we might attempt on a relation R: we can attempt to remove R entirely from the specification; we can attempt to map R into an operational form. The relation-manipulating methods which deal with these actions are rated using the following criteria:

Relation removal: There exists a straightforward method for removing a relation R, namely RemoveRelation. This method is unfiltered, meaning a) it can be applied to any relation, and b) determining its likelihood of success falls on the shoulders of the selection rules. The necessary selection criteria is as follows: the likelihood of removing a relation R depends on how R is used within the specification.

1. If R is never referenced then it can be removed trivially.
2. If R is a composite object (e.g., sequence) then we may be able to remove

(actually replace) R by showing that only certain attributes or elements of the object are ever referenced.

3. Otherwise, if R can be shown to be acting as an intermediate place holder for some other relation Q then with suitable value replacement, R can be removed.

The detection of case 1 is straightforward and is handled as part of the more general case of choosing methods that are cheap to apply (see rules ReadyToGo, CheapRemove). Noticing that only certain attributes of an object are ever referenced is also straightforward and is handled by *RemoveRelation3 (see step 3.1). However, noticing that only certain elements of a set or sequence are used can be a more difficult proposition. This analysis is currently left to the user by rule *RemoveRelation1. After it is determined that R's argument is a sequence, the following question is asked of the user:

Is only one element (i.e., first, last) of <sequence> ever referenced?

In step 1.1, the sequence in question is **PACKAGES_EVER_AT_SOURCE** and the answer is yes (the last). Further work towards removing the sequence **PACKAGES_EVER_AT_SOURCE** requires noticing various features of the sequence's construction: the ordering of the objects (packages) of the sequence relative to some event (their entry into the router). Or equivalently, whether new objects are appended or prepended to the sequence (they are appended). This information helps select which element of the sequence to focus on during the removal process.

Finally, case 3 is based on noticing whether R is a "temporary variable":

Is R acting as an intermediate place holder for some other relation? If so, which? "

In the case of step 2.1, the answer is yes, for relation **LAST_PACKAGE**.

Relation mapping: Given a derived relation R, there are several mapping strategies available: compute the relation on demand at each of its reference points; maintain the relation explicitly; some combination of the two (e.g., memo functions). Selecting among these strategies relies on the various resource costs associated with computation and maintenance. Currently the system can do some of the low-level computation automatically, e.g., counting the number of references to a relation R in simple cases. A more hefty part is left to the user. We give some of the selection analysis associated with mapping relations below.

- In choosing among explicitly maintaining a derived relation or computing it on demand, we must estimate, among other things, the recomputation cost. This estimation is left to the user in the form of the following question:

  The cost of computing relation <name> is what (low, medium, high?)

  The range of the cost has been broken into three discrete classes from which the user must choose. For example, in step 5.5 we would expect the user to answer "low" while in step 5.1 we would expect an answer of "high".

- Given that a derived relation is static, one method for mapping it is to explicitly store a separate relation for every one implied by the definition. When deciding on the cost of such a mapping strategy, an estimate of the number of explicit relations required is necessary. We rely on the user to supply the estimate. For example, in the router development the relation LOCATION_ON_ROUTE_TO_BIN implicitly computes the reachability matrix for the physical router network. The user is asked to supply the number of relations (non-zero entries) needed to explicitly store the matrix.

  How many relations must be inserted to explicitly store LOCATION_ON_ROUTE_TO_BIN?

7.4. Problem solving features used in selection

In the previous section, we focused on *program* features useful in the selection process. Here we will look at some of the *problem* features which can be used in selecting among competing methods. In particular, we will be using the problem solving context in terms of the superstructure built on top of the current candidate set. Figure 7-1 provides a graphic description of the goal/method tree. This entire tree is available for analysis by the selection rules. Given that we are selecting on some method candidate set S, the following information is available:

- $▶_1$ The individual methods of the candidate set S.
- $▶_2$ The immediate goal G_{Imm} producing (triggering) S.
- $▶_3$ The brothers of G_{Imm}, both previously achieved and currently pending.
- $▶_4$ The method M_{Sup} producing G_{Imm}.
- $▶_5$ The selection rules firing and participating in the posting of G_{Imm}, i.e., the reason M_{Sup} was selected.
- $▶_6$ The transitive relation of all of the above, i.e., the goal G_{Sup} which triggered M_{Sup}, G_{Sup}'s brothers, the method producing G_{Sup}, why it was selected, etc.

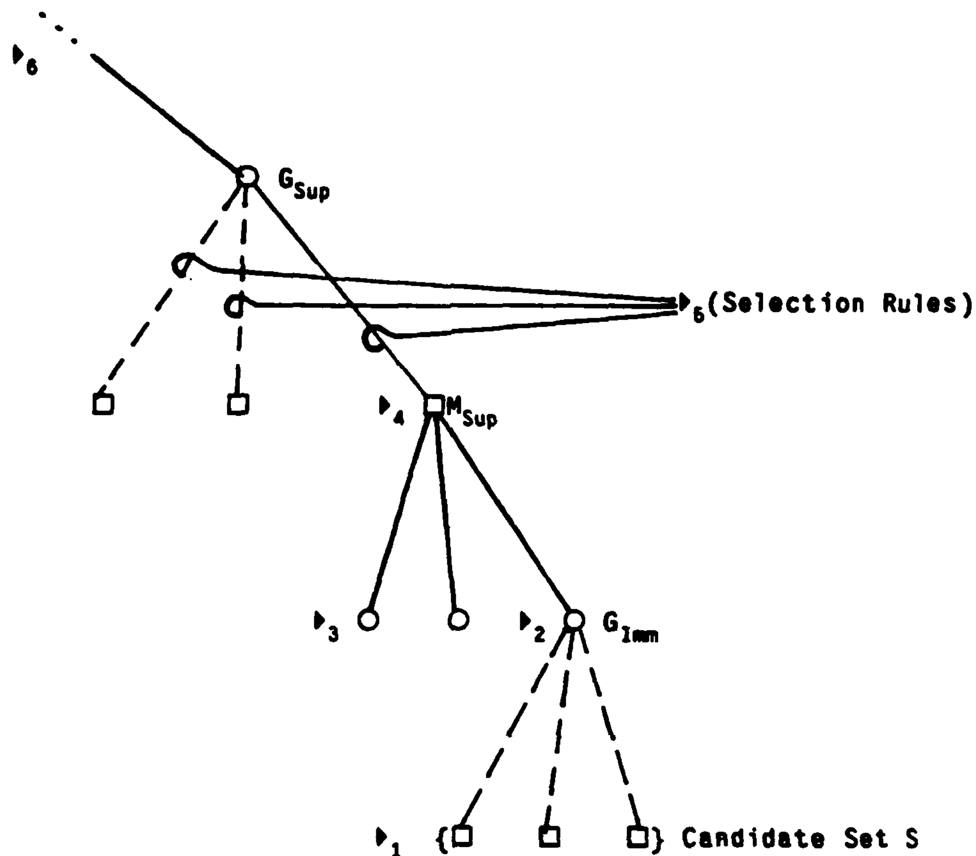


Figure 7-1: Goal context available to selection process

7.4.1. Method Inspection

As we have described earlier in section 7.2.2.2, the action fields of a method are open to inspection by the selection process. We may use this information to determine the trivial achievement of one or more of a method's goals. This allows us to make selections based on the following criteria:

- In the absence of a single method becoming a strongly supported candidate, we fall back on how cheap in terms of problem solving a method is to apply. A method is viewed as cheap if

  1. The only actions not achieved trivially at triggering time are *Apply* goals, i.e., the method's only effect will be to apply transformations. Especially high marks are given to methods that cheaply achieve *Remove* goals.

2. One or more of the method's actions are trivially achieved, a weaker case of 1.

- A set of methods may be (currently) ungeneralizable and hence, broken out into specific cases. Instead of restricting the methods triggering by placing special case checks within their filters, we generally prefer using sub-goalable restrictions, i.e., goals placed within the method's action set that jitter the program into the special case required (see property 3, chapter 6 for further discussion). However, in such cases we want to give high marks to a method that meets its jittering subgoals trivially. In most instances, this has the same effect as using filtering restrictions. However, it is more general in that it allows alternate selection knowledge to intervene and possibly rule out the method on other grounds.

  A good example of such a set of methods is the set associated with casifying a local constraint. Each method expects the constraint to be in a particular form. If it is not, its first action is to *Reformulate* it into the correct form. RequireReformUnnecessary is the selection rule which notices when one of these methods is triggered and no reformulation is necessary. It rewards such a method accordingly.

We may also be interested in the effects a method has when applied. For instance, the selection rule DemonsAreGood rewards methods that introduce demons as intermediate forms in pursuit of mapping goals. Selection rule ReformAsExtreme rewards a method that attempts to reformulate a sequence retrieval as either a retrieval on the first or last element.

Many of the method inspection rules are only one level deep. That is, they do not attempt to reason beyond the actions of the competing methods. This means we must generalize what we know about certain actions without relying on the details of those actions. Therefore, we punish reformulation goals in general (we know they are often hard) without checking to see if the particular reformulation under analysis is easy.

There are several schemes for providing better analysis of a candidate method's actions. We could try applying the method (and other of its competitors) and follow the consequences to some specified depth. Kant does a form of this in her rule selection system [Kant 79]. In a very practical sense, this allows us to view the method's affect on problem solving. As an alternative, we could attempt to reason abstractly about an action. For instance, if a method's action will be to reformulate some expression E into P then we can look at E for clues on the likelihood of success:

- A connection matrix might be defined for expressions, i.e., what expressions can

be transformed into what other expressions (Mostow discusses the use of such a connection structure in his rule-based system). Cross indexing E and P would give us some estimate of the likelihood of reformulation (and possibly an abstract plan for carrying it out).

- If E is a reference to a defined object D, we can check if D contains something that might match P. If so, we know we can unfold D to get at it.

Currently, selection rules exist for doing specialized reasoning of the above form, but rely on the user to fill in information. To remove the reliance on the user requires adopting a more general and powerful analysis approach similar to the ones discussed above. We view this as a large enough effort to be considered as part of future work.

7.4.2. The Goal Tree

A large part of the Selection Rule catalog consists of method-specific rules. These rules trigger on a particular method becoming a candidate and use the *local* context of the method to analyze its potential effectiveness. There are cases, however, when we need a bigger if not the big problem solving picture. We discuss below the ways in which Glitter uses the goal tree in the selection process.

7.4.2.1. Goal-specific knowledge

Sometimes we can generalize a set of method-specific rules into one or more goal-specific rules. That is, we can define selection criteria based on a goal as opposed to a method. This knowledge usually comes in the form "*If goal G is posted then methods that have property P are useful (unuseful)*". Clearly, we can unfold this knowledge onto each method indexed to G. However, placing it on a goal facilitates the addition of new methods indexed to G; they are automatically included in the analysis set. As a simple example, the knowledge that "*The reformulation of the relative retrieval of an element of a sequence to a positional retrieval (first, last, Nth) is most often successful when the position chosen is an extreme (first, last)*", is embodied in the rule ReformAsExtreme. In previous versions of the catalog, this knowledge was spread over the two methods ReformulateRelativeRetrievalAsFirst and ReformulateRelativeRetrievalAsLast. Although it is unlikely that any new methods will be added that will be effected by ReformAsExtreme, it is possible that the value of the knowledge will change, e.g., experience may show that the "extreme" strategy is successful n% of the time, causing the value to increase as n increases. Such rule maintenance is most easily carried out when selection knowledge has been localized to the greatest degree.

7.4.2.2. Meta-goals

As in any problem solving domain, it is sometimes difficult to select among competing actions without knowing the overall goal or goals being pursued. Wilensky [Wilensky 80] defines the notion of a *meta-goal* to describe properties that we wish to hold during the planning process and a *meta-plan* as an action we can take to achieve a meta-goal. Glitter uses the goal tree to detect supergoals (i.e., ancestors of the current goal) which become easier or harder to achieve with the selection of certain methods. In Wilensky's paradigm of meta-planning, Glitter's use of the goal tree could be expressed by the following two meta-goals:

> Meta-goal 1: Avoid choosing (weight negatively) plans (methods) which cause other goals to become more difficult to achieve.
>
> Meta-goal 2: Choose (weight positively) plans (methods) which cause other goals to become easier to achieve.

These are actually a cross between several of Wilensky's meta-goals, including "Don't waste resources", "Achieve as many goals as possible", "Don't violate desirable states". Glitter has no explicit representation for meta-goals and only represents the above two implicitly. Likewise, there is no explicit representation of meta-plans. However, certain rules do implicitly implement goals of the above type. One such rule is MapSubOfRemove. This rule recognizes situations where there exists a supergoal of removing some construct from the program and a subgoal of mapping it. It rewards a method which avoids scattering the construct throughout the program and punishes one that does. In a slightly more recognizable domain, suppose that Johnny wants to haul (remove) a bag of tin cans to the dump[40], but the bag is too big to fit in his wagon. He has several (mapping) options: 1) stomp the bag until the cans are flattened to an acceptable size, 2) take the cans out of the bag and haul them in acceptable size subsets to the dump, using repeated trips. Johnny, being a bright meta-planner, chooses the first.

---

[40] In the dark ages before recycling centers.

7.5. Extending the Rule Catalog

Adding new selection knowledge is straightforward if the knowledge can be encapsulated in one of the forms of selection rules we have defined. As we have seen, the current form of our rules is useful for capturing knowledge about competing methods and their application. However, there are other types of selection knowledge that users employ in a development which are not addressed by Glitter. First, there is the case of dead-end solution paths, i.e., states where the current goal cannot be achieved. The user must choose a new state in the problem solving tree, kept by the system, from which to continue the development. If the system is to get involved with this backing-up process, several things must happen. One, a *problem solving* goal language must be defined. The goals here reference not a *program* but a *development*. We might expect goals of the following form:

Back-up to a state before a mapping decision was made on Foo.

Back-up to a state where Foo is in demon form.

Help![41]

Two, backtracking-methods must be defined for achieving the corresponding goals. For example, using our current method formalism, we might define the following:

```
| Method  TryAlternativeMethod                              |

          Goal:   Help!
          Filter:  a) Current active method is M1
                   b) Next best alternative of M1 is M2
                   c) M2 is not too bad
          Action:  1) Back-up to state where M1 was chosen
                   2) Choose M2

       [Maybe another method will work.]
| End Method                                                |
```

Three, given a collection of such methods, selection rules must be defined for choosing among them:

---

[41] i.e., *Back-up to a state from which I can continue.*

```
| SelectionRule *TryAlternativeMethod                                   |
      IF  a) TryAlternativeMethod is a candidate
          b) difference between weight of M1 and weight of M2 is small
          c) no ordering rule explicitly ordered M1 before M2
      THEN   +2
      [Choice looks somewhat arbitrary; M2 could be just as good.]
| End Selection Rule                                                    |
```

The point here is that the same mechanism used for defining development methods and associated selection rules looks, at least initially, capable of representing this new type of problem solving knowledge.

A second type of knowledge which we have not addressed in Glitter is the large scale organizational knowledge a user employs in producing a development. As an example, what knowledge does the user employ in choosing 1) the first steps of each section of the router development, and 2) the particular order that they occur? Why choose to map PACKAGES←EVER←AT←SOURCE before mapping PACKAGES←DUE←AT←SWITCH? Why bother mapping LAST←PACKAGE at all? We currently rely on the user to provide the insight to answer these questions, and hence supply the top level organization of the development. Being a helpful partner, we might expect Glitter to field the following type of questions from the user:

1. *Will mapping Foo before Fum cause problems?*

2. *What should I do next?*

3. *Please develop the rest of the program.*

Request 1 requires some form of hypothetical reasoning and in its full generality is a very difficult question to answer. However, it is likely we can generalize some of this ordering information in rule form (see for instance, Kant's plausible-implementation rules).

Request 2 is actually subsumed by request 3. In request 2, the user is not ready to relinquish control of the development, but simply wants some guidance on what he should focus on next.

Request 3 is more of a case of clean-up: "I've supplied the interesting steps, now you wrap it up". Of course this request given with only a small portion of the development complete must rely on some powerful clean-up mechanisms.

While we assume that some types of organizational knowledge can be generalized in rule form, we suspect that much of such knowledge is domain and even development dependent. That is, we cannot define a strategy "always map constraints before demons" across specifications. Further, our small data base of development examples prohibits us from more specific rules of the form "map constraints before demons when P is true"; identification of conditions like P will require more experience with Glitter developments. Given this, the user must continue to provide high level guidance and the system must be prepared to support him in the ways mentioned above. Providing the right set of support tools we view as future research.

7.6. Hearsay-III Rule Representation

Glitter selection rules are implemented as *scheduling knowledge sources*, or simply SKS. A SKS has the same form as a KS. The difference is that an SKS triggers on *changes to the scheduling blackboard*, a structure which holds activation records of triggered methods. In Glitter, as each method triggers on a particular development goal (represented by a goal-unit on the domain blackboard), its corresponding activation record is placed in a candidate set on the scheduling blackboard. When all such methods have triggered, the set of scheduling rules (as represented by SKS) are run. Their effect is to order the competing activation records.

7.6.1. The Scheduler

Hearsay provides both a simple default scheduler and hooks for defining more sophisticated application-specific schedulers. The Glitter scheduler, a form of the latter, provides for the overall control of the system. It sequences through goal posting, method triggering, SKS execution and method selection. It implements the various faith modes and drives the user interface.

Chapter 8
Related Work

People are usually more convinced by reasons they discovered themselves than by those found by others.

-- Blaise Pascal

In this chapter, we will look at research that either has influenced our work or that has interesting similarities. Most generally, we will be looking at software development systems. More specifically, we will be interested in development systems which are either knowledge-based, use problem solving or provide an interesting interaction between user and machine.

Many of the development techniques proposed by the systems that we discuss are of the hypothetical or future work variety. This encourages us. It appears that Glitter addresses problems recognized as important by other research efforts in program development. While we find it instructive to compare Glitter to these proposals, we are lead to an admonition, "A running system in the hand is worth n hypothetical scenarios in the bush".

8.1. The PSI system

The PSI system integrates a set of expert modules into a single software development system [Green et al. 79]. The modules include a trace expert [Phillips 77], a model building expert [McCune 79], a domain expert, a discourse expert [Steinberg 80], a coding expert [Barstow 79c] and an efficiency expert [Kant 79]. The usage scenario is for the user to describe his problem to the system in an interactive fashion, using English and execution traces. Using consultation with other modules, the model expert creates a high level procedure (called a model) by using the discourse, domain and trace experts to extract the necessary problem information. The model is passed to the coding expert for refinement into a Lisp implementation. The coding expert calls on the efficiency expert for advice on optimal

implementation decisions. We will look at the bottom end of this process[42] -- the PECOS/LIBRA modules [Barstow & Kant 77] -- in more detail.

There are several similarities between the Glitter and PECOS/LIBRA systems: both start with a formal, abstract specification; both use a rule-based refinement (mapping) paradigm; both have a selection engine for choosing among competing tasks. There are differences as well:

- The formal specification languages of the two systems are at different abstraction levels. The PSI system makes *control* design decisions before the coding expert is called. Hence, the model language specifies the basic algorithm structure and need not contain the abstract control of Gist, e.g., inference, demons, constraints, non-determinism. One of the foundations of Gist is that no implementation need be ruled out; refinement of Gist's abstract control is left to the transformation phase.

- PECOS uses no explicit goal structure. A limited form of sub-goaling is achieved through the QUERY feature. However, the basic search is one of running rules in a forward chaining fashion. The effect is to produce a decision tree where the leaves are final implementations. LIBRA allows PECOS to control exploration of this tree. When a decision node is reached, LIBRA is called to analyze one or more of the alternatives and choose one as best. This automatic, non-backtracking search rests on uniformly cheap, non-deductive monotonic refinement: each rule refines an abstract structure into a more concrete structure; no deductive machinery is needed to prove rule preconditions; the cost of applying a rule is of no consequence. In Glitter, no such constraints are placed on development methods, forcing the system to deal with deduction, backtracking and possibly infinite paths. In many cases, the user is called on to supply the needed information or control.

  It should be noted that Barstow discusses the need for more sophisticated problem solving techniques in a later paper [Barstow 79b]). In particular, he argues that as specifications become less algorithmic (more abstract), refinement rules with non-trivial pre-conditions will be required as well as a theorem prover to verify them. He also argues that controlling search will become important as the cost of applying a rule increases and the avoidance of backtracking becomes difficult.

- LIBRA incorporates the following types of selection knowledge: heuristic rules for choosing among competing implementations; formal techniques for analyzing the concrete computational complexity of alternative implementation choices; heuristic rules for organizing the refinement process in general and its resources in particular. Glitter incorporates the same knowledge with two differences: 1) the

[42] In both PSI and SAFE, the top end process of acquiring the problem description in English has been limited to small problems.

formal analysis while machine guided is basically user supplied, 2) selection knowledge must exist for making choices which will effect both overall development strategies as well as tactical program transformations.

In summary, the abstraction found in PSI's model language is at a low enough level to be refined automatically by a catalog of approximately 400 refinement rules. PSI, therefore, puts the burden on the problem acquisition modules of translating the English description into a low enough level to be encoded in the model language. PSI's model language is at about the same level as the final program produced by Glitter; we foresee PECOS and LIBRA-like systems acting as "compilers" on Glitter output.

8.2. The CHI system

The CHI system is a more recent attempt to use some of the PSI technology in building a knowledge-based programming environment [Green et al. 81]. Instead of a system of autonomous experts, CHI proposes a homogeneous collection of tools all sharing a common data base of program objects, operations and refinement rules. The synthesis portion of PSI, PECOS and LIBRA, have been incorporated into the CHI system. The major change has been the definition of a base language "V" that a) replaces PSI's model language, i.e., it is used for program specifications in CHI, and b) describes the system's data base and tools as well. This form of representational transparency allows the user of CHI to query all parts of the environment in a uniform way. Of course, this extends to the individual tools as well. At least one tool in the CHI environment, the rule compiler, has been implemented using CHI itself.

8.3. The Programmer's Apprentice

The Programmer's Apprentice system (PA) is both knowledge and partnership-based [Rich & Shrobe 78, Rich et al. 79]. The PA functions in two ways. First, given a Lisp program, it can *analyze* the underlying structure of the program and *recognize* it as a structure composed of known program building blocks [Waters 78]. Second, given an abstract or incomplete description of an algorithm, it can help the user fill in details and debug incorrect portions. The final result is a working Lisp program and a layered description of the program moving from abstract to concrete.

The Programmer's Apprentice consists of several related research efforts. Rich [Rich 81] has

built the knowledge-base portion of the PA around what is called a *plan*, the basic representation of programming structures. A plan consists of a network of ported operators linked by data and control flow. Each operator may be primitive or may be an abstract description fillable by some more detailed plan. The PA's knowledge is embodied in its *plan library*, a collection of commonly occurring, language independent plans, or as Rich calls them, programming cliches.

Shrobe [Shrobe 78] has implemented a deductive system for reasoning about PA plans. The deductive system can verify that a plan matches a portion of a program, point out bugs in a plan, and especially important, reason about the consequences of modifying an existing program.

Waters has implemented a system that given a program is able to produce a low level plan structure for it [Waters 78]. That is, Waters' system provides the analysis and rudimentary recognition task of the PA.

Chapman [Chapman 82] has implemented a testing assistant that watches over the user's shoulder during program development. The assistant helps define cogent test cases and executes them at appropriate times. In particular, the assistant can sometimes automatically update old test cases when the program is modified.

Waters [Waters 82] presents a development scenario using the PA. The development follows a knowledge-based editing approach. In it, the user constructs a Lisp program by naming the general algorithm (plan) he wishes to employ and then refining the abstract components of the plan down into Lisp code. The refinement can be done either by naming more concrete plans or by filling in literal values. There are several things worth noting about this development approach. First, there is no formal problem specification. Thus, second, arbitrary modifications, within the bounds of plan compatibility, can be made. The tradeoff here is between informality and validity. The editing approach is closer to current programming techniques. It allows a user to code up an initial solution, and then go back and modify various parts that he is unhappy with. Flexibility is gained by allowing the program to be described, both by and to the user, at various abstraction levels. Validity must be gained by the traditional testing paradigm (see [Chapman 82]). Third, the PA does not assist in collecting relevant refinements or choosing among a competing set. Most of these issues have been earmarked for further work by the PA research group (see for instance the hypothetical synthesis scenario in [Rich 81], pages 21-31).

In comparison, Glitter supports a more formal and rigid view of development, requiring the user to stick to a top-down development paradigm. Specification construction is supported through symbolic execution and paraphrasing. While the goal language can be viewed as editing commands similar to the PA editor, they expect a formal abstract specification and allow only correctness preserving modifications. Thus, if non-correctness preserving changes are to be made, they must be done at the top level. Because of this, much TI research centers on easing the burden of re-implementation. Finally, Glitter provides a knowledge-based selection engine for choosing among competing strategies and tactics.

These two approaches to development -- the PA's more informal and flexible editing paradigm and Glitter's more formal and inflexible top-down mapping paradigm -- place different requirements on their respective knowledge representations. In the PA it is important that the knowledge representation allow both synthesis and analysis, i.e., going from abstract to concrete and vice versa. Glitter does not attempt to map implementations back into specifications or allow the user to arbitrarily change a program under development. Hence, it does not require the type of analysis capability provided by the PA. In the PA, there is no formal abstract problem specification. Hence, the plan language does not need to represent the type of specification freedoms provided by Gist (demons, inferencing, global constraints). Glitter must capture the strategies and tactics necessary for mapping these freedoms. In conclusion, it appears that the two research efforts may be heading for some common ground: Waters [Waters 82] mentions plans to add formal specifications to the PA; part of TI's (and hence Glitter's) future plans include studying various alternatives to the classical top-down refinement approach to design.

8.4. The FOO system

Mostow [Mostow 81] has built a system, FOO, for *operationalizing* a problem description stated at the domain level into a procedure executable by a task agent. In the domain of the card game Hearts, for instance, the problem of deciding whether the queen of spades is out (in an opponent's hand) can be stated to FOO as ***(Evaluate (Out QS))***. Through operationalization, a procedure is developed which relies only on actions executable in the task environment by the task agent. In this case the task agent is the player requesting the information. In particular, a procedure is derived which is based on the pigeon-hole principle: if an object must be in one of a finite number N of locations, then you can show that it is in

location I by showing that it is not in any of its other possible N-1 locations. In this case, the locations are the agent's hand, cards that have been played, and cards held by other players. The procedure checks if the queen is in the agent's hand or has already been played. If not then it must be in an opponent's hand, i.e., out.

The operationalization process is carried out by the user repeatedly selecting one of FOO's 300 transformations to apply and a portion of the current problem expression to apply it to. The user may also be called on to select among competing instantiations of the same transformation. As can be seen, FOO is similar to TI except for its problem domain: AI problem solving.

Of interest here is Mostow's proposal for automating portions of the operationalization process (see [Mostow 81], page 327). He suggests using a *means-end analysis* to guide rule selection. In Mostow's hypothetical scenario[43], the user provides the left hand side pattern of some rule that he wishes to apply. The means-ends analysis module would compute the difference between this and the current expression, using the difference as an index to rules that might help reduce the difference. This approach is similar to the one implemented by the Jitterer as discussed in 6.1.4. While there are fundamental problems with mapping low level difference descriptions onto high level domain operators, in Mostow's world (and Glitter's subworld) of expression reformulation it appears to have promise. In particular, Mostow hypothesizes several differencing analysis techniques that look useful in selecting among competing methods of a Glitter *Reformulate* goal[44].

8.5. The IPMS system

des Rivieres proposes a system (unimplemented) that crosses a structured editor with a transformation catalog [des Rivieres 80]. In des Rivieres's system, IPMS (Interactive Program Manipulation System), the user gives standard structured editing commands to modify or optimize *a functionally correct* Pascal program. IPMS guarantees that the user's commands are valid by implementing them as source-to-source transformations similar to those found in

[43] Mostow has since implemented a prototype that handles several examples from his thesis, albeit with a limited set of rules.

[44] This isn't surprising since Glitter's *Reformulate* goal was influenced by Mostow's notion of reformulation in FOO.

the Irvine Transformation Catalog [Standish et al 76]. There are several interesting things to note about this approach. One, the user is moved up a level from selecting transformations by name to using an editing language presumably closer to his modification goals. At the very least, the language provides an index into the transformation catalog. Two, given that a transformation is indexed to a posted goal, the transformation may contain, instead of a replacement pattern, one or more goals to achieve. In this way, IPMS could directly address the *problem* space as well as the program space in a similar way to that of Glitter's methods.

At least one weakness of the proposal is the selection process. Competing transformations are chosen in the order they appear in the catalog. Further, recursive transformations are kept from infinite sequencing by employing an application threshold: after n transformations are applied the system returns to the user with a failure message. Both of these problems are overcome in Glitter by providing a more powerful selection engine. In general, however, we find much agreement with des Rivieres approach, although we question the need for maintenance at the code level. To our knowledge, the IPMS system was never implemented.

8.6. The DRACO system

Neighbors has constructed a system, DRACO, that takes a program written in a high level domain-specific language and refines and optimizes it into a Lisp program [Neighbors 80]. The DRACO user is expected to identify the necessary objects and operations of his problem domain, and define a domain specification language around them. This involves several related tasks. First the syntax is defined through a BNF type formalism. Second, the semantics are defined by providing mappings from statements in the newly defined language into statements in one or more previously defined domain languages. Finally, a set of optimizing transformations must be defined for the new language.

The DRACO system addresses several interesting development points. First, domain analysis is *reused*. That is, once a user has carried out the difficult task of analyzing his domain's objects and operations, defining a language, mapping it to other domains and producing optimizing transformations, the newly defined DRACO domain can be used by future domain writers as a mapping target for their domain languages. Second, the idea of optimizing a program at the right level of abstraction is an important one. Domain-specific transformations provide much more powerful optimizations than is possible if optimization is postponed until

the code level. As an example, one of DRACO's domains is augmented transition networks (ATNs). One transformation defined for this domain looks for unreachable transition states and eliminates them from the specification. This is trivial when applied to the ATN language, but intractable when applied to the final Lisp code. The actual refinement process consists of the system presenting the competing refinements and the user choosing among them.

THe Gist/Glitter paradigm addresses the issue of optimization level, but currently says nothing about reuse of domain analysis. Given the difficulty of constructing correct specifications, once a correct Gist specification is achieved the effort of building it should not be wasted. This suggests interesting future research on cataloging skeletal, domain-dependent, Gist specifications which can be filled in with the necessary details when specifying a specific problem. For example, in the package router domain discussed in this thesis, we might catalog a skeleton specification for routing problems[45] that included: 1) the constraints on the items being routed and the mechanical hardware for routing them, 2) demons to flag misrouting, and 3) an environment for adding and deleting items from the system. Each new routing system could use this as a base to start the specification.

8.7. The DEDALUS system

The DEDALUS system [Manna & Waldinger 79] takes a deductive approach to program synthesis. The problem is specified using a predicate logic-like formalism that includes primitive control structures such as conditionality. A Lisp program is produced by applying both domain-specific transformations and general programming principles. If a transformation matches its pattern in the current state, the task of proving its applicability conditions is set up as a separate sub-goal. DEDALUS is run without user intervention.

Although there are major differences between DEDALUS and Glitter, there are also some interesting similarities. For one, DEDALUS allows a limited form of subgoaling (jittering) in order to prove a transformation's conditions. Secondly, DEDALUS uses selection knowledge to compute the best choice among competing transformations. This selection knowledge comes in several forms:

[45] While one can think of more general domains to abstract, e.g., physical systems, abstracting routing can still be useful. For instance, think of routing packages across wider expanses than a simple pipe/switch network such as a city, a state, a country or the world.

- There exist explicit orderings among transformations that have much the same flavor as Glitter's ordering rules.

- There exist *strategic conditions* which keep a transformation from being applied foolishly. These act in much the same way as a method's filter in Glitter. As in Glitter, this type of filtering can prevent a desirable transformation from triggering. In Glitter, the consequences of this are that the user will have to step in and produce some portion of the development himself; in DEDALUS it appears that the transformation will fail to be applied.

- Finally, there exist procedures tied to specific transformations which provide localized knowledge. These have the same flavor as Glitter's method selection rules.

Since DEDALUS is a totally automatic system, its control mechanism is vital, i.e., the onus is on the selection process to keep the search on the right path. Because DEDALUS's selection knowledge is hard wired into the program, it is difficult to say how easily it can be examined, modified or added to, or how this will affect efforts to scale up to larger problems.

8.8. The ZAP system

Feather [Feather 79, Feather 82b] has constructed a program transformation system, ZAP, based on Burstall and Darlington's fold/unfold model [Burstall & Darlington 77]. Darlington's implementation [Darlington 81] uses the recursive equation language NPL [Burstall 77] (which has evolved into HOPE [Burstall et al. 80]) to specify a simple but not necessarily efficient applicative procedure. By applying 6 basic transformations over and over their system is able to develop the simple function into an efficient one. Because of the generally large search space, the user is required to provide detailed development guidance.

Feather notes that overloading the user with the large amount of mundane detail in the Burstall and Darlington system makes it impractical to apply to large problems. Feather's solution is to automate the detailed portions by relying on the user to supply enough guidance to allow transformations to be strung together by the system (Darlington [Darlington 81] presents another approach based on user control of folding/unfolding similar to Glitter's faith modes.). This guidance comes in several forms:

- The general development context is set. This includes choosing which functions to use in the fold/unfold operations, what simplifications may be useful and the functions that may appear in the optimized result.

- The general form of the optimized function. For instance, what terms and embedded functions are likely to be present.

- A concise specification of the instantiated cases that will be useful in development.

Feather further defines a set of tactics for choosing the functions to optimize and a strategy for ordering the tactics. Neither the tactics nor the strategy are part of the system, i.e the user is responsible for carrying them out. The ZAP system allows the transformation *process* to be viewed as a structured *object* which can be interpreted by the system, thus formalizing the development. Feather uses ZAP to develop a text formatting system taken from Kernighan and Plauger [Kernighan & Plauger 76], a specification and development clearly larger and more complex than any attempted previously by a transformation system that we are aware of.

We find many philosophical agreements with Feather's work. Like Glitter, the ZAP system builds on a formal development paradigm hampered by poor use of its human partner. Like Glitter, ZAP attempts to automate the detailed portions of a development that have little intellectual content. Like Glitter, ZAP provides the user with a higher level language to guide transformation application. In ZAP, this takes the form of a) descriptions of the goal state, b) the functions and simplifications that will likely be useful, and c) development organizational knowledge in the form of tactics and a strategy[46]. We will look at each of these in terms of Glitter.

a) In Glitter, we have chosen to provide a broader goal language than that provided by ZAP. While it is sometimes useful to state a development goal as an abstract state (see *Reformulate*), the class of transformations that can be carried out on a Gist program is much richer than the six defined for NPL and hence leads to a richer set of development concepts. Many of these concepts are not easily described by a pattern language (see for instance our method derivations in section 6.2.1).

b) The ability to provide hints to the problem solving system in the form of methods that are likely to be useful in achieving a goal is missing in Glitter. This ZAP feature would be a useful Glitter extension.

[46]Not actually a part of the ZAP system per se, but guidelines furnished in the ZAP user manual.

c) ZAP's three tactics and single strategy address the same problem as Glitter's methods -- the representation of the organizational, problem solving knowledge necessary to develop non-trivial programs. The main difference is Glitter's attention to a representation that will allow this knowledge to be examined, reasoned about, modified and added to.

8.9. The PADDLE system

The PADDLE system [Wile 81a] is a tool produced for use within the TI development model. PADDLE addresses the problem of re-implementing a specification that has changed since the initial implementation, i.e., the TI maintenance problem. A major part of the PADDLE system is a language for describing a TI development. By using the powerful editor it sits atop [Wile 81b], PADDLE allows the user to document the TI development process in much the same way as a Program Design Language (see [Caine&Gordon 75]) allows a user to document a program: by providing a skeletal structure for English description leading to primitive items. In a PDL the primitives are statements from the target language; in PADDLE they are primitive development commands such as *Match* and *Replace*.

When a specification change forces a re-implementation, PADDLE allows the development document to be *applied* to the new specification. That is, the document can be treated as a program which accepts as input a specification, and produces as output (if the program completes without error) an implementation of that specification (see ZAP,discussed previously, and a ZAP descendant defined by Darlington [Darlington 81] for similar approaches). There likely will be places where the document cannot be applied verbatim. It is these places where the user must step in and attempt to patch things up. Wile proposes several tools for helping the user in the patching process: 1) a high level language for specifying what portions of the program to focus on, 2) the attachment of templates the user expects to be matched by certain key states, and 3) an identification of milestone steps which, when reached, signal the system to print the current state of the program. The first of these seems to be tied to particular problems of the PADDLE language. However, the last two look like general techniques useful in any replay effort (we discuss maintenance issues as they relate to Glitter in section 9.1.4).

PADDLE handles jittering (what Wile calls *conditioning*) by augmenting Gist with tables of

associative and communative constructs. Thus, if the user attempts to *Match* the current expression against a pattern, the system will consult the appropriate tables and apply the associated re-combination laws to attempt to force the expression and pattern to match.

More interesting is Wile's proposal (unimplemented) for a broader jittering technique. PADDLE has two basic primitive editing commands: *Match* the current expression against a pattern; *Replace* the current expression with another expression. Wile suggests that jittering be keyed to a failure on *Match*. On failure, the system will search for a transformation that does a *Replace* and produces a new expression that can be matched against the desired pattern, i.e., the pattern we are trying to *Match*. If this process fails then the system would try a breadth-first search of all *jittering* transformations (see section 5.3.4) until either success or a depth threshold is reached (Wile suggests level 2). We see this simple search technique working well in cases where a) the set of transformations used can be kept small, b) they can also be kept cheap, and c) the search depth is low. Our experience with the Glitter development of the package router shows that many jittering tasks are lengthy (see for instance, section C.1 and for related, [Mostow 81]) and rely on selection knowledge to order potentially costly methods. In this thesis we have argued that a full blown problem solving engine is required, *in general*, to carry out jittering, with attendant vocabulary, plans and selection. However in particular cases, simple expression reformulation for example, it may be more cost effective to try the proposed brute force search first. A stronger conclusion must wait for a larger empirical base.

Chapter 9
Summary and Future Work

In this chapter, we will first summarize the major points made in this thesis, presented around the four automation issues raised in Chapter 1. Next we will discuss the usability of both the Glitter model and its implementation as viewed from a practitioners standpoint. In section 9.3 we present several usage scenarios of a hypothetical maintenance tool, future research that we view as an interesting follow-on to Glitter. Finally we compare the TI and Glitter report cards.

9.1. Automation Issues Revisited

In chapter 1 we presented four software automation issues:

1. Formalization of the development process.
2. Detail management.
3. Man/machine partnership.
4. Production of a development document usable by other tools.

In this section we will summarize the degree to which Glitter solves each.

9.1.1. Formalization of the Development Process

We view the production of software using a transformation-based model as a full blown problem solving activity. Hence, it is this problem solving activity that must be formalized. In particular, there must exist a notation or vocabulary for stating development problems, describing techniques for achieving those problems and describing how to select among competing techniques.

Problem vocabulary

To formalize the problem solving process, the first order of business is explicating the type of problems encountered. Glitter's goal descriptors provide the explication means. The current set of goal descriptors captures the freedom mapping that the developments we have studied hinge upon. With further experience, we expect that the need for new problem descriptions will become apparent. Because the goal descriptor notation allows a wide range of problems to be stated[47], we expect that new problem descriptions can be encapsulated as goal descriptors.

A drawback to our current goal representation is the atomic level at which goals are defined. Their semantics are embedded in the Lisp code that acts as the achievement checker. Hence, they cannot be reasoned about directly. Thus, any process that must analyze the goal structure must rely on some other means of finding a goal's semantics, e.g., explicitly building the information into the process.

Description of development techniques

Given a notation for describing problems, we next must find a means of describing techniques for solving them. Development techniques are represented as methods. The method template provides an index, a hook for stating application constraints and an action field. Each of these can be filled with either goal descriptors, predefined functions or user defined functions. We found that the set of method building-blocks described in chapter 6 provided the right support for defining the development methods required in the router development. Wile [Wile 81a], for one, points to the need for more sophisticated plan notation including conditionality: *if goal A cannot be achieved, try goal B* (see also [Wilkins 79]). However, this type of increase in notational power comes at a cost: the ability of the selection engine to analyze the effects of a method diminishes. We currently favor simplicity, the consequence being the need for possibly many methods to capture a piece of development knowledge which is representable as a single method in a more powerful notation. We currently value the content reference property over catalog economy.

There also is a question of development robustness or freedom. Glitter implements a basic

---

[47] i.e., any goal whose achievement can be monitored by a Lisp procedure.

top down refinement paradigm. A perusal of the general AI problem solving literature shows that many others are possible. We have discussed some of these as they relate to software development in Chapter 8. It seems certain that some will have to be considered for inclusion in Glitter. In particular, the ability to provide hints and/or constraints on the development process seems an attractive one, a capability that is supplied in one form or another by both the CHI [Green et al. 81] and ZAP [Feather 82b] systems.

Description of the Selection Process

In chapter 7 we laid out four choice points where a selection had to be made: 1) which method out of a competing set, 2) what action ordering for a selected method, 3) what state to backup to from a dead-end path, and 3) what goals implement the overall development strategy. Glitter provides notation for both 1 and 2. Section 7.5 conjectures on a notation for the last two which we won't examine further here.

Method selection is represented as 1) a set of candidate methods and 2) a selection process which weights. orders and refines the set into a final selection. Weights and ordering are supplied by selection rules. The selection rule notation is uncomplicated: an antecedent describes some selection event; the consequence either weights a set of methods or orders two methods. The things addressable by a selection rule include the competing set of methods, the current goal, the current method being applied, the reason for its selection and the planning superstructure that lies above it. In other words, the entire development history is in machine usable form.

There are several weaknesses in the selection notation:

- The system relies on the overly simple process of weighted votes to record preference and summation to represent overall worth. However, its replacement with a more sophisticated selection engine must meet certain properties necessary in a partnership model. One, the reasoning used for selection must be analyzable by both user and machine. Two, the user should be allowed to selectively request selection knowledge, e.g., "what can you tell me about method M".

- Because they are implemented as arbitrary Lisp functions, the effects of transformations are not analyzable. In section 7.2.2.2, we discuss ways of solving this problem.

- Much of the selection knowledge is based on surface features. In Kant's system,

we find these type of surface rules *in addition to* a formal analysis model. A similar model is needed in the Gist domain if we are to make more accurate estimates of cost. Indeed, some preliminary work has begun on this problem within the TI group.

- The ability to explore alternative paths to some depth would be useful not only in breaking ties, but in analyzing selections in general. Kant provides this capability in her system for choosing among alternative data structure refinements; a similar capability is needed for choosing among alternative mapping methods.

9.1.2. Detail Management

On the largest development attempted to date, the package router development of appendix C, Glitter produced 146 out of the total 159 planning steps automatically. The 13 steps provided by the user were the type of high level design goals that are the user's responsibility in the Glitter model. Out of the 146 steps produced by Glitter, 60 were actual program transformations. In a very narrow sense, we have leveraged transformation application from [60 steps/60 transformations] in the TI model to [13 steps/60 transformations] in the Glitter model. However, we argue that the total number of planning steps automated is the crucial number. The implicitness and non-automation of these steps in the TI model forestalls the organization of transformations into coherent chunks and leaves the user to reason informally about the plan space. Thus the measure of [13 steps/159 steps] is a truer indication of the automation provided by the system.

This degree of automation corresponds closely to that found in the other development produced by Glitter, the optimization of a text preprocessor (see [Balzer 76] for the TI development). However, a sample of only two is hard to extrapolate from. That is, what makes us suspect that the next development we attempt will be able to use the current system knowledge to get the same degree of automation? The answer comes in two parts, one "because of ...", and the other "even if not ...":

1. Care was taken in defining the methods and rules of the two catalogs to avoid defining development-dependent knowledge. That is, once a method was discovered for achieving a particular goal in a particular situation, an attempt was made to generalize the method to other situations. For example, most of the current methods for equivalencing two expressions started out as much more specific cases.

   Less generalizable are the selection rule weightings. The effectiveness of the selection rules can be measured by the number of times backtracking had to be

performed to undo a bad choice. In the router example this occurred only twice. However, the weights, and to a lesser extent the rules themselves, are difficult to motivate with a small sample. For example, the rule DemonsAreGood gives a method +1 if it jitters a construct into a demon during mapping. We believe the rule itself will be useful across an interesting set of problem domains, i.e. demons have some inherent development-facilitating properties. However, the weighting is close to arbitrary: we have little confidence that +1 is the right value outside the current development or even that weighting should be uniform.

2. As future developments are attempted, holes in both catalogs will be highlighted. The system facilitates the identification of missing knowledge and makes its addition to the system straightforward. We expect missing methods to manifest themselves as manual manipulation of either the problem state (user-posted goals) or the program state (manual editing of the program). The system records instances of each for post-development analysis. As new methods are identified, they can be incorporated into the existing catalog as described in section 6.2.1[48].

   We expect missing selection knowledge to manifest itself as a) excessive backtracking from dead-end or non-optimal states, or b) user override of the system's choice (when in cautious mode). Both events are recorded by the system. Adding a new selection rule is straightforward, deciding on its relative merit within the current weighting scheme less easy. We believe that a more rigorous selection process will be required as both catalogs continue to grow (see for instance [Barnett 82]).

9.1.3. Glitter As a Development Partner

In an ideal partnership, the strengths of each partner would compensate for the weaknesses of the other. This should allow the partnership as a whole to tackle much tougher problems than either of its members individually. Below we take a look at these strengths and weaknesses in the Glitter model.

First, a view of the partnership's strengths (*and corresponding weaknesses*) in regards to development methods:

- Glitter provides a repository for useful development methods.

  It is unlikely that a single user can discover or remember the collective store of development techniques.

---

[48] The user is responsible for both defining new methods and placing them in the catalog. Chiu [Chiu 81] discusses means for automating this process in a TI system; we believe that similar techniques can be used in Glitter.

- Glitter finds all methods that are applicable to a given goal.

  It is unlikely that a user can find all methods that apply to his problem. This is especially true as the catalog of methods grows.

- Glitter handles much of the mundane detail of method application, e.g. finding all places X is referenced, Y is maintained.

  The user is likely to find these details tedious to compute and easy to miss.

- The user provides overall development organization.

  Our experience base is weak in the area of high level organizational knowledge. This remains an area where !Eureka is often heard.

- The user provides insightful reasoning.

  As is evidenced by the slow progress of formal verification research, there remains much in the program-property proving business that is beyond mechanization.

Next, a view of the partnership from the perspective of selection:

- Glitter finds all selection rules that are applicable to a given selection problem and computes an ordered set of method candidates.

  It is unlikely that a user can find all selection rules that apply. This is especially true of rules that reference methods not by name but by effect or compatibility with the overall goal structure.

- Glitter handles much of the mundane detail of rule application, e.g. counting number of times X is referenced, counting number of places where Y must be unfolded.

  Again, tedious to compute and easy to miss.

- The user provides unavailable information.

  In general, this involves supplying domain-specific information, e.g. how large will some sequence grow, how often will some event occur. In some cases, the system will accept a simple estimate if exact figures are not known.

- User is responsible for exploring the development space. In particular, he is responsible for backing up from dead-end development paths.

  While we speculate on its encapsulation by current notation (see section 7.5), the knowledge necessary to control development exploration is left as future research.

9.1.4. The Development History

The output of Glitter is the full development exploration tree as pictured in figure 9-1. While at least one development path must exist from initial specification to final implementation, no restrictions are placed on the completeness of the remainder of the tree: not all paths need be explored or terminate before a final implementation is reached.

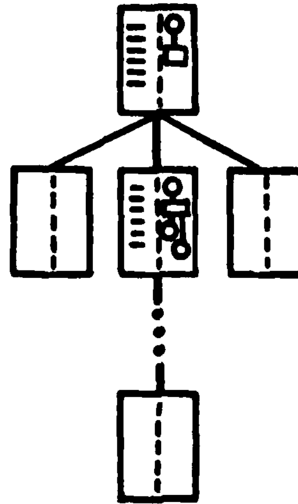


Figure 9-1: Development exploration tree

Each node in the tree represents a particular problem solving state; figure 7-1 gives a graphical representation. Included in each state are 1) the current active goal, 2) the methods competing to achieve it, 3) the selection criteria used to order them, 4) the planning superstructure that sits above all this. In the section 9.3, we show how the exploration tree can be used as a corporate body of knowledge, shared among development tools.

9.2. Usability

In this thesis we have shown the *feasibility* of automating and documenting the transformational development of software. However, we have not addressed, at least directly, the *usability* of the system. This revolves around two separate questions: how useful is the Glitter development *model*; how useful is the Glitter system as a *tool*. We will look at each in turn.

9.2.1. The model

Throughout this thesis we have stressed Glitter's partnership approach to development. There are several requirements on the human half of this partnership:

1. *Knowledge of Gist*. The user must be familiar with the syntax and semantics of the Gist specification language.

2. *Organizer*. In the ideal model, this becomes the user's primary role, the overall organization of the development.

3. *Knowledge of mapping techniques*. Interactive problem solving requires that the user follow and sometimes provide development strategies and tactics.

4. *Knowledge of selection criteria*. Choosing among competing methods often requires the user to become involved in supplying unavailable information about the program as well as arbitrating ties.

5. *Knowledge of the application domain*. The user must supply information pertaining to the particular problem domain of the program under development.

The first requirement, that of knowing Gist, currently limits the potential users of the model to less than a dozen. Assuming for the moment that the Glitter model can be used with formal specification languages other than Gist, ones that still rely on transformational development, the problem remains: the use of TI-type models has yet to gain acceptance outside of the laboratory. We have argued in this thesis that Glitter addresses part of the acceptance problem, that of automating portions of the development. We have pointed to other research which addresses another major acceptance problem, that of the construction, debugging and maintenance of formal specifications. Because a formal specification is a major component of the Glitter model, final acceptance must remain tied to the success of this work.

The second requirement, that of the user's organizer role, may slowly become less of a

burden as experience is gained. That is, as the base of example developments grows, general organizational methods should become apparent. At this point we should be able to introduce higher level goals such as *Develop* which take as arguments either large physical or conceptual chunks of the program. This addresses part of the organizational problem. The other part is the more general problem of development robustness or freedom. The Glitter model forces a basic top down design paradigm. As discussed in Chapter 8, there are others that are also attractive. General acceptance of the model will likely require a less rigid stance, i.e., the ability to provide alternative design paradigms.

The third and fourth requirements, that of mapping techniques and selection criteria, are linked to our evolutionary approach to competency. As the catalogs become filled, these should become less and less of a requirement. However, as long as a reasoning engine (e.g., theorem prover) is absent, the user must remain a part of the sometimes tedious process of proving program properties.

Finally, the fifth requirement that the user be familiar with the domain is a generally difficult one to overcome. Its solution relies on capturing in some machine usable form knowledge about the domain in general and the application in particular. See DRACO in chapter 8 for work in this area.

In summary, we might take a look at how the model has advanced the field of software development. We clearly have not added new hordes of neophytes to the ranks of Gist developers (or possibly not even performed any conversions from non-Gist sects); the use of the model requires the same development knowledge as that of Gist/TI. However, by making the Gist/TI model a more attractive one to use, we have hastened the demonstration that such a model, as it is now embodied in Glitter, can be the basis of a practical development tool. Once this has been shown, the transformational model of development will stand or fall on the attributes presented in Chapter 1.

A separate point needs to be made here. While Glitter addresses the implementation as opposed to the maintenance process, in the TI model they become almost one in the same. Maintenance is a process of changing not the final product but instead the formal specification. This in turn requires a re-implementation. As we argued in the introductory chapter, and demonstrate later in this chapter, a rationalized devalopment history becomes an input to a mechanized TI maintenance tool. Because of the large impact maintenance has

shown to have on the overall software lifecycle, one could argue that the production of a development history useful to a maintenance tool is useful in and of itself. That is, the documentation of the development planning structure can be viewed as a necessary product, regardless of the user effort needed to produce it, i.e., the automation issue can be separated from the documentation issue. However, the knowledge necessary to produce the history is the same as that used in automation. This symbiotic relationship can be used to achieve both.

9.2.2. Tool Usability

Separate from the usability of the model is the usability of the system. There are several issues to consider, the first of which is the user/machine interface. While minor enhancements can be made to Glitter's menu and command structure, the major payoff will come with better presentation of the development, problem solving and program state. Currently this information is presented in textual form. The abundance of information and its general structural nature make this often a poor medium. We view the ability to produce the various forms of development information in a graphical form and simultaneously as an important future area of research.

Secondly, there is the responsiveness of the system. The Hearsay-III implementation of Glitter, running on either a normally loaded DEC 2060 or VAX 780, is too slow to be practical as an interactive partner. This is in spite of attention paid to selection using estimated problem solving costs (see chapter 7).

There are several potential solutions to the responsiveness problem. First, Hearsay-III was designed to be a *general* system, not tied to any particular application domain. A significant number of Hearsay features are unused by Glitter, making them excess baggage unnecessarily taking up resources. Trimming off these unneeded features will buy some increase in responsiveness (for instance, replacing the general relational representation of the parse tree with a more economical Lisp structure). The extreme would be to re-build Glitter from the ground up, borrowing only those features that were found useful in the Hearsay implementation; we have attempted to avoid this alternative up to now.

However efficiently we construct our systems, ones which are knowledge-rich and interactive such as Glitter will remain in need of powerful processors. The most leverage can be gained

by moving off a time sharing system with its load average vagaries and onto powerful single user machines, i.e., the cheap[49] hardware mentioned in the introduction. While Hearsay does not currently run on a single user machine, efforts are under way to rectify this; we look forward to their successful completion.

9.3. Some Maintenance Examples

We have argued that the document produced by the development process should be a formal, machine usable product. In this section, we will look in a little more detail at how such a product might be used in software maintenance. We will present two examples, the first an interrogation of the development history by the user, the second a modification of the original development to accommodate a specification change, i.e., a *re-implementation*. While both examples are based on the planning structure produced by Glitter, we stress that no such tool currently exists.

9.3.1. A browsing example

Suppose we are given the following fragment from the initial specification of some program (an abstraction of the conditional wait in demon **RELEASE_PACKAGE_INTO_NETWORK**):

```
...
if ∃x || P(x,y) then A;
...
```

Suppose further that one step in a development was to replace y with $Q(x)$ by applying some transformation T (an abstraction of the first program transformation of the router development):

```
...
if ∃x || P(x, Q(x)) then A;
...
```

We assume that a maintainer is browsing through the development looking for places where performance improvements could be made, i.e., design decisions could be improved upon. We see the ultimate goal of such a browser as providing a "you-are-there" capability, allowing

[49] Currently, not so cheap, but rapidly falling.

the maintainer to place himself at any decision point and see exactly what the the developer saw in the original development.

The above step in isolation is quite unmotivated: take some variable and replace it with a relation reference. The rationale behind this step in the full development is fairly deep (steps 1.1 - 1.7): 1) we want to get rid of the relation P, 2) to get rid of P we have to remove all references to it, 3) to remove all references we have to remove this reference, 4) to remove this reference we can try folding the existentially quantified expression involving P into a new relation and then worry about getting rid of P from there, 5) to fold the expression we would like to get rid of any references to non-quantified variables, e.g., *y*, 6) one way of getting rid of a reference to a non-quantified variable V is to replace the reference with an equivalent expression that doesn't reference V, 7) replace *y* with the relation reference Q(*x*). We can view this as a (small) portion of a plan for getting rid of a particular relation within the specification[50]. In the process of forming this plan, points were encountered where a choice had to be made among competing methods. Selection knowledge was applied to choose the best method among the candidates.

Suppose now that the user (i.e., maintainer) was interested in the reference to Q(x). He or she might ask the following:

User: *Where did Q(x) come from?*

MaintenanceTool: the application of transformation T replaced *y* with Q(*x*).

Note that this question can be answered by recording nothing more than transformation applications. However, suppose that we asked the following:

User: *Why was y replaced by Q(x) (or why was T applied)?*

MaintenanceTool: because we want to replace *y* with an expression using non-quantified variables.

User: *Why?*

MaintenanceTool: because we want to fold an expression that contains *y*.

---

[50]The motivation for getting rid of P in the first place must be supplied by the user.

User: *Why?*

MaintenanceTool: ...

User: *Can I replace the transformation $y \Rightarrow Q(x)$ with $y \Rightarrow S(z)$?*

MaintenanceTool: yes. However that choice was rejected because it introduces a new free variable *z*.

User: *Under what circumstances can I get rid of this step altogether?*

MaintenanceTool: in the most general case, when you are not trying to get rid of P.

Each of these questions relies on some portion of the development history produced by Glitter, e.g., the goal structure, the selection process. Each could be answered by the current Glitter system if stated in very restricted terms. Our goal is to generalize the access to the information and allow a much richer retrieval language. Swartout [Swartout 81] discusses the use of a similar question answering capability in the domain of expert medical programs.

9.3.2. A re-implementation example

Here we will look at the manipulation of the package router development given a specification change[51]. Suppose we notice the following:

> After running the package router for sometime, it is discovered that consecutive packages rarely have the same destination. Hence, most packages entering the router are delayed. A decision is made to do away with the conditional check in **RELEASE_PACKAGE_INTO_NETWORK** and simply delay each package, i.e., unconditionally wait. Note that this is a specification modification as opposed to a development step.

To achieve this modification, assume that the following *specification transformation*[52] has been made, replacing $\blacktriangleright_1$ with $\blacktriangleright_2$:

---

[51]We add as anecdotal information that the spec change of this example was motivated by browsing (manually) through the design decisions made in the original development history.

[52]At least part of the effort of building a maintenance assistant will be to classify the various types of transformations that are made to a specification, e.g., enhancement, disambiguation, constraint.

Old Spec:

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
▶1    if (the package.previous ||
              package.previous immediately before package.new
                    wrt PACKAGES_EVER_AT_SOURCE(*)
          ):DESTINATION ≠ package.new:DESTINATION
        then invoke WAIT[];

      update :LOCATED_AT of package.new
           to (the source):SOURCE_OUTLET
    end;
```

New Spec:

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
▶2    invoke WAIT[];

      update :LOCATED_AT of package.new
           to (the source):SOURCE_OUTLET
    end;
```

We would like to reuse as much of the development in appendix C as possible in reimplementing the new spec. We begin most naturally at the highest portions of the goal tree (appendix B may be useful in following this discussion). The first goal posted by the user (step 1.1) was the removal of the relation PACKAGES_EVER_AT_SOURCE. This goal is still valid, as is the method chosen to achieve it, RemoveRelation. That is, both goal and method use as context the definition of the relation PACKAGES_EVER_AT_SOURCE which is still around. The RemoveRelation method attempts to remove a relation by first removing all reference to it. The re-application of RemoveRelation to the new spec produces an interesting result: because the single reference to the relation has been eliminated, the definition can be removed without further ado. In other words, the entire goal structure below step 1.1 (steps 1.2 - 1.22) is eliminated. Imagine the effort involved given only the transformation sequence: each transformation application would have to be examined individually to determine its use in the old development and its potential need in the new. As an illustration, one of the transformations we would need to study is the one presented in the previous browsing example, given there in its abstract form. We would be required to answer similar questions to those given, but now inferring the corresponding goal structure.

The next user goal (step 2.1) is the removal of the relation PREVIOUS_PACKAGE. However, this relation in the original development was a part of the residue of removing PACKAGES_EVER_AT_SOURCE; it doesn't exist in the new development and hence the step (and those subordinate to it, 2.2 - 2.14) can be eliminated. Whether *in general* a goal that loses its context can be eliminated entirely is open to question. We plan to explore these type of questions in our future work.

The next user goal (step 3.1) is the removal of the relation LAST_PACKAGE. Again, this relation in the original development was residue from the removal of PACKAGES_EVER_AT_SOURCE; it doesn't exist in the new development and can be eliminated (along with 3.2 - 3.5).

The last three user goals -- mapping the constraint DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE, mapping the relation PACKAGES_DUE_AT_SWITCH and mapping the demonic structure -- can be run verbatim as in the old development.

Let's examine what has happened here. Close to half of the original development has been eliminated and the remainder run verbatim. In eliminating the portions of the old development, we were required to examine only three high level user goals. We can see in retrospect that these three goals should be subgoals of some still higher level goal such as "optimize use of package history". Because of its domain dependence, this goal would likely be defined by the user (see section 5.5.1). With this new structure, even less of the development would need to be examined, i.e., optimizing something that is no longer needed is wasteful and should be eliminated from the development.

In summary, we have illustrated through hypothetical scenarios two important uses of the full problem solving history produced by Glitter. One, it might be used in a browsing system that allows the user to inspect and attempt to improve on the design decisions made in a development. Two, it might be used by a maintainer to index chunks of the development to the goal they achieve. There are clearly further things to consider, such as a broader model of both spec and development dependencies. Also, previous research into plan reuse in other problem solving areas looks worth investigating here (see for instance [Hayes-Roth et al. 81]). Our future plans include defining a maintenance tool that integrates spec changes, goal structure, development modification, and replay. As in Glitter, an important goal of this work will be to identify the role both user and machine will play.

9.4. Final Grades

In figure 1-1 we graded the TI model of software development according to Balzer's six criteria. It is only fair that we now provide a report card on the Glitter system using the same criteria.

Glitter Report Card	
1. Ease of Specification	B
2. Efficiency of the Implementation	B
3. Ease of Maintenance	B/B-
4. Correctness of the Implementation	A
5. Resources Required	C
6. Type of Problems Handled	B+
Comments: *Gets along well with others.*	

Figure 9-2: The Glitter Report Card

The grading rationale for figure 9-2 follows.

1. *Ease of Specification* (B). Unchanged from TI model.

2. *Efficiency of the Implementation* (B). The efficiency of the final implementation rests on the combined strengths of user and machine (see 9.1.3).

3. *Ease of Maintenance* (B/B-). The first grade, the ease of modifying the specification, is unchanged from the TI model. The second grade reflects the ease with which a new implementation can be obtained incorporating the modification. The development history

produced by Glitter provides the rationale behind each step in the original development. The drawback is that the no tool currently makes use of this during re-implementation. That is, the user must use the history as documentation in constructing the new development. In section 9.3, we speculate on how the development history can be used as a machine usable product during re-implementation.

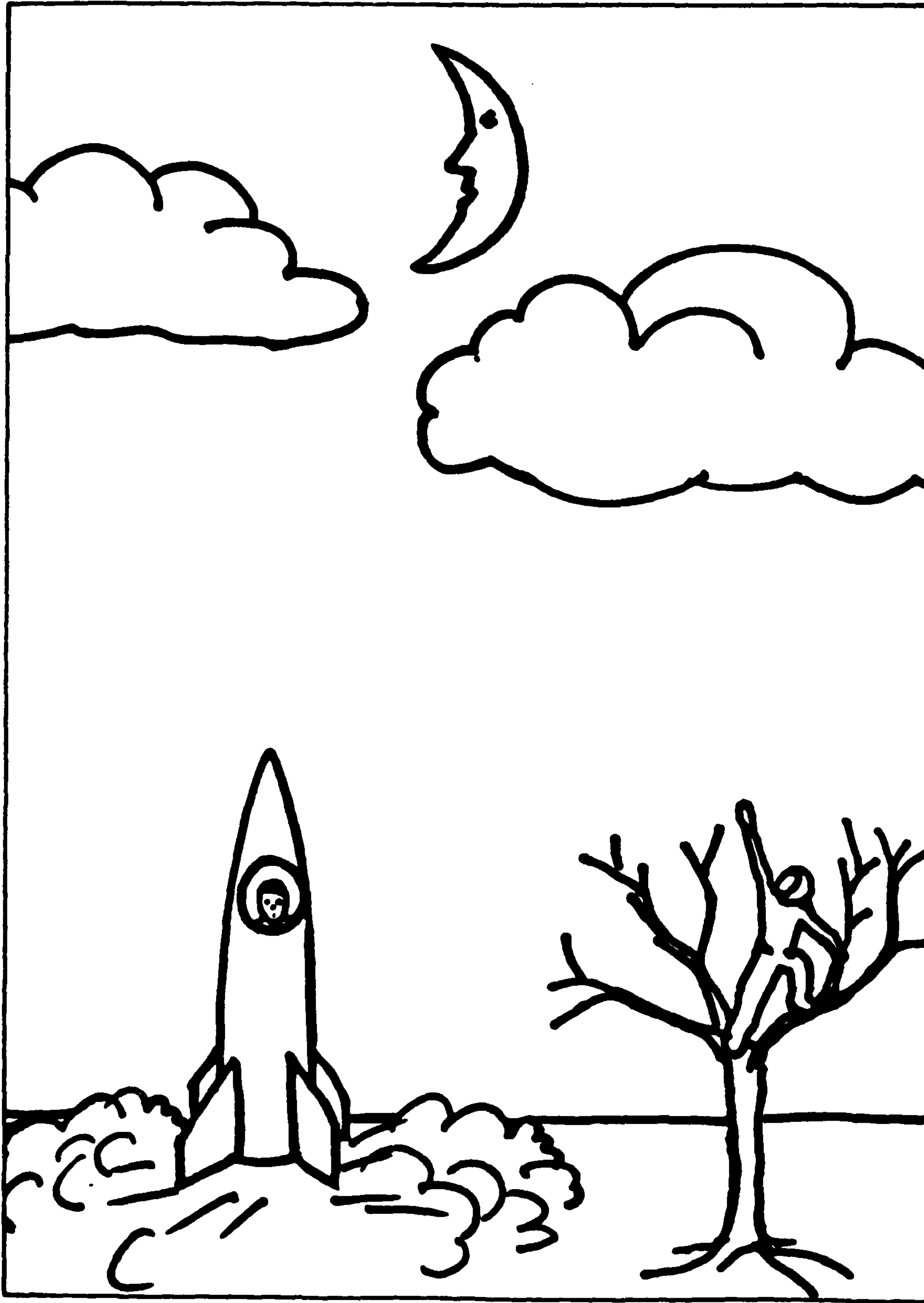
4. *Correctness of the Implementation* (A). Unchanged from the TI model.

5. *Resources Required* (C). This is the same TI grade, but now for different reasons. We gave TI a C because of its overburdening of the user. We give Glitter a C because of its overburdening of the underlying hardware.

6. *Type of Problems Handled* (B+). Our addition of one more development example is not enough to significantly raise the TI grade.

In summary, we have improved the grades in several categories. Just as importantly we have improved the ease with which grades can be further improved. That is, our knowledge-based approach and reliance on the machine to carry out details should pay big dividends as experience is gained and more powerful hardware built. As with our two astronauts in figure 9-3, both mechanized and non-mechanized approaches are far from solving the software problem. However we argue that the formalization and automation provided by Glitter is a step in the right direction.

Figure 9-3: The Space Race



References

[Balzer 72] Balzer, R.
Automatic Programming.
Technical Report 1, Information Sciences Institute, September, 1972.

[Balzer 73] Balzer, R.
A global view of automatic programming.
In *ijcai3*, pages 494-499. 1973.

[Balzer 76] Balzer, R., Goldman, N., and Wile, D.
On the Transformational Implementation approach to programming.
In *Second International Conference on Software Engineering.* , October, 1976.

[Balzer 80] Balzer, R., Erman, L., London, P., and Williams, C.
Hearsay-III: A domain-independent framework for expert systems.
In *First National Conference on Artificial Intelligence.* AAAI, 1980.

[Balzer 81] Balzer, R.
Transformational Implementation: An Example.
IEEE Transactions on Software Engineering SE-7(1):3-14, 1981.

[Balzer & Goldman 79]
Balzer, R. and Goldman, N.
Principles of good software specification and their implications for specification languages.
In *Specification of Reliable Software*, pages 58-67. IEEE Computer Society, 1979.

[Balzer et al. 78] Balzer, R., Goldman, N. and Wile, D.
Informality in program specifications.
IEEE Transactions on Software Engineering SE-4(2):94-103, 1978.

[Balzer et al. 82] Balzer, R., Goldman, N. and Wile, D.
Operational specification as the basis for rapid prototyping.
In *Proceedings, ACM SIGSOFT Software Engineering Symposium on Rapid Prototyping.* 1982.

[Barnett 82] Barnett, J.
How Much is Control Knowledge Worth?
June, 1982.
Working paper available from author, USC/Information Sciences Institute.

[Barstow 79a] Barstow, D.R.
"Knowledge-Based Program Construction".
Elsevier North-Holland, 1979.

[Barstow 79b] Barstow, D.
The roles of knowledge and deduction in program synthesis.
In *IJCAI 6*, pages 37-43. 1979.

[Barstow 79c] Barstow, D.
Automatic Construction of Algorithms and Data Structures Using a Knowledge Base of Programming Rules.
PhD thesis, Computer Science Dept., Stanford University, 1979.

[Barstow & Kant 77]
Barstow, D., and Kant, E.
Observations on the interactions between coding and efficiency knowledge in the PSI system.
In *Second International Conference on Software Engineering*, pages 19-31. 1977.

[Bauer et al 77] Bauer, F.L., Partsch, H., Pepper, P. and Wossner, H.
Techniques for program development.
In *Infotech State of the Art Report, Software Engineering Techniques*, pages 25-50. Infotech Information Ltd, Maidenhead, Berkshire, England, 1977.

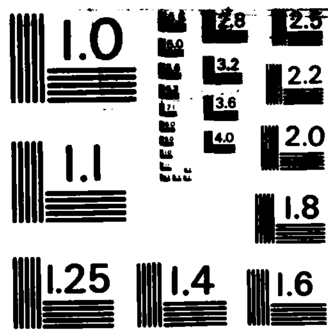
[Bauer et al 78] Bauer, F.L., Broy, M., Gnatz, R., Partsch, H., Pepper, P. and Wossner, H.
Towards a wide spectrum language to support program specification and program development.
SIGPLAN Notices 13(12):15-23, December, 1978.

[Bentley 81] Bentley, J.
Writing Efficient Code.
Technical Report CMU-CS-81-116, Carnegie-Mellon University, April, 1981.

[Broy & Pepper 80]
Broy, M. and Pepper, P.
Program development as a formal activity.
IEEE Transactions on Software Engineering SE-7(1):14-22, 1980.

[Bulnes-Rozas 79]
Bulnes-Rozas, J.
GOAL: A Goal Oriented Command Language for Interactive Proof Construction.
PhD thesis, Computer Science Dept., Stanford University, 1979.

AD-A139 860 AUTOMATING THE TRANSFORMATIONAL DEVELOPMENT OF SOFTWARE 3/3
VOLUME 1(U) UNIVERSITY OF SOUTHERN CALIFORNIA MARINA
DEL REY INFORMATION S.. S F FICKAS MAR 83
UNCLASSIFIED ISI/RR-83-108 NSF-MCS79-18792 F/G 9/2 NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

[Burstall 77] Burstall, R.M.
Design considerations for a functional programming language.
In *The Software Revolution: Proceedings, Infotech State of the Art Conference, Copenhagen*, pages 45-57. Pergamon, 1977.

[Burstall & Darlington 77]
Burstall, R.M. and Darlington, J.
A transformation system for developing recursive programs.
JACM 24(1):44-67, January, 1977.

[Burstall et al. 80] Burstall, R.M., MacQueen, D.B. and Sannella, D.T.
HOPE: an experimental applicative language.
In *Proceedings, 1980 LISP Conference, Stanford*, pages 136-143. 1980.

[Caine&Gordon 75]
Caine, S., and Gordon, E.
PDL--a tool for software development.
In *National Computer Conference Proceedings, 1975*. AFIPS, 1975.

[Chapman 82] Chapman, D.
A program testing assistant.
CACM 25(9):625-634, September, 1982.

[Cheatham 81] Cheatham, T.
Program refinement by transformation.
In *5th International Conference on Software Engineering*. San Diego, Ca., March, 1981.

[Chiu 80] Chiu, W.
Structure comparison and semantic interpretation of differences.
In *First National Conference on Artificial Intelligence*. AAAI, 1980.

[Chiu 81] Chiu, W.
Structure Comparison and Semantic Interpretation of Differences.
PhD thesis, USC, 1981.

[Cohen et al 82] Cohen, D., Swartout, W. and Balzer, R.
Using symbolic execution to characterize behavior.
In *Proceedings, ACM SIGSOFT Software Engineering Symposium on Rapid Prototyping*. 1982.

[Darlington 81] Darlington, J.
An experimental program transformation and synthesis system.
Artificial Intelligence 16(1):1-46, March, 1981.

[Davis 77] Davis, R., Buchanan, B., and Shortliffe, E. H.
Production rules as a representation for a knowledge based consultation system.
Artificial Intelligence 8:15-45, Spring, 1977.

[Davis 80] Davis, R.
Meta-Rules: Reasoning about control.
Artificial Intelligence (15):179-222, 1980.

[des Rivieres 80] des Rivieres, J.
The Design of an Interactive Program Manipulation System.
Master's thesis, University of Toronto, May, 1980.

[Erman et al. 81] Erman, L., London, P., and Fickas, S.
The design and an example use of Hearsay-III.
In *IJCAI-7*, pages 409-415. Vancouver, BC, 1981.

[Feather 79] Feather, M.S.
A System for Developing Programs by Transformation.
PhD thesis, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, 1979.

[Feather 82a] Feather, M.S.
Mappings for rapid prototyping.
In *Pre-Proceedings, ACM SIGSOFT Software Engineering Symposium on Rapid Prototyping*, pages Paper 47. April, 1982.

[Feather 82b] Feather, M.S.
A system for assisting program transformation.
ACM Transactions on Programming Languages and Systems 4(1):1-20, January, 1982.

[Fickas 80] Fickas, S.
Automatic goal-directed program transformation.
In *AAAI80*, pages 68-70. Palo Alto, CA, Aug, 1980.

[Gerhart 75] Gerhart, S.
Knowledge about programs: A model and case study.
In *Proceedings of the International Conference on Reliable Software*, pages 88-95. June, 1975.

[Goldman & Wile 79]
Goldman, N., and Wile, D.
A data base specification.
In *International Conference on the Entity-Relational Approach to Systems Analysis and Design.* , UCLA, 1979.

[Green et al. 79] Green, C.
Results in knowledge based program synthesis.
In *IJCAI 6*, pages 342-344. 1979.

[Green et al. 81] Green, C., et al.
Research on Knowledge-Based Programming and Algorithm Design - 1981.
Technical Report KES.U.81.2, Kestrel Institute, Kestrel Institute, 1801 Page Mill Road, Palo Alto, Ca. 94304, August, 1981.
Revised 9/82.

[Hayes-Roth et al. 81]
Hayes-Roth, B., Hayes-Roth, F., Shapiro, N., Westcourt, K.
Planner's Workbench: A Computer Aid to Re-planning.
Technical Report Rand Paper P-6688, RAND, October, 1981.

[Hommel 80] Hommel, G.
Vergleich verschiedener Spezifikationsverfahren am Beispiel einer Paketverteilanlage.
Technical Report, Kernforschungszentrum Karlsruhe, August, 1980.

[Kant 79] Kant, E.
Efficiency Considerations in Program Synthesis: A Knowledge-Based Approach.
PhD thesis, Stanford, 1979.

[Kernighan & Plauger 76]
Kernighan, B.W. and Plauger, P.J.
Software Tools.
Addison-Wesley, 1976.

[Kibler 78] Kibler, D.F.
"Power, Efficiency, and Correctness of Tranformation Systems".
PhD thesis, University of California, Irvine, 1978.

[Kibler et al. 77] Kibler, D., Neighbors, J., Standish, T.
Program manipulation via an efficient production system.
In *Proceedings of the Symposium on Artificial Intelligence and Programming Languages.* SIGART/SIGPLAN, 1977.

[London & Feather 82]
London, P. and Feather, M.S.
Implementing Specification Freedoms.
Technical Report RR-81-100, ISI, 4676 Admiralty Way, Marina del Rey, CA 90291, 1982.
Submitted to Science of Computer Programming.

[Manna & Waldinger 79]
Manna, Z. and Waldinger, R.
Synthesis: dreams => programs.
IEEE Transactions on Software Engineering SE-5(4):294-328, 1979.

[Mark 80] Mark, William.
Rule-based inference in large knowledge bases.
In *Proceedings of the National Conference on Artificial Intelligence*. American Association for Artificial Intelligence, August, 1980.

[McCune 79] McCune, B.
Building Program Models Incrementally from Informal Descriptions.
PhD thesis, Stanford, October, 1979.
available as AIM-333, STAN 79-772.

[McDermott 77] McDermott, D.
Vocabularies for problem solver state descriptions.
In *Fifth International Joint Conference on Artificial Intelligence*. , 1977.

[McDermott 78] McDermott, D.
Planning and Acting.
Cognitive Science 2(2), April-June, 1978.

[Mostow 81] Mostow, D. J.
Mechanical Transformation of Task Heuristics into Operational Procedures.
PhD thesis, Carnegie-Mellon University, 1981.
Available as CMU-CS-81-113.

[Neighbors 80] Neighbors, J.M.
"Software construction using components".
PhD thesis, University of California, Irvine, 1980.

[Newell 72] Newell, A., and Simon, H. A.
Human Problem Solving.
Prentice-Hall, Englewood Cliffs, N. J., 1972.

[Paige & Koenig 82]
Paige, R. and Koenig, S.
Finite differencing of computable expressions.
ACM Transactions on Programming Languages and Systems 4(3):402-454, July, 1982.

[Phillips 77] Phillips, J.
Program inference from traces using multiple knowledge sources.
In *IJCAI 5*, pages 812. 1977.

[Rich 81] Rich, C.
Inspection Methods in Programming.
PhD thesis, MIT, 1981.
available as AI-TR-604.

[Rich & Shrobe 78]
Rich, C., and Shrobe, H.
Initial report on a LISP programmer's apprentice.
IEEE Transactions on Software Engineering SE-4(6):456-467, 1978.

[Rich et al. 79] Rich, C., Shrobe, H., Waters, R.
An overview of the Programmer's Apprentice.
In *6th International Joint Conference on Artificial Intelligence*. Tokyo, Japan, August, 1979.

[Rutter 77] Rutter, P.
Improving Programs by Source-to-source Transformations.
PhD thesis, University of Illinois, 1977.

[Shrobe 78] Shrobe, H.
Reasoning and Logic for Complex Program Understanding.
PhD thesis, MIT, 1978.

[Sproull 81] Sproull, R.
Using Program Transformations to Derive Line-Drawing Algorithms.
Technical Report CMU-CS-81-117, Carnegie-Mellon University, April, 1981.

[Standish et al 76] Standish, T.A., Harriman, D.C., Kibler, D.F. and Neighbors, J.M.
The Irvine Program Transformation Catalogue.
Technical Report, University of California, Irvine, 1976.

[Steinberg 80] Steinberg, L.
A Dialog Moderator for Program Specification Dialogues in the PSI System.
PhD thesis, Stanford, 1980.

[Swartout 81] Swartout, W.R.
Explaining and justifying expert consulting programs.
In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*. University of British Columbia, August , 1981.

[Swartout 82] Swartout, W.
GIST English generator.
In *AAAI82*. Aug, 1982.

[Swartout & Balzer 82]
Swartout, W. and Balzer, R.
On the inevitable intertwining of specification and implementation.
CACM 25(7):438-440, 1982.

[Tappel 80] Tappel, S.
Some algorithm design methods.
In *AAAI80*, pages 64-67. Stanford University, 1980.

[Terry 82] Terry, A.
Hierarchical Control of Production Systems.
PhD thesis, UC Irvine, 1982.

[Waters 78] Waters, R.C.
Automatic Analysis of the Logical Structure of Programs.
PhD thesis, MIT, December, 1978.
available as AI-TR-492.

[Waters 82] Waters, R.
The programmer's apprentice: Knowledge based program editing.
IEEE Transactions on Software Engineering 8(1):1-12, January, 1982.

[Wegner 79] Wegner, P. (editor).
Research Directions in Software Technology.
MIT Press, 1979.

[Wile 81a] Wile, D. S.
Program Developments as Formal Objects.
Technical Report RR-81-99, ISI, 4676 Admiralty Way, Marina del Rey, CA 90291, 1981.

[Wile 81b] Wile, D. S.
POPART: Producer of Parsers and Related Tools, System Builders' Manual
ISI, 4676 Admiralty Way, Marina del Rey, CA, 90291, 1981.

[Wilensky 80] Wilensky, R.
Meta-planning.
In *1st National Conference on AI*, pages 334-336. American Association for Artificial Intelligence, Stanford, August, 1980.

[Wilkins 79] Wilkins, D.
Using plans in chess.
In *IJCAI6*, pages 960-967. Tokyo, Japan, August, 1979.

END

FILMED

5-84

DTIC