CIS-TR-84-04

A Problem Solving Approach to Software Development

by

Stephen Fickas
Computer Science Department
University of Oregon
Eugene, Oregon 97403

July 1984

ABSTRACT

We present a new model of software development by transformation. The model uses a problem solving approach to automate and document the development process. The model is incorporated in a system called Glitter, which explicitly represents the goals and methods that lead to transformation applications, and the selection criteria used to select one transformation over another. Glitter, using a partnership approach with the user, has been able to automatically generate 90 percent of the planning and transformation steps in the examples studied. Further, by using a by-product of the Glitter system – a record of the planning that leads to a final implementation – we are able to begin to look at automating other software lifecycle processes.

1. Introduction

This paper presents a problem solving model of software development. The model starts with an abstract, formal specification and *transforms* the specification into a compilable implementation. Transformation relies on interactive problem solving during which the user specifies development goals, and the machine responds by finding and applying methods to achieve those goals. The model incorporates each of the following items:

- A *language* for stating software design goals. This language is the communication medium between user and machine.
- A *catalog of methods* for achieving those goals.
- A *catalog of selection rules* for choosing among competing methods.
- A *partnership*, which plays off the strengths of both user and machine.
- Automatically generated *documentation* of the software design process that includes the goal structure, competing methods, and selection rationale used to reach an implementation.

Based on the model, a system that partially automates the software design[1] process has been built [1]. The system, called Glitter,[2] has been used to produce implementations of a text editing program and the controller for a postal package router.

---

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

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

1.1. A Problem Solving Approach

The Glitter system uses a problem solving model to provide mechanized support for software development. The user provides the development goals that he wants achieved, and the system finds means of solving them. The aim is to have the system achieve the user's goals automatically, bothering the user only when missing (e.g., insightful, 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. Further, the system should document its problem solving process in a way that can be used by other development processes, e.g.; maintenance, replay, explanation.

The use of an interactive problem solving model for transformational software development has several benefits:

- *What vs. How.* The model advocates a shift in the way transformational developments 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, and uses the goal language to state it. The system then makes available all methods which might achieve the goal and all knowledge that might help choose among them.
- *Automation.* Many development steps involve low level transformations. These steps can be automatically produced by the system, i.e., one user goal may lead to many transformation applications. The user is left free to attend to the more important strategic issues involved in organizing a development.
- *Partnership emphasized.* The design process is viewed as a joint activity with the strengths of each partner emphasized: the user supplies strategic and insightful knowledge; the machine supplies precision, analysis, recording, and catalogs of knowledge found useful in past development efforts. 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. Using a partnership model, tougher problems can be handled by including the user; research progresses by gradually removing the user from the process.
- *Development history recorded.* The by-product of a Glitter development is the planning structure which produced the actual transformation steps. The planning structure provides the teleology of statements in the implementation; it shows how an implementation was derived, not simply what the implementation does. Other tools can use this structure to analyze and change a development, i.e., do maintenance.

The Glitter problem solving model can trace its roots to 3 key ideas developed by past research efforts in the area of transformational development of software (a good review of work in the transformational area can be found in [2]).

Key Idea 1: a partnership model that uses the human for guidance, and the machine for application

allows both a greater variety and complexity of problems to be tackled than a strictly automatic approach. See [3,4,5] for some representative systems that use a form of man/machine partnership to build non-toy programs.

Key idea 2: a transformational development can be viewed as just another product to be created, analyzed, edited. Systems that allow a development to be treated as an object of study include [6,7].

Key idea 3: a development is structured. A transformational development should record more than just the program transformations applied. In particular, a user will employ various strategies that help organize and decompose a development into manageable pieces. A development should capture this planning structure. The PADDLE system [6] in particular allows the user to organize a development around subtasks; the subtasks are recorded as part of the development.

In this paper we present the **fourth key idea**: producing a transformational development is a problem suited to an automated, problem solving approach. Glitter uses all four of these ideas to provide an interactive, problem solving system to automate the production of transformational developments.

1.2. The Foundations

Glitter is based on the transformational model of software development. In particular, it was developed as one part of the Transformational Implementation (TI) project at USC/Information Sciences Institute [4,8,9]. The three major components of the TI project are

(1) The formal specification language called Gist [9,10].

(2) The interactive transformation engine, which incrementally maps specifications into implementations [4,8].

(3) The development-structuring language PADDLE [6]. PADDLE allows a developer to record his development as an executable program. This program can be run during maintenance to automatically produce portions of the original development. We will discuss PADDLE in more detail in later sections.

1.2.1. Specification in TI

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.

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 [11] 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.

1.2.2. Implementation in TI

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.[3] The mapping process may involve mapping operational freedoms, informational freedoms or efficiency freedoms.

Using PADDLE, the specification writer defines the tasks or goals that he wishes to pursue. He then specifies what subtasks must be accomplished to complete the tasks. This process continues until he defines the primitive transformations to be applied to the program. The application of a transformation produces a new program state. The final development is a series of transformation applications leading from the initial specification to the desired implementation. The development is structured by the tasks the user defined along the way.

1.2.3. Maintenance in TI

The development process, whether 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 frequently 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. That is, a change in the problem is made as a change to the specification. Each time such a specification change is made, a new development is produced by repeating the mapping process. Because PADDLE allows the user to document his development, the potential for *reusing* a past development in a new situation is much enhanced. In particular, PADDLE allows the user to transform a development[4] to add, delete or modify sections to correlate with the new specification. Note that the treatment of the development as just another program allows the application of the same machinery defined to handle a specification program, a powerful generalization.

1.3. Adding automation to the TI Model

In the TI model as implemented by PADDLE, the user is responsible for defining the task and sub-task structure, as well as deciding what transformations to apply and where to apply them. The machine is responsible for record keeping and faithful application of transformations. While this model forms the

---

[3] We will discuss Gist freedoms in more detail in later chapters. For now, a freedom can be viewed as a specification construct that avoids implementation or efficiency concerns.

[4] Primitive transformations in PADDLE are akin to editing steps. Hence, it is up to the user to define "higher level" transformations that preserve some property (correctness, consistency). In this case, it is up to the user to insure that a change to one part of a specification is consistent with the remaining portions of the development. In section 6, we will discuss this issue further.

right formal foundation for an automation effort, it does not address some sticky issues. We will discuss these issues in this section.

1.3.1. Process Formalization

Much of of the work in complex problem solving domains such as software development involves 1) formulating the right goals or tasks to pursue, 2) finding the right strategies or plans for refining them into more manageable subgoals, and 3) finally manipulating the domain objects (e.g., programs) through primitive operators (e.g., program transformations). For example, one high level plan for achieving the goal "implement this Gist specification" is "first find implementations for all data structures, and then work on implementing actions". Plans of this type are far removed from actual program transformations; many intermediate problem solving steps must be generated before even the first change is made to the program.

If we wish to formalize the problem solving process, we must define a language for 1) stating particular problems (goals), and 2) building plans for solving those problems (achieving those goals). PADDLE provides the framework for stating goals and plans, but not the goal language and plans themselves.

1.3.2. Detail Management

Our experience with transformational developments [4,8,9] has produced an important result: many of the transformation steps are not the interesting and clever optimizations we rely on from expert designers, but the mundane preparatory and clean-up steps which are the filler between. Often, the attention which must be paid to these burdensome steps distracts a designer from the more important optimizations that lead to real performance gains. We would like the the machine to automatically select and apply entire sequences of low level transformations to meet some higher level development goal.

1.3.3. User's Role

Our experience is that the complexity of a transformational development requires a high degree of automation. The machine should help organize the development and automate as much of it 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. The catalog of development methods can be expected to both grow large and be modified often as new methods are added, old ones deleted and others updated. 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

machine becomes more knowledge rich through the addition of more transformations, the partnership as a whole becomes weaker because of the the user's increasing difficulty in 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. Selecting the best one to apply is generally a non-trivial task, and one that the machine should participate in.

1.3.4. 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? Answers to these questions will allow the tool to decide whether parts of the original development can be reused given changes to the original specification. Answers will come from the planning structure that overlays and rationalizes 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. PADDLE captures a portion of the planning space, the tasks explicitly defined by the user. Our goal can be more ambitious: by automating the development process, we capture all system problem solving.

In summary, a transformational development is lengthy, and contains many low level steps. In general, the transformational development model is only partially formalized and documented, and practically unautomated. The model presented in this paper addresses these problems by extending the TI model to include an automated partner.

2. A small example

In this section we will look at the development of a problem specified in Gist.[5] The English description of the problem, the control of a postal package router, is as follows:

> Consider a routing system for distributing packages into destination bins. The topology of the system is a network consisting of a *source*, plus a set of binary *switches* connected by *chutes*, terminating in *bins*. Figure 2.0 depicts this graphically. Packages move through the network by gravity feed. A switch setting may be changed only when the switch is empty. Packages may bunch up, and hence prevent a switch from being set until the entire bunch passes through. Finally, the destination of a package can only be physically sensed when it is in the source station (i.e., when it first enters the network) and when it reaches a bin; switches and chutes have no sensors for determining package destination. However, the presence of a package in a chute or switch can be sensed. The problem is defining a switch controller that will minimize package misrouting.

[5] We will use a stylized version of Gist to avoid introducing detailed syntax. See [9] for a more complete description of Gist.

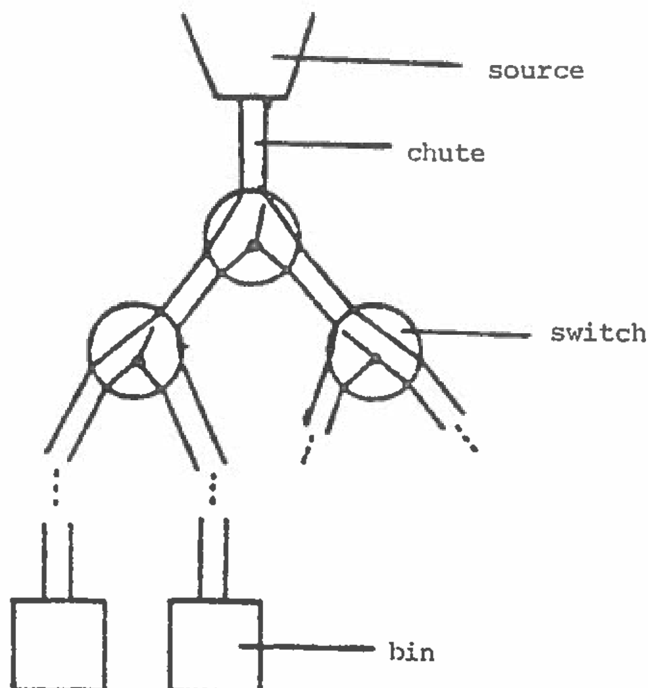

Figure 2.0: Package Routing Network

Glitter was used to develop the router specification into an implementation that controls the switches of the network. Here we will look at a snippet of that development. While the main points will be annotated, it is not expected that the reader will be able to follow all of the details of the development; our purpose is to give a flavor of a Glitter development session. Sections 3-5 will provide more detail.

To summarize briefly, the objects in the router domain are packages, chutes, switches, and bins. The operations are putting a package into the input chute, and changing the setting of a switch. Assume that the package-router specification-writer decided to include a specification of the entire sequence of packages that have ever entered the router, ordered by their arrival time. Suppose this sequence was to be called PHIST (short for PACKAGE_HISTORY). Below is a stylized Gist representation of this sequence.

> *relation* PHIST(packages | *sequence of* package)
> *definition* packages = (*sequence of* package) *ordered wrt* LOCATED_AT(package, source)
>
> *relation* LOCATED_AT(package, location)

The definition of PHIST contains several specification freedoms. First, it is a derived relation: it defines *packages* in terms of other data, namely the relation LOCATED_AT. Second, it makes use of historical reference: the ordering of packages is by their arrival time, a past event. Finally, it makes use of an efficiency freedom: there is no need to know *all* of the packages that have entered at any one time. That

is, certain portions of the specification (which have been elided) reference pieces of PHIST, but no portion requires the entire history; the specification writer has overgeneralized to promote clarity. All of these freedoms must be removed or mapped away during the development process. We will use Glitter to carry this out.

2.1. Start of session

Given a Gist specification to develop using Glitter, the organization of the development must be determined. This translates into deciding what order Gist statements should be tackled, i.e., in what order freedoms should be mapped away. Decisions made here can be crucial to both the success of a development, and the difficulty producing it. For instance, dealing with non-determinism before unfolding constraints can lead to with much wasted effort (London and Feather [9] look at this in more detail). While we hope to automate this ordering process at some time, it currently falls into the realm of insightful reasoning that we have placed with the user. Hence, as part of the user/Glitter partnership, the user is responsible for overall development organization.

Suppose that the user has decided that the relation PHIST is now ready for development. To use the Glitter system, he must have a means of stating this goal. Glitter supplies the user with a goal vocabulary for stating development goals. The vocabulary is discussed in detail in section 3. Here we will assume that the user chooses the *Develop* goal from the vocabulary:

>Develop PHIST[6]

As can be seen above, a goal may take an argument (more than one is possible). Arguments can be names of objects in the Gist specification, or other development related information. The system responds by finding methods for achieving the develop goal. To do this, it searches through its catalog of development methods. This catalog captures our current state of knowledge of software development using the Transformational Implementation approach. Each method in the catalog is indexed by the goal that the method helps achieve. Here, Glitter collects together all methods that are indexed by the goal "Develop".

Note that the Develop goal is high level (the highest currently): it requests simply that a Gist statement be mapped into a suitable implementation without specifying how this is to be done. We would hope that the methods associated with this goal would also be high level or strategic in nature, and general enough to be applied across statement types. In fact, this is exactly what we have found: a small set of strategic methods is capable of handling most of the Gist statements that occur as arguments to the Develop goal. This finding is quite encouraging. It leads us to believe that our method catalog will not grow to prohibitive dimensions, at least at the high end.

Glitter will find at least the following methods applicable to the development of PHIST: 1) a method that attempts to delete all or portions of a specification statement, 2) a method that maintains a derived relation, and 3) a method that rederives a relation on demand. We will call these methods DELETE, MAIN-

[6] The system's prompt is ">". Anything following is user typein.

TAIN, and REDERIVE respectively. DELETE is motivated by the knowledge that a specification writer will often include Gist statements that are a help in writing and understanding the specification, but have no real use in the implementation. Such statements may specify redundant or unused information. As we shall see shortly, PHIST overgeneralizes the package history; it includes unused information. However, the inclusion of PHIST in the specification is in no way erroneous, or even bad style. PHIST allows a clear description of the problem without getting bogged down in efficiency concerns.

The remaining methods, MAINTAIN and REDERIVE, are general methods for implementing Gist relations. Clearly these two methods should *not* be the first choice if the DELETE method can be employed here. That is, if PHIST is used in the specification as an understanding aid, then we should delete it now and move on to the next problem. Else, we should get to work on implementing it. Who will decide? Glitter contains a catalog of selection rules for choosing among competing methods. When all appropriate methods are gathered, Glitter finds all selection rules that might help in choosing among them. Each rule is applied to deny or support one or more of the competing methods.

In the case at hand, we wish to first know whether DELETE can be employed. Deciding whether a particular statement will be either redundant or unused in the implementation is not an easy task for either human or machine. Current selection rules exist for answering the question for very specific instances. One such rule deals with deleting relations that define a sequence of objects (which PHIST does). A paraphrase of the rule is

```
IF  1) a method M attempts to delete a relation X
    2) X is not referenced anywhere in the specification
THEN M will be successful
```

Note the analysis that must be performed to show that the antecedent of this rule is true. First, the rule must determine that PHIST is a relation. This analysis can be done easily by either human or machine by looking at the definition of PHIST. Next, the rule must search the specification for references to the relation. This process is quite tedious and error-prone when done by a human. At least one of the advantages of encapsulating selection knowledge in Glitter is the automation of mundane analysis problems, e.g., searching for references, objects, actions.

Does the above rule help us in our current selection? The first clause is true: there is a candidate method (DELETE) that, if chosen, will attempt to delete a relation (PHIST). The second one isn't: there is one reference to PHIST, shown in figure 2.1.[7] Hence, this rule can conclude nothing about the method DELETE. However, there does exist a more specific rule dealing with deleting sequences. It can be paraphrased as

[7]Notation: *obj:att* represents the value of the attribute (att) of an object (obj). The notation *rel(*)* represents the value of the starred argument, in 2.1 the sequence of packages.

IF 1) a method M attempts to delete a relation X
2) X is a derived relation that defines a sequence S
3) only the last-minus-N element of S is ever referenced
THEN M will be successful (only the end of the sequence is needed)

In the case of the router specification, the specifier used the sequence defined by PHIST to keep track of all packages that have entered the router. We can conjecture that he was not sure whether all would be needed when writing the specification, but at least some history would be required by other parts of the specification. Given the entire history, other portions of the specification could pick out what was needed. Notice that this is contrary to what we are taught as programmers. At the programming level, we are conditioned to be economical in both space and time. Whatever the reason for defining PHIST, it leads to a clear description, and hence is quite proper at the specification level.

```
...
    ;;;when a new package enters the network, hold it up if its destination is not
    ;;;the same as the previous package. This helps cut down on bunching, and hence misrouting.

if package.new:destination ≠ [package preceding package.new in PHIST(*)]:destination
then WAIT;
...
```

Figure 2.1: reference to PHIST

Lets look at the type of analysis performed by the new rule. Again, checking that PHIST defines a sequence can be done by looking at the definition. Evaluating the third clause – whether only the last-minus-N element (N=1 in this case) is referenced – is much more difficult. For the system to answer correctly (i.e., yes, it is last minus 1) in the router case would require an eight step reasoning process that is beyond the capabilities of the current system. However, following the partnership model, the system should do as much of the mundane analysis as possible. In this case that means finding each reference to PHIST, and verifying with the user that each reference is to the same element.

To reiterate, the system finds all references; the user verifies that each references the same relative position within the sequence. Since the left hand side of the rule evaluates to true, the right hand side action is taken. The only action currently defined for selection rules is that of adding a positive or negative increment to the weight of one or more methods in the candidate set. After all selection rules have run, the method with the highest weight is chosen for application.[8] Hence, the above rule will add some positive increment to DELETE's weight. Other rules will fire, and weight MAINTAIN and REDERIVE accordingly, e.g., maintaining the sequence is straightforward (other than size considerations), rederiving the entire package history on demand is at best intractable. The final outcome will be the system's selection of the DELETE method.

[8] This is a simplification. The system actually looks for a clear cut winner, one whose weight is far above all others. Failing this, it attempts to do more detailed tie-breaking.

The DELETE method simply states that to achieve a Develop goal whose argument is X, try achieving a Delete goal (another goal from Glitter's vocabulary) whose argument is X. In other words, you may be able to achieve a more general goal G1 by achieving a more specific goal G2. When the DELETE method is applied by the system, the delete goal becomes a subgoal of Develop.

We have now completed one cycle of the system: 1) post a goal, 2) find all applicable methods, 3) find all rules that help select among them, 4) apply the rules, 5) choose and apply the best method, 6) post new goals and repeat. Figure 2.2 represents this model graphically.

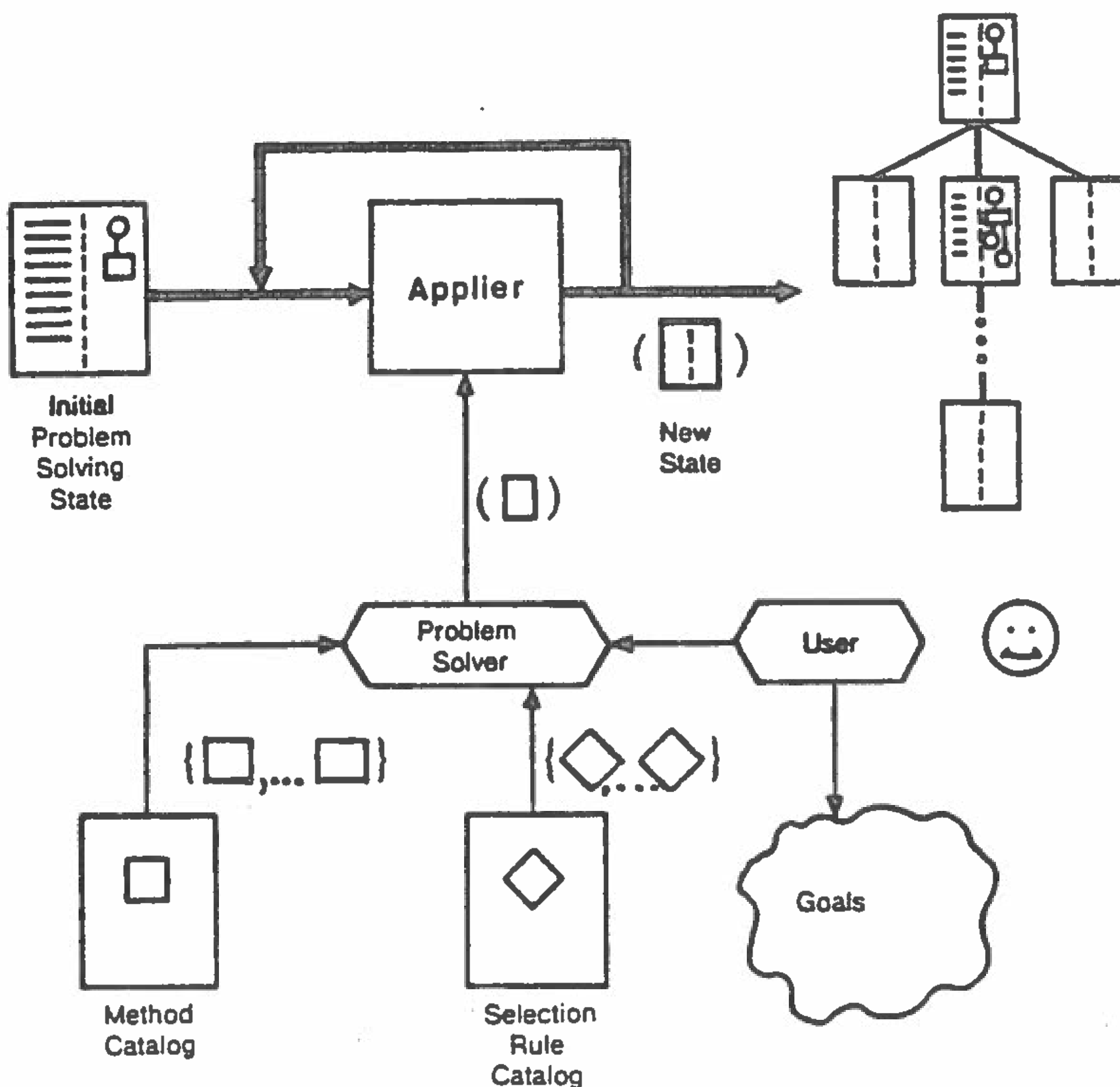

Figure 2.2: Glitter model

Note the automation lever here: the user posts strategic goals in the outer loop; for each goal, the system finds methods and transformations to achieve it in the inner loop. The final output is an exploration tree,

where each branch is an or-node, i.e., sibling nodes are alternative refinements of the parent node. Each node contains the current Gist program, and a record of the problem solving state that produced it. A development is a path from root node to leaf node. The next three sections focus on the major components of this model: goals, methods, selection rules.

3. Goals: the Development Vocabulary

The next three sections discuss the three major components of the Glitter system: goals, methods, selection rules. In this, the first, we look at goals.

In the Glitter model, the user is responsible for guiding a development by providing development goals. The system is responsible for providing the steps which achieve those goals. In this section we look at the Glitter representation of development goals, what development concepts goals must represent, and how the goal vocabulary may be extended.

3.1. Goal Representation

A *goal descriptor* as the formal notation for stating development goals in Glitter. A goal descriptor consists of a unique name,[9] a set of typed parameters or slots, and a predicate which tests whether the goal has been achieved. The goal predicate provides the semantics of a goal descriptor. It 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 predicate becomes true. This allows the flexibility of incremental achievement by the combination of several method applications (c.f., means-ends analysis).

Diagrammatically we have

Goal <unique name>
Slot-1: <Gist construct>
Slot-2: <Gist construct>
...
Predicate: <lisp code>

3.2. Characterization of the TI Development Space

The set of goal descriptors define the type of development problems that the system can work on. In the development of Gist specifications using Transformational Implementation, the major concern is with mapping away specification freedoms to produce efficient code. In general, there are three freedoms that we must deal with, *information, operation*, and *efficiency* (see [10] for related discussion).

---

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

3.2.1. 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 (cf. [12]).
- *Unfold* the code necessary to rederive the information at each place that makes use of the information.

In section 2, we saw two manifestations of these strategies in the methods MAINTAIN and REDERIVE. The corresponding goals are

Goal MaintainInformation
Slot1: a relation

Goal DeriveInformation
Slot1: a relation

3.2.2. Mapping Operation Freedoms

A Gist specification may contain non-deterministic choice points, which allow the specifier to indicate equally acceptable alternatives. The constant companions of non-determinism are constraints, which allow the specifier to declaratively 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 [8] uses this strategy (see also [13]).

The following goal, among others, deals with constrained non-determinism:

Goal Reformulate
Slot1: a constraint
Slot2: <demon template>

Note that as specified, the above goal is specific to handling constraints. In reality, it is more general: Reformulate will accept any Gist construct in its first slot, and any template in its second.

3.2.3. Mapping Efficiency Freedoms

There are two efficiency concerns that must be dealt with. The first centers on Gist's role as a problem description language. A problem description should contain what ever information is necessary to convey the objects, operations and constraints of the problem. In particular, a Gist specification writer need not be concerned with the efficient ordering of operations,[10] the elimination of unneeded (e.g., redundant, unused) information or operations, the sharing of information or computations among program parts. Only if any of the above make the problem description less clear should action be taken to remove them, *before* development commences. The relation PHIST introduced in section 2 is an example of a Gist efficiency freedom. Once development started, it was recognized that the relation was unneeded, and hence it was deleted.

The second efficiency concern is the choice of concrete representations for abstract Gist constructs. For example, is it more efficient to maintain a derived relation or rederive it on demand? Is it more efficient to backtrack when a choice is made that violates a constraint, or only generate those choices that do not violate the constraint? This type of efficiency knowledge resides in Glitter's selection rules, which are discussed in section 5. In this section we deal with only the first efficiency concern, mapping away efficiency freedoms.

Glitter supports three basic efficiency freedom mappings:

- Efficient ordering of operations by making non-deterministic control deterministic and resequencing actions.
- Removal of unused information or operation structures.
- Sharing of like parts among compound structures by consolidation and factoring.

These clearly do not cover all efficiency concerns. For instance, [14,15] prescribe a set of techniques for squeezing the last ounce of power from a program. Because these techniques are based on having a concrete algorithm in a standard programming language, we expect that they will be applied after the development process.

Some examples of Glitter goals dealing with mapping efficiency freedoms are given below:

---

[10]Clearly if the problem itself calls for a particular ordering then the specification must reflect it.

Goal Determinize
Slot1: a non-deterministic action A

Goal Delete
Slot1: a Gist construct C

Goal Consolidate
Slot1: a construct C1
Slot2: a construct C2

3.3. Other Development Concerns

The mapping away of specification freedoms can be viewed as the flashy part of a development. Unfortunately, there are other mundane chores that must be performed as well. These include, establishing applicability conditions on correctness-preserving transformations, transforming the program into a state where another transformation can be applied, simplifying the program, restructuring the program to ease development. Each of these concerns must have an associated goal vocabulary.

3.3.1. Applicability Conditions

Most Glitter methods rely on some program or domain property holding before they can successfully be applied. 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 [16] automates this process through the use of an automatic subgoaling mechanism. Barstow [17] further speculates on the automation of condition proving in a knowledge-based system. The freedoms afforded by Gist make the construction of a general purpose property prover an unlikely prospect in the foreseeable future. Glitter currently has no automatic means of proving the applicability conditions of methods, hence it becomes the purview of the human half of the partnership.

There is a single goal defined for stating conditions:

Goal Show
Slot1: a predicate P

The achievement condition for the Show goal is a Glitter-defined function called PUNT. PUNT specifies that an instance of the goal is achieved only when explicitly stated by the user. Here, it is up to the user to determine if P can be shown to hold, and tell the system to mark it as such.

3.3.2. Jittering

We define jittering[11] to be the process of transforming the current program state to match the state required by some development method. Typically, we decide that we want to apply some method M that is inapplicable in the current state. We jitter the state until M can be applied.

In Glitter, jittering is made part of the overall problem solving process. Jittering subgoals are posted automatically by the system, and methods are defined for achieving them. The jittering vocabulary provided by Glitter is an offshoot of some earlier work on a means-ends problem solver for TI [18]. Some examples from the jittering vocabulary follow.

Goal Equivalence
Slot1: a construct C1
Slot2: a construct C2

Goal Swap
Slot1: a construct C1
Slot2: a construct C2

Goal Reformulate
Slot1: a construct C
Slot2: a template T

3.3.3. Simplification

As a Glitter development progresses, the intermediate forms of the program tend to become messy and hard to read. A major part of this has to do with the movement of code from one context to another. 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 is a deleterious interaction between jittering and simplification. Methods used to achieve a jittering goal will often change the program into unsimplified states as part of their means of satisfying the goal. In other words, they may move one step back to move two steps forward. If simplification is carried out after every step, then the effects of some jittering will be lost, i.e., a method will jitter, a simplification method will unjitter, and hence the jitter goal will never be achieved. For example, a jittering goal may be posted to change an action statement into a conditional (because some other method only works on conditional action statements). There is a method in the catalog that can be paraphrased as

[11] Current usage is now *conditioning*. However, we will stick with the older term.

EmbedInCond: *action* ⇒ **if true then** *action*

However, there is also the following simplification rule:

TrueCond: **if true then** *action* ⇒ *action*

Systems that apply simplification rules immediately after all development steps (see for instance [19]) cannot allow the above type of jittering. Glitter finesses the problem by requiring explicit application of simplification steps, i.e., the user must post a simplification goal to activate simplification rules.

3.3.4. 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. Both of these address not program efficiency, but problem solving efficiency, whether it be by human or machine.

Casify is an example of a pragmatic goal, which breaks a construct into smaller cases:

Goal Casify
  Slot1: a construct C

3.4. Robustness

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

(1) The user should be able to move among all descriptive levels, from stating abstract mapping goals to naming particular transformations to be applied.

(2) The user should not be restricted to a particular development organization arbitrarily imposed by the system. The current system errs in the other direction by providing minimal organizational guidance.

(3) There may be two or more equivalent ways of viewing a problem. Given that the goal language supports multiple descriptions, a means of mapping each onto the known development techniques is needed. Both Mark [20] and Mostow [21] discuss related problems.

(4) Glitter provides general, *domain-independent* mechanisms. A means of incorporating *domain-specific* information into a development should also be provided.

Each of the above is supported by Glitter. The first relies on the Glitter goal vocabulary as we've presented it, plus two more descriptors for gaining more fine grained control of the system. In particular, Glitter supplies a *Use* goal descriptor which allows the user to name a particular method to invoke, and a *Manual* goal descriptor which allows the user to manually edit the program.

For the second item, organizational restrictions, the user is free to choose the portion of the specification to work on, and as in (1), the level.

The third item relies on having a set of "translation" methods that map alternative task descriptions onto the known techniques. While it is hard to say that the current system has a complete set of such methods, the current set has been sufficient for the developments studied. Because it is easy to add new methods to the system, we see no problem adding new translation methods as they become necessary.

The last item deals with the inclusion of domain-specific information within the development. It is likely a development will involve goals specific to that development or application. For instance, in the package router problem, the user may wish to post a goal *OptimizePackageUsage*. Glitter provides the user with a facility for defining development-dependent goals, such as this, which remain defined throughout a particular development. User defined goals take the same form as Glitter goals. Their construction is carried out interactively at the point in the development where they are needed. 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 into ad hoc methods. Once a domain-specific goal G has been posted, all subsequent goals posted become subgoals of G until the user marks G as achieved (domain-specific goals all are defined with the PUNT achievement checker; see section 3.3.1). The use of ad hoc goals both organizes a development, especially when a large number of steps are necessary to carry out some task, and documents the problem solving process for maintainers of the system.

4. Development Methods

In the previous section we looked at one component of the Glitter problem solving model, the vocabulary for stating development goals. In this section we look at another component, the methods necessary for achieving those goals. We will first present the important properties of representing development knowledge in the TI domain. Next, we will present Glitter's method representation, and demonstrate how it can be used to capture development knowledge. Below is a summary of the major points made in this section.

Knowledge should be accessible.

> 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.

The entire planning space must be covered.

> In Glitter, methods must represent knowledge about the manipulation of not only the *program* space

(i.e., transformations), but 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, Glitter's selection rules require information about competing methods including the actions they propose to take, and the particular goals that triggered them.

4.1. Knowledge Accessibility

Given a problem description (development goal), we would like to find *all* relevant methods for solving the problem. A system that forces the user to manually search a 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 posted, all methods indexed to it are are formed into a candidate set. 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 [22] 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 [19] uses *meta-rules* to derive information about which new transformations will be applicable after a particular transformation has fired.

We view both of these capabilities as useful extensions of the current Glitter system. Both rely on a form of representational transparency discussed in section 4.5.

4.2. Adding new knowledge

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.

The second aspect is what McDermott refers to as *additivity* [23]: 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 [19,24,25]. They pay the price in additivity: the addition of new knowledge to these systems requires a re-organization of the knowledge base.

4.3. Coverage of the development planning space

Glitter's 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 implement a complex demon then try splitting it into several simpler demons and implementing each individually (i.e., post a Casify goal).*

An example of the former is:

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

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 achieves the Casify goal posted by the first.

4.4. Automatic Jittering

Once a user finds a transformation he would like to apply, it is often the case that the transformation's left hand side does not match the current state exactly, i.e., subgoaling is necessary. Glitter provides for this. In Glitter, given that a particular method has been judged appropriate for achieving the current development goal (i.e., selected by the user or the system's selection process), jittering steps necessary to apply the goal will be automatically generated.

The chronology of building an automatic jittering capability in Glitter is of interest. A predecessor of Glitter called the Jitterer [18] used a GPS style control structure 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 [26], 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 forerunner of Glitter's goals). 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 was a major problem with this approach: 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.[12]

Because of the above problem, 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 post the goals needed to produce its pattern, *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 section 3, Glitter goals are independent of methods and allow a more general GPS type of control if necessary.

The next three sections describe some further aspects of Glitter's approach to jittering.

4.4.1. 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:

[12] We have stressed the word general above. 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.

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 they can be consolidated into a single demon.

Suppose that this method has been selected to consolidate two constructs S1 and S2, i.e., D1 is bound to S1, and D2 to 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. Section 5 discusses the system's selection knowledge in more detail.

4.4.2. Restraint

A positive consequence of the eagerness property is a collection of methods with wide applicability. That is, methods can represent general development knowledge without being tied to specific cases. 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 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* currently lack precise definitions. Using the MergeDemons example, it is unlikely that the method should be attempted if D1 and D2 are not initially demons: reformulation of non-demonic constructs into demonic ones is a dubious undertaking.[13]

If a method is erroneously filtered out (i.e., overrestrained), 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.

---

[13]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.

4.4.3. Level of Effort

We have described eagerness and restraint 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 an important future project.

4.5. Representational transparency

The Glitter problem solving model is based on the user playing an active role in development 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 [27] 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, which names a program transformation to apply. 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.

In section 5, we will present examples of method analysis during the selection process. In the next section we define the method notation used in Glitter.

4.6. Method Template

A Glitter method represents development knowledge. A method template takes the following form:

Method <unique name>
Goal: <development goal>
Filter: [<boolean expression>]
Action: <development action>
End method

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 (see section 4).

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 5.2). The filter provides a hook for non-subgoalable pre-conditions on a method (see section 4.4.2).

Action field: filled with an ordered sequence of one or more *development actions*. A development action is either a subgoal to be posted, a transformation application, or the action-mapping function *forall*, which maps one or development actions onto one or more components of a structure.

Actions are initiated after the method is a) triggered, b) filtered, c) added to the candidate set and d) chosen by the selection engine (see section 5). When all actions have successfully completed, the method in turn 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 system collects a new set of candidate methods, and the selection process starts anew..

When adding a new method to the method catalog, the construction process below is used. While this process must be currently carried out manually, Chiu [26] discusses ways it might be automated in the future.

(1) Translate implicit intent into an explicit development goal. Make this goal the trigger of the method.

(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 4.4.1).

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

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

(5) Incorporate any local constraints that are possible (see section 4.4.2). 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 will first apply the above method-building process to 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.

A slightly sugared notation will be used when presenting the fillers of the various method fields.

(1) *Goal definition.* The effect of this method is to *Consolidate* two demons (*Consolidate* is a built-in Glitter Goal); this is made the goal of the method.

```
Method MergeDemons
  Goal: Consolidate D1 and D2
  Filter: ...
  Action: ...
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 boldfaced items are pattern variables, others are literal):

```
demon D1 (vars)
   trigger t
   response r1
...
demon D2 (vars)
   trigger t
   response r2
```

Represented as subgoals, we get the following:

a. *Reformulate* D1 as demon

b. *Reformulate* D2 as demon

c. *Equivalence* triggers of D1 and D2

d. *Equivalence* declared variables of D1 and D2

The system has a vocabulary for talking about portions of a Gist specification. The terms *triggers of* and *declared variables of* are examples.

We now have

```
Method MergeDemons
  Goal: Consolidate D1 and D2
  Filter: ...
  Action-1: Reformulate D1 as demon
  Action-2: Reformulate D2 as demon
  Action-3: Equivalence triggers of D1 and D2
  Action-4: Equivalence declared variables of D1 and D2
  Action-5: ...
end method
```

(3) *Subgoal on applicability condition.* The applicability condition of MergeDemons is **mergeable_demons**, a built-in Glitter predicate that checks whether the two responses r1 and r2 can be interleaved to form the new response. After defining the corresponding *Show* goal (see section 3.3.1) we have:

```
Method MergeDemons
  Goal: Consolidate D1 and D2
  Filter: ...
  Action-1: Reformulate D1 as demon
  Action-2: Reformulate D2 as demon
  Action-3: Equivalence triggers of D1 and D2
  Action-4: Equivalence declared variables of D1 and D2
  Action-5: Show mergeable_demons(D1, D2)
  Action-6: ...
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 finally deletes the two old demons and inserts the new. The procedure is made the argument of an *Apply* goal.

```
Method MergeDemons
  Goal: Consolidate D1 and D2
  Filter: ...
  Action-1: Reformulate D1 as demon
  Action-2: Reformulate D2 as demon
  Action-3: Equivalence triggers of D1 and D2
  Action-4: Equivalence declared variables of D1 and D2
  Action-5: Show mergeable_demons(D1, D2)
  Action-6: Apply demon_merge(D1, D2)
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 is a demon
Filter-b: D2 is a demon
Action-1: Equivalence triggers of D1 and D2
Action-2: Equivalence declared variables of D1 and D2
Action-3: Show mergeable_demons(D1, D2)
Action-4: Apply demon_merge(D1, D2)
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]
Action-2: Equivalence declaration-of[D1] and
declaration-of[D2]
Action-3: Show mergeable_demons(D1, D2)
Action-4: Apply demon_merge(D1, D2)
end method

The previous example showed a method that mixed jittering and program transformation. The second example shows a method that operates strictly in the problem space. The method, called MaintainDerivedRelation, can be stated as follows:

MaintaintDerivedRelation:
If the goal is to develop 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 *Develop* a derived-relation.

Method MaintainDerivedRelation
Goal: Develop DR|**derived-relation**
Filter: ...
Action: ...
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 Develop goal into a more concrete goal, i.e., incrementally maintain the relation.

> **Method** MaintainDerivedRelation
> *Goal*: Develop DR|**derived-relation**
> *Action*: Maintain DR
> **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 methods such as the DERIVE method introduced in section 2. This selection knowledge is defined in terms of selection rules, and will be applied during method selection as described in section 5. The second piece of selection knowledge notes that it is not useful to attempt to incrementally maintain a relation which is unchanging, e.g., the static relation representing the package router connection matrix of chutes and switches. 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. Since all filter predicates are checked before the selection engine is invoked, there can be no control over the computation of the static predicate.

> **Method** MaintainDerivedRelation
> *Goal*: Develop DR|**derived-relation**
> *Filter*: non-static(DR)
> *Action*: Maintain DR
> **end method**

Each of the methods in Glitter's catalog was defined using similar steps. Currently the catalog contains approximately 75 methods.

5. The Selection Process

This section presents the third and last major component of the Glitter model, the selection engine. During a Glitter development, there arise various points where selections must be made:

(1) Given one or more competing methods, we must decide which <u>if any</u> should be selected.

(2) Given an unachievable goal, we must decide what previous problem solving state the development be should backed up to.

(3) 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 selection

knowledge lies in area 1; in this section we will describe how this knowledge is represented and used. Glitter currently relies on the user to make selections in areas 2 and 3. However, [1] discusses how selections in these areas might be automated in the future.

Before delving into the details of Glitter's selection process, we will summarize the important points made in this section:

Method selection is a partnership task.

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, insightful reasoning still falls on the shoulders of the user.

The system records its mistakes.

The system monitors the actions of the user to detect its own selection mistakes; when the user undoes or overrides a system decision, the system records the necessary context to allow future knowledge maintenance.

The problem solving structure is accessible.

Glitter's selection process 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 are introduced here as useful concepts.

5.1. 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 Glitter.

Selection Process Summary

(1) **Goal G is posted.** If G is satisfied in the posting state then it is marked as trivially achieved.

(2) **Initial method candidate set is formed.** Given that G is not trivially achieved, all methods that are indexed to G, and whose filter predicates evaluate to true, are placed in the initial candidate set.

(3) **Method agenda formed.** Selection rules are run to form an ordered agenda of weighted candidate methods.

(4) **Method chosen from final set, and applied.**

We will follow the steps in the selection process in more detail in the following sections.

5.2. 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 [27]). If this set turns out to be empty the user is informed and control reverts back to him. The empty candidate set is an interesting case which will look into in more detail in section 5.8.1.

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 [29]). Davis [27] 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 section 4.4.1), the initial set may be saturated. However, Davis fails to mention another aspect of method scheduling: the problem solving cost of applying a method. Even in cases where only a few methods are competing, their individual resource costs may be large. For example, assuming that the user's time and knowledge are viewed as resources, then an interactive system like Glitter must be concerned with both timely response, and knowledge utilization. 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. Hence, Glitter cannot be "gone" for arbitrary lengths of time. Further, the user may need to get involved with lower-level problem solving to supply information unavailable or uncomputable by the system, e.g., applicability conditions on transformations. Using an exhaustive search, the user may be asked to perform a number of tedious reasoning steps, few of which will may be relevant to the final choice.

5.3. Forming the method agenda

The next step is to from the method candidate set into an ordered agenda. This is accomplished by running selection rules. Such rules embody knowledge on how useful a method is in achieving a particular goal, and how it compares with competing methods. The form of selection rules is

Selection Rule <unique name>
    IF   <selection expression>
    THEN   <ordering action>
End Selection Rule

The fields of a selection rule are broken out as follows:

<unique name>: provides a unique textual handle.

<selection expression>: in general, some problem solving event such as a method joining the initial candidate set or a goal becoming active.

<method ordering action>: a weighting or ordering action

5.4. Method chosen

The system selects the highest ranked method from the agenda, and applies it.

If everything runs smoothly then the system will start the selection process anew, and continue until the user's goal is achieved. What is generated is an augmented AND/OR tree. The root of the tree is the user's goal (an OR node). Under this node are all of the methods that triggered on the goal, plus all of the selection rules that were used in weighting and ordering the set. One of the methods is chosen and applied, and it in turn will generate one or more new subgoals (an AND node). The process is repeated with new candidate methods being collected under each goal, and selection rules run to choose the best.

The user has complete freedom in traversing the AND/OR tree. Once gaining control, the user can move to any node, examine what methods are competing, which one was chosen, and what selection knowledge lead to it being chosen. At this point, the user can choose to follow (select) one of the alternative candidate methods. The system will generate a new problem solving context in which to explore the method.

5.5. Profiting from mistakes

Glitter is based on an interactive, partnership model of problem solving. Hence, we expect that the user will be actively involved during development. However, our goal is to have the machine gradually take on more and more of the development task. At least part of the knowledge shift from user to machine will be brought about by fleshing out both the method and selection rule catalogs. This in turn will brought about by attempting more, and a bigger variety of developments. As other developments are tackled, we expect deficiencies in the catalogs to become apparent. As they do, we would like Glitter to attempt to 1) ascertain what type of knowledge is missing, and 2) record the context so that the missing knowledge can be added later.

5.5.1. Missing methods

Suppose that no methods are found for a posted goal, i.e., the initial method candidate set is empty. There are two possible causes: 1) the goal is unachievable, or 2) one or more methods for achieving the goal are missing from the method catalog. To determine which is the case, the system monitors the next user action.

- If the user next does a manual editing operation on the program, the system infers case 2, i.e., a program transformation is missing.
- If the user next posts a subgoal, the system infers case 2, i.e., a method reduction is missing.
- If the user switches to an alternative problem solving context (backtracks), the system infers case 1, i.e., a deadend has been reached.

The system monitors for each of the above. For the case that the user fills in some actions of his own, the system records the state and actions. Frequently these actions can be generalized into a method or transformation to be included in Glitter's catalog. In particular, the goals posted by the user at the top-level are viewed as comprising an ad hoc method. Hence, the user's organization of a development is automatically considered for inclusion in the method catalog as a general method for developing Gist specifications.

For the case that the user backtracks, the system records 1) the deadend state, and 2) the state backtracked to. At the end of development, the system computes what states 1) were backtracked to, *and* 2) are also on the final solution path. These states are listed as likely candidates for stronger selection knowledge. The assumption is that some missing (or weak) piece of selection knwoledge allowed a wrong path to be taken intitially, causing backtracking.

5.5.2. Missing selection rules

Suppose that the user overrides the system's selected method. There are two possible causes: 1) the system's choice is wrong, or 2) the user is wrong. The system assumes the former. Any time the user overrides the system's method selection, the context is recorded. The assumption is that either a) an existing rule added an inappropriate weight, b) a selection rule is missing that would have lead to the correct method being chosen, or c) both. Currently it is up to the catalog maintainer to analyze which is the case.

5.6. Design decisions

Much of Glitter's selection knowledge focuses on producing an efficient final implementation, i.e., the selection of appropriate data structures and control structures to map away Gist freedoms. For example, the following (paraphrased) selection rules are found in Glitter's catalog:

MapDerivedRelation:
Given the goal is to map a derived relation R, and given a choice of maintaining R or recomputing R, choose maintenance over recomputation when "the cost of recomputing R" is greater than "the cost of maintaining R".

MapConstraint:
Given the goal is to map a constraint C, avoid unfolding C when backtracking is costly/impossible.

Both of the above rules point out a major benefit of a partnership model such as embodied by Glitter: problem solving can be cooperative. Glitter's analysis routines are not powerful enough to evaluate the above rules in their entirety. For instance, determining the recomputation costs of a derived relation, or whether backtracking is possible in a particular domain is beyond the capabilities of the system for the general case. What Glitter does do is to compute all rule conditions within its power, and then ask *focused* questions of the user. In the above rules this means finding all points a relation is referenced and changed, or finding all points a constraint will need to be unfolded. This low level analysis is well suited to the machine. A more concrete example is given below.

Suppose the current goal was to *Equivalence*[14] expression E1 and expression E2. One candidate method is as follows:

> **Method** Anchor1
> *Goal*: Equivalence E1 and E2
> *Action*: Reformulate E2 as E1
> **End Method**

Anchor1 attempts to make two expressions equivalent by making the second conform to the first. Other methods would be competing here: Anchor2 which makes the first conform to the second expression; Anchor* which makes both conform to some new third expression. The selection process is particularly important here. Each method is in some sense overeager, having no filtering knowledge. If chosen indiscriminately, any can lead to long detours ending in dead-end states. The following selection rule provides some discrimination:

> **Selection Rule** BuriedInDefinition
> *IF* 1) method Anchor1 is a candidate
> 2) E2 is a reference to a defined object D
> 3) the definition of D is reformulatable as E1
> *THEN:* Anchor1 is likely to succeed
> **End Selection Rule**

The above rule embodies the following heuristic:

> The goal is to make two expressions equivalent. A method M exists for fixing one of the expressions (E1) and attempting to make the other (E2) conform. It is known that references to defined objects can often be "unfolded", i.e., an instantiated copy of the object can replace the reference. If E2 is a reference to such an object then check if the object is something that can be reformulated into E1. If so, then reward the method M; there is a good chance of it succeeding.

Glitter can compute the first two conditions of the rule, and hence avoid asking the user questions such as

---

[14] Another goal from Glitter's development vocabulary.

Is Anchor1 a candidate method?

Can the body of the definition associated with E2 be reformulated as E1?

The first question shows an assistant who is unable to analyze portions of the its own planning space. The second question shows an assistant who is unwilling to do mundane analysis, e.g., finding the object that E2 references and printing it out as opposed to forcing the user to search for it. The actual question Glitter asks the user is

Can <object referenced by E2> be reformulated as <expression E1>?

For example, suppose that the goal was to make the two expressions below equivalent:

E1: not located_at(p|package, s|switch)
E2: empty(s|switch)

To evaluate the LHS of rule BuriedInDefinition, Glitter first checks to see if the method Anchor1 is a candidate. Assume that it is. Glitter next checks to see if E2 is a reference to a defined object. Assume that it is as given below:

relation empty(switch)
definition not E package | located_at(package, switch)

Since the answer is yes, Glitter must determine whether the body of the relation "empty" can be reformulated into something matching E1. Glitter is unable to do this in the general case, and so would ask the user to determine if the body is transformable to E1:

Can
not E package | located_at(package, s|switch)
be reformulated as
not located_at(p|package, s|switch) ?

We expect it is possible to answer questions like the one above by machine *in special cases.* In the above case, a rough hewn rule might notice that the target expression E1 appears in the definition of empty. A more polished rule might note that a negatively-quantified existential can be transformed into a negatively-quantified ground instance. Knowledge of this sort would relieve the user of at least some low level reasoning; it is one type of knowledge that we expect to incorporate into future versions of the system.

5.7. Pragmatic Rules

Selection rules also deal with the pragmatics of a transformational development. For instance,

> **CasifyWhenInDoubt:**
> *Choose a method that will break the current expression into simpler cases when no other methods look promising.*
>
> **MergableDemon:**
> *If a demon D triggers randomly, then any method that attempts to merge D with another demon should meet with success.*

Note that both of the above rules reference a method by description rather than by name, i.e., "a method that casifies" or "a method that merges" as opposed to specific methods like "CasifyDemons" or "MergeDemons". Glitter's representation of methods allows selection rules to analyze a method's fields to answer questions like this.

5.8. Planning Rules

There is another component to Glitter's selection knowledge that relies on planning as opposed to program features. It focuses on producing an efficient development process. It uses the problem solving context to avoid following circuitous routes and blind alleys. It relies on being able to analyze the current planning state which includes 1) candidate methods, 2) the current goal, and 3) the planning history that got us to this point, i.e., the AND/OR goal tree.

As in any problem solving domain, it is sometimes difficult to select among competing methods without knowing the overall goal or goals being pursued. Wilensky [28] 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.

Certain Glitter selection rules use 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". Note, however, that Glitter has no explicit representation for meta-goals; the above two are only present implicitly. On the other hand, some of Glitter's rules do act like meta-plans. One such rule recognizes a situation where there exists a supergoal of deleting some construct C. It rewards a method which avoids scattering C throughout the

program, and punishes one that does. Using an analogy, suppose that Mary wants to haul (remove) a bag of tin cans to the dump[15] , but the bag is too big to fit in her wagon. She has several options:[16] 1) stomp (reformulate) the bag until the cans are flattened to an acceptable size, 2) scatter the cans on the ground, and use multiple trips to haul them in acceptable subsets. Mary, being a bright meta-planner, recognizes that the second will make the job much harder, while the first is relatively easy.

6. Summary

There are four automation issues that Glitter addresses: 1) formalization of the development process, 2) detail management, 3) man/machine partnership, and 4) documented history of the development process. We will look at each in turn.

6.1. Formalization

In using a problem solving approach, at least three things must be formally defined: a notation or vocabulary for stating problems; a notation or vocabulary for describing techniques for solving those problems; a notation or vocabulary for describing rules for selecting among competing techniques. The question is how well has Glitter done in defining each.

Problem Vocabulary

Glitter's goal descriptors provide a problem vocabulary. Each goal is built to be both problem independent, and handle a general class of development concerns. If a user wishes to state a goal that either is problem or development specific, Glitter allows an escape mechanism through user defined goals, goals that can be defined dynamically as a development progresses.

A goal both draws its power, and inherits its weakness, from its Lisp achievement condition. The power comes from using a functional language as the basis for describing achievement semantics. Arbitrarily complex conditions can be set up for defining goal achievement. The weakness comes from the inability of the system to reason directly about a goal. Because the lisp code is opaque to the system, any analysis of the semantics of a goal must rely on built-in descriptions.

Method Vocabulary

Glitter's methods are the repository for problem solving techniques in the TI world. Methods have a relatively simple form: index, filter, actions. All but a few actions are analyzable by the system. The construction of a method is based on configuring pre-defined components, i.e., goals for the index, Gist-specific predicates for the filter, goals and transformations for the actions.

The simplicity of a method leads to its weakness: complex control and planning knowledge often

---

[15] In the dark ages before recycling centers.

[16] Maybe the most obvious is to get someone else (a graduate student?) to do it.

cannot be easily represented. For instance, we might want to encapsulate the technique "try this action, and if it fails then try that action" in a method. This type of conditional problem solving is not easily represented in Glitter.[17] Since our partnership model centers on playing off strengths and weaknesses, this Glitter weakness is mitigated by relying on the user to supply complex problem solving organization.

Selection Vocabulary

As with methods, the representation of selection knowledge in Glitter takes a simple form. As with methods, a rule is constructed out of pre-defined components, i.e., the LHS from built-in functions that access the method candidate set and the planning tree, the RHS from weighting and ordering functions. While representational simplicity makes it easy to construct new rules and understand existing ones, as with methods, it leads to weaknesses. For instance, it is quite hard to rationalize weights associated with rules. What does it mean for a method to be given a +3 by one rule and -3 by another? Does the resulting 0 mean that nothing is known about the method? Overall, the weight calculus used to build the agenda is under-defined.

Another weakness of Glitter's rule-based representation is its reliance on surface features of a problem. In Kant's system [31], we find surface rules augmenting a more formal analysis model, a model that is capable of following (i.e., applying) several competing methods down to some depth before deciding which is best. Such a model is clearly more powerful, however much more difficult in the Gist/TI world: the PECOS/LIBRA specification language is at a much lower level than Gist.

6.2. Detail management

On the largest development attempted to date, the package router development, 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 non-automation of these steps in the TI model 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.

6.3. Glitter as a development partner

In an ideal partnership, the strengths of one 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

---

ADDLE can handle this case and a wider variety of control strategies in general. However, this is offest by PADDLE's inability to reason about control directly, e.g., there are no meta or scheduling rules in PADDLE.

individually. We will show below that Glitter has gone a long way towards meeting this goal.

- *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.
- *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 changed. The user is likely.to find these details tedious to compute and easy to miss.
- *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 overall development organization. Our experience base is weak in the area of high level organizational knowledge.
- *The user* provides insightful reasoning. While progress is slowly being made in automated reasoning, the complexities of Gist put this beyond the capabilities of the machine.
- *The user* provides unavailable information to the selection process. 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.
- *The user* is responsible for exploring the development space, e.g., backing up from dead-end or losing development paths. Sophisticated control is lacking in Glitter. The user is relied on to compensate.

6.4. The development history

The output of Glitter is the full development exploration tree as pictured below in figure 6.4 (and figure 2.2). 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.

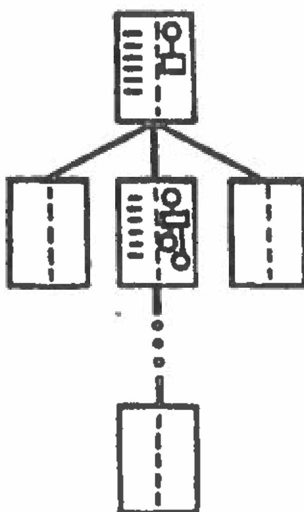

Figure 6.4: Development exploration tree

Each node in the tree represents a particular problem solving state. Figure 6.5 shows the problem solving portion of one node where G-current has been posted, and the methods for achieving it have been gathered in set S. The next step will be to call the scheduler to choose a method among the competitors in S. Note that the state includes, working from the bottom-up, 1) the current active goal, G-current, 2) the methods competing to achieve it, set S, 3) the method M applied to generate G-current, 4) the supergoal G-super that M was chosen to achieve, 5) M's competitors, 6) the selection rules that were used to choose M from among its competitors, 7) the rest of the goals, methods, and selection rules that lead from the user-supplied root goal to this point. In general, we wish to provide the user with a "you-are-there" perspective: any state can be chosen from the exploration tree, and the user can see exactly what goal was active, what methods were competing, what selection knowledge was available. From this point the user is free to further explore an existing arc/path, or strike out on his own by generating a new path. In the next section, we will see how Glitter's form of development rationalization can be used by the machine as well.

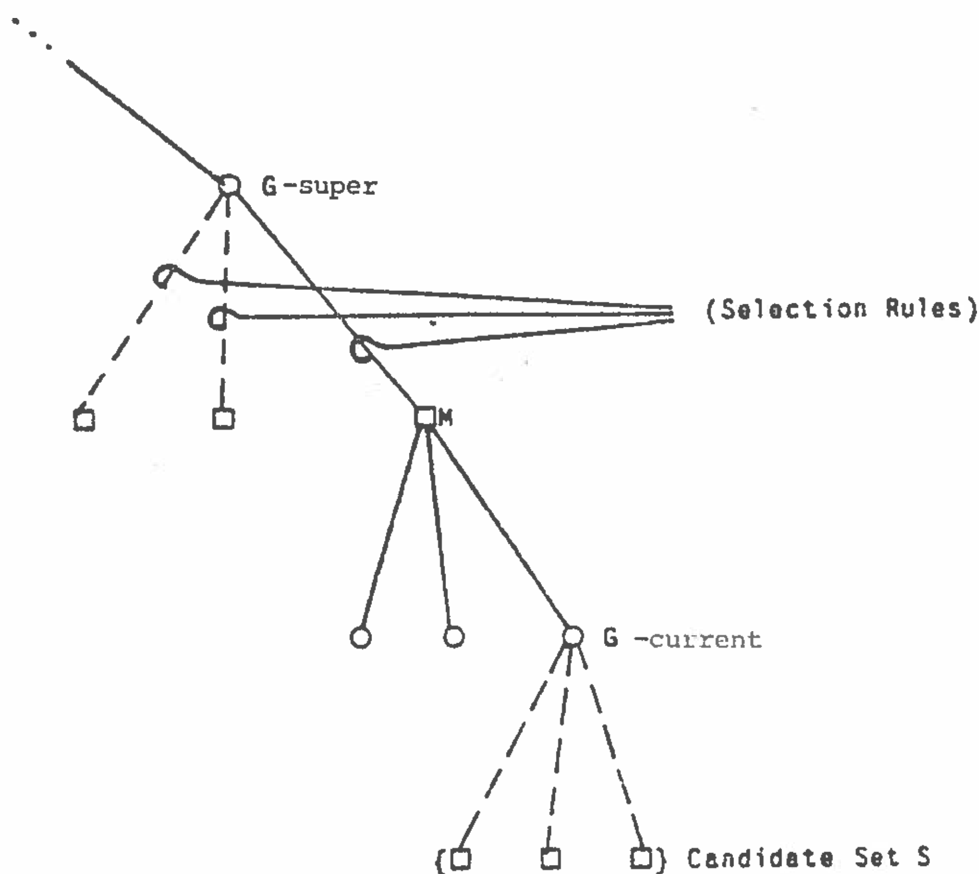

Figure 6.5: Problem solving state

7. Future Work

We are currently exploring two other areas of transformational development where the Glitter model might be applied: specification construction, and maintenance. In this section, we will look at both.

7.1. Applying Glitter to maintenance

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 more detail at how such a product might be used in an important area of the software lifecycle, software maintenance. We will present an example of a modification to the package router development to accommodate a specification change. We stress that this is strictly speculative; no maintenance tool currently exists.

Suppose we notice the following:

> The package router specification specifies that a package entering the network should be delayed if it does not have the same destination as the the most recent package to enter the network; this prevents problematic bunching. However, statistics gathered over time have shown that consecutive packages rarely have

the same destination. Hence, most packages entering the router will be delayed. A decision is made to change the specification: all packages will now be delayed unconditionally. Note that this is a specification modification as opposed to a development step.

To achieve this modification, assume that the following *specification transformation* is made (section 7.2 discusses a specification tool for making such modifications):

Old Spec:

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
 trigger package.new:located_at = source
 response
  begin
      ;;;when a new package enters the network, hold it up if its destination is not
      ;;;the same as the previous package. This helps cut down on bunching,
      ;;;and hence misrouting.

    if package.new:destination ≠ [package preceding package.new in PHIST(*)]:destination
    then call WAIT[];
   ...
  end;
```

New Spec:

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
 trigger package.new:located_at = source
 response
  begin
      ;;;Delay new packages to avoid bunching, and hence misrouting

    call WAIT[];
   ...
  end;
```

Remember that when using the TI model, any change to the specification requires a redevelopment. Naturally, we would like to reuse as much of the old development as possible in reimplementing the new specification. The original development commenced when the user posted a goal to Develop the relation PHIST (see section 2). Assume our maintenance tool starts by trying to repost this goal as the first step in the new development. The goal is still valid, as is the method chosen to achieve it, DELETE (again, see section 2). That is, both goal and method reference the *definition* of the relation PHIST which is still around (but not shown above).

Assume the maintenance tool chooses to apply the same method. The DELETE method attempts to remove a relation by first removing all references to it. The re-application of DELETE to the new specification produces an interesting result: because we have removed the only reference to the relation in the new specification, the relation definition can be removed without further ado. In practice this means that all of the problem solving structure below the user's goal in the original development (22 steps!) can

be eliminated in the new development. Imagine the effort involved in removing these steps given no problem solving structure (i.e., 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. Our experience has been that 1) pulling a single transformation application out of a long sequence, and then 2) attempting to describe its role in the sequence is a very difficult task.

In our hand analysis, the remainder of the router development replay ran just as smoothly. Overall, more than a third of the original development was trivially deleted, while a majority of the remaining steps were replayed verbatim. It is clear that radical changes to a specification will require more sophisticated problem solving to reuse the old development. However, a starting point for any such effort is a detailed, rational development history such as produced by Glitter.

7.2. Applying Glitter to specification construction

Our second extension involves using the Glitter model itself, as opposed to its output, on another software development process: specification. The basis of our work on Glitter has been that software development can be viewed as a problem solving task, with goals, methods and selection rules. Glitter supplies a representation for each. We conjecture that software specification can also be viewed as problem solving task. Hence, the Glitter model should be useful in this new domain. Our current research focuses on verifying this claim, i.e., applying the Glitter model to the software specification process [32,33]. We briefly describe the new model in the following sections.

7.2.1. Initial input

Glitter expects a complete, formal specification as input to the development process. What can we expect as input to the specification process?. Most methodologies would answer a blank piece of paper. However, recent work by Rich and Waters [34] and Neighbors [19] show that this does not have to be the case. In particular, we can attempt to catalog previous specification efforts for use in new but similar problems. This is the approach we are taking in our specification assistant. The specification writer peruses a catalog of abstract specification schemas, and composes a skeleton specification out of various domain-specific components. The final step is the tailoring, interfacing, and smoothing of components into a "concrete" specification.

7.2.2. Goals

Our original goals were specific to a transformational style of development. Our new goals must focus on tailoring abstract components into the desired specification. Goldman [35], among others, has shown that the specification process can be at least paritally described by a set of general, domain-independent, problem solving tasks. Such tasks will likely include *refine, generalize, specialize, constrain, instantiate, describe, add_domain_object*, among others. These tasks will be the initial goals of the system.

7.2.3. Methods (and transformations)

As with software development, there are two concerns in building a software specification,: 1) representing the problem solving or planning techniques and 2) representing the specification modification techniques. In the development domain, the task was to find a sequence of *correctness preserving* transformations that would move us from specification to implementation. Glitter's techniques centered on automatically generating such a sequence. In the specification domain, we are concerned with achieving specification goals through the use of problem solving methods and specification transformations. However, we are no longer concerned with producing a sequence of *correctness preserving* transformations, but instead a sequence of consistency preserving transformations. That is, we expect non-correctness-preserving changes when building a specification. We need to guarantee that these changes remain consistent with the rest of the specification. A small example might be useful here.

Suppose that the current (incomplete) state of an elevator problem specification included the following part:

```
type floor with attributes waiting|integer
type elevator with attributes num_occupants|integer, location|floor;

demon enter (e|elevator, f|floor)
  trigger: e:location = f
  response: e:num_occupants ← e:num_occupants + f:waiting

...
```

Assume that the above view of an elevator is too abstract for the problem. For instance, the notion of doors must be added. The user of the specification assistant might do the following[18]:

>Refine elevator to include the attribute door with values {open,closed}

To achieve the Refine goal, we want a method that will 1) add an attribute to an object, 2) make sure that the attribute value's type is well defined, and 3) find all references to the object and make any changes necessary to accommodate the new attribute. In our elevator example, this means 1) adding the attribute door to elevator, a simple specification transformation, 2) noting that by describing the explicit set of all possible values is a valid means of typing an attribute value (if the value type was undefined, we would want the method to jitter to define it), and 3) finding where an elevator object is referenced, and checking to see if the addition of doors has any effect. Among others, we would expect the system to find the three references to elevator in the enter demon, and post goals to Refine each to meet the newly refined view of elevator. Only one of the three refinement sub-goals is difficult to achieve, the one associated with the demon trigger:

[18] We use a stylized interaction. The actual interface relies on graphics and user fill-in to describe a particular goal.

Refine *trigger:* e:location = f given e:door = {open,closed}

Is it possible for a built-in method to achieve this goal? Not unless we allow domain specific methods. Such a method might read

> If you are dealing with a container-type object that has a location and a door, then entrance and exit should depend on both the location and the state of the door.

We have not attempted to build this type of domain specific knowledge into the system as of yet. In our current system, the user would be responsible for supplying the step that refined the trigger to include the door's state:

trigger: e:location = f *and* e:door = open

7.2.4. Selection rules

In the development domain, selection rules encapsulated knowledge about the computational efficiency of various implementation choices. In the specification domain, our concern is not with efficiency, but with clarity of the description. Suppose, for example, that the user wished to further constrain the non-determinism of a specification. Two alternative methods may be used: add the constraint directly using the global constraint construct; add the constraint implicitly by limiting non-determinism to specific choices. In our elevator world,

demon move (e|elevator, f|floor)
  trigger: e:location = f *and* e:door = closed
  response: e:location = (*a* floor)

...

To constrain movement to either the next floor up or below, we could choose to add to the above state a new constraint:

```
demon move (e|elevator, f|floor)
  trigger: e:location = f and e:door = closed
  response: e:location = (a floor)

constraint restrict_movement_to_one_floor (e|elevator, f1|floor, f2|floor, S|state)
  always  e:location = f1 in S and
          e:location = f2 in next(S) and
          adjacent(f1,f2)
  ...
```

Alternatively, we could place the constraint within the demon:

```
demon move (e|elevator, f|floor)
  trigger: e:location = f and e:door = closed
  response: e:location = (a floor || adjacent(f,floor))
  ...
```

We would expect our rules to note the strengths and weaknesses of each, e.g., the latter is clearer, the former is more general.

In summary, the construction of a knowledge-based system for helping a user build a software specification depends on the solution of a number of problems. To tackle these problems requires a repository of expert knowledge. We have found the Glitter framework defined in our work on software development to be useful in representing the knowledge in this new domain.

Acknowledgments

Members of the ISI Transformational Implementation Group -- Bob Balzer, Martin Feather, Neil Goldman, Jack Mostow, and Dave Wile -- had a great influence on this work. A further thanks to Martin, Jack and Dave for reading early drafts of this paper. I'd also like to thank Phil London and Lee Erman for their help on the Hearsay III implementation of Glitter. The construction of the Glitter system was supported by National Science Foundation grant MCS-7918792. Current work on applying Glitter to specification construction is supported by National Science Foundation grant DCR-8312578.

References

(1) Fickas, S., Automating the Transformational Development of Software, PhD Thesis, ICS Dept, University of California Irvine, 1982

(2) Partsch, H., Steinbruggen, R., Program Transformation Systems, *Computing Surveys*, **15**(3), (1983)

(3) Darlington, J., An experimental program transformation and synthesis system, *Artificial Intelligence*, **16**/(1), (1981)

(4) Balzer, R., Goldman, N., and Wile, D., On the Transformational Implementation approach to programming, *Second International Conference on Software Engineering*, 1976

(5) Cheatham, T., Holloway, G., Townley, J., Program Refinement by Transformation, *Proc. 5th International Conference on Software Engineering*, 1981

(6) Wile, D., Program Developments: Formal Explanations of Implementations *CACM* **26**(11), 1983

(7) Feather, M., A system for assisting program transformation, *ACM Trans. Program. Lang. Syst.*, **4**(1) (1982)

(8) Balzer, R., Transformational implementation: an example, *IEEE Trans. Softw. Eng.*, **7**(1) (1981)

(9) London, P., Feather, M., Implementing specification freedoms, *Science of Computer Programming*, Number 2, 1982

(10) Balzer, R., Goldman, N., Wile, D., Operational specifications as the basis for rapid prototyping, *SIGSOFT Softw. Eng. Notes*, **7**(5) (1982)

(11) Swartout, W., Balzer, R., On the inevitable intertwining of specification and implementation, *CACM* **25**(7) (1982)

(12) Paige, R., Koenig, S., Finite differencing of computable expressions, *ACM Trans. Program. Lang. Syst.*, **4**(3) (1982)

(13) Tappel S., Some algorithm design methods, *Proc. 1st NCAI*, 1980

(14) Bentley, J., Writing Efficient Code, Tech Report CMU-CS-81-116, Carnegie-Mellon University, 1981

(15) Standish, T., Harriman, D., Kibler, D., Neighbors, J., The Irvine Program Transformation Catalogue, ICS Dept., UC Irvine, 1976

(16) Manna, Z., Waldinger, R., A deductive approach to program synthesis, *ACM Trans. Program. Lang. Syst.*, **2**(1) (1980)

(17) Barstow, D., The roles of knowledge and deduction in program synthesis, *Proc. 6th IJCAI*, 1979

(18) Fickas, S. Automatic Goal-Directed Program Transformation, *Proc. 1st NCAI*, 1980

(19) Neighbors, J., Software construction using components, PhD. Thesis, ICS Dept., UC Irvine, 1980

(20) Mark, W., Rule-based inference in large knowledge bases, *Proc. 1st NCAI*, 1980

(21) Mostow, D.J., Mechanical Transformation of Task Heuristics into Operational Procedures, PhD. Thesis, CMU 1981

(22) Green, Cordell, Phillips, Jorge, Westfold, Stephen, Pressburger, Tom, Kedzierski, Beverly, Angebranndt, Susan, Mont-Reynaud, Bernard, Tappel, Steve, Research on Knowledge-Based Programming and Algorithm Design - 1981, KES.U.81.2 ,Kestrel Institute, Kestrel Institute, 1801 Page Mill Road, Palo Alto, Ca. 94304

(23) McDermott, D., Planning and acting, *Cognitive Science*, 2(2), 1978

(24) Kibler, D., Power, efficiency, and correctness of transformation systems, PhD. Thesis, ICS Dept., UC Irvine, 1978

(25) Terry, A. Hierarchical control of production systems, PhD. Thesis, ICS Dept. UC Irvine, 1982

(26) Chiu, W., Structure Comparison and Semantic Interpretation of Differences, *Proc. 1st NCAI*, 1980

(27) Randall, D., Meta-Rules: Reasoning about control, *Artificial Intelligence*, Pages 179-222

(28) Wilensky, R., Meta-planning, *Proc. 1st NCAI*, 1980

(29) Barstow, D., Knowledge-based program construction, Elsevier North-Holland, 1979

(30) Erman, L., London, P., and Fickas, S., The design and an example use of Hearsay-III, *Proc. 7th IJCAI*, Vancouver, BC, 1981

(31) Kant, Elaine, Efficiency Considerations in Program Synthesis: A Knowledge-Based Approach, PhD. Thesis, Stanford, 1979

(32) Fickas, S., Mechanizing software specification: a proposal, Technical report, CS Dept, U of Oregon, 1984

(33) Fickas, S., Mechanizing software specification, *Proc. Workshop on Models and Languages for Software Specification*, Orlando, 1984

(34) Rich, C., Waters, R., Formalizing reusable software components, *Proc. Workshop on Reusability in Programming,*, 1983

(35) Goldman, N., Three dimensions of design development, *Proc. 3rd NCAI*, 1983