CIS-TR 85-01

Automating the Transformational Development of Software

Stephen Fickas

Department of Computer and Information Science
University of Oregon

This report will appear in IEEE Trans. Soft. Eng. 1985.

Automating the Transformational Development of Software

by

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

January 1985

ABSTRACT

This paper reports on efforts to extend the Transformational Implementation (TI) model of Software Development [1]. In particular, we describe a system that uses AI techniques to automate major portions of a transformational implementation. The work has focused on the formalization of the goals, strategies, selection rationale, and finally the transformations used by expert human developers. A system has been constructed that includes representations for each of these problem solving components, as well as machinery for handling human/system interaction and problem solving control. We will present the system and illustrate automation issues through two annotated examples.

1. Introduction

In a previous issue of this journal, Balzer presented the Transformational Implementation (TI) model of software development [1]. Since that article appeared, several research efforts have been undertaken to make TI a more useful tool. This paper reports on one such effort.

A general model of software transformation can be summarized as follows[1]: 1) we start with a formal specification P (how we arrive at such a specification is a separate research topic), 2) an agent S applies a transformation T to P to produce a new P, 3) step 2 is repeated until a version of P is produced that meets implementation conditions (e.g., it is compilable, it is efficient). Figure 1 presents a diagram of the model.

[1]A survey and more detailed discussion of transformation systems can be found in [15].

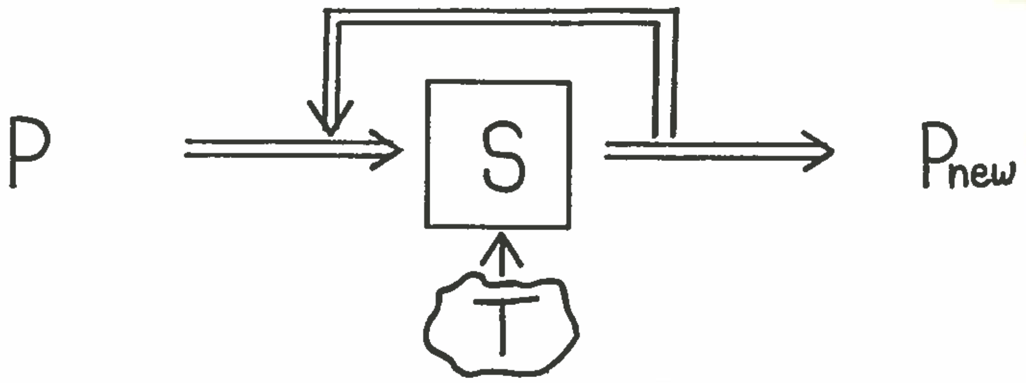

Figure 1

In the TI model, the specification language P is Gist [12], the agent S is an expert human developer, and T is taken from a catalog of correctness-preserving transformations[2] (Section 2 discusses other bindings of P, S, and T). Hence, the human is responsible for deciding what should be transformed next, and what transformation to use; the system checks the transformation's pre-conditions and applies it to produce a new state. As Balzer noted in [1], the TI model provides at least two advantages:

(1) Focus is shifted away from consistency problems and towards design tradeoffs.

(2) The *process* of developing a program is formalized as a set of transformations. Thus the process itself can be viewed as an object of study.

Since Balzer's article appeared, we have attempted to use the TI model on several realistic problems. This work has confirmed one of the article's conjectures:

> "... it is evident that the developer has not been freed to consider design tradeoffs. Instead of a concern for maintaining consistency, the equally consuming task of directing the low level development has been imposed. While the correctness of the program is no longer an issue, keeping track of where one is in a development, and how to accomplish each step in all its fine detail diverts attention from tradeoff questions. It is quite clear that if transformation systems are to become useful, this difficulty must be removed."

One solution is to automate portions of a TI development[3]. That is, find a way for the machine to automatically find and apply a portion of the transformations. This paper reports on our efforts to bring about this solution. In particular, we discuss the Glitter system, which uses problem solving techniques to generate automatically many of the steps of a development. Glitter is a working system implemented in the expert system writing language Hearsay III [5]. We will present Glitter through an example in section 4. Section 5 contains another, more lengthy example of a Glitter development.

[2]Correctness rests both on Gist's formal semantics and on pre-conditions attached to transformations. Proofs of correctness are carried out by inspection; there has been no attempt to date to apply a more formal proof method.

[3]Here, and throughout the paper, we will use *TI development* to mean the mapping of a specification into an implementation using transformations.

2. A Closer Look at the Problem

Glitter traces its roots to prior efforts in the area of transformational development. To produce a working transformational system, several portions of the abstract model in figure 1 must be further specified. First, the form of the initial P determines both the level of specification, and the form the transformations will take. If our goal is to specify only *what* the problem is, as opposed to *how* to solve it, we want the initial P to be a high level specification. On the other hand, if we are interested in applying efficiency tricks in a production language like Pascal, our initial P may be low level, e.g., Pascal itself. The variety of specification levels possible is shown by a set of illustrative transformation systems:

- The TI model uses a high level specification language called Gist as the initial starting point. The designers of Gist have set as a goal freeing the specification writer from any implementation concerns. Thus both data and control representations are defined to allow, through skillful and careful construction, implementation-independent specifications. The form of transformations range from ones that deal with the most abstract design decisions, to ones that involve the nitty-gritty optimization of operations and data structures.
- The PSI system (in particular the PECOS/LIBRA components [3]) starts with an abstract algorithm defined as a *Program Model* [13]. In a program model, data remains at an abstract level, while control is more concrete. The form of transformations (called refinements) in PECOS [2] ranges from intermediate to low level design decisions.
- The Irvine Transformational System [16] starts with a program written in a Pascal-like language. Transformations are source-to-source, and tend to involve optimization of expression evaluation, although some control optimizations are included.

There are conflicting concerns in choosing the initial level of specification. The lower we start, the better are the chances of building a complete set of transformations: our software design knowledge at the lower levels is better studied. However, the higher we start, the better are the chances of bringing about real efficiency gains: it is often the high level design decisions that most affect the final space and time taken by the implementation. Unfortunately, at the higher levels our design knowledge is less formal and not well charted. The result of this conflict is that systems that attempt to use a completely automatic model of transformation selection and application have necessarily been forced to use an initial specification that is either below the desired level, or at the right level but not expressive enough to handle realistic problems. More recent work has attempted to handle high level specifications by involving the user in the transformation process. This is the first key idea on which Glitter builds: an *interactive* system stands a chance of transforming realistic, high level specifications into an efficient implementation.

This brings us to our second decision point, namely the constitution of the agent S. In completely automatic systems such as PSI, S is a computer-based component that decides 1) what portion of P to work on, 2) what set of transformations are applicable, and 3) which transformation to apply from the set. Conversely, in the TI model, S is a human. The human is responsible for all of the above items except transformation application, which is handled by the system. Other systems such as ZAP [6], CHI [10], Draco [14], and Glitter attempt to strike a better balance by automating more extensive portions of transformation selection and application, and leaving smaller portions to the human. The second key idea used in Glitter is then that interactive transformation systems can still have a significant degree of automation.

Finally, there are various ways to control the search for the "right" set of transformations. The PSI system uses a modified best-first search. Systems such as PDS [4] and the Irvine Transformation System employ a formal production system approach which applies all transformations until none are applicable (this type of control is often found in rewrite systems where Church-Rosser properties hold). The PADDLE system [17] allows backtracking to a decision point. In particular, PADDLE records the set of transformations used, and allows the user to *replay* or rerun them to an arbitrary point. Hence, if the user wishes to try an alternative to the kth transformation applied, he or she may request PADDLE to replay k-1 transformations. Thus PADDLE treats a sequence of transformation selections and applications as a program. Parts or all of it can be deleted, modified or executed. This leads to the third key idea used by Glitter: the *process* of selecting and applying transformations can be viewed as a *product* as well. Glitter declaratively stores all of the information it is able to capture during the transformation process. This can be used to aid in maintenance.

To summarize, Glitter is built on three key ideas:

(1) An expert human user must be included in the transformation process if we wish to study realistic problems.

(2) An interactive model can still include useful automation.

(3) Given that a transformational development is no more static than a normal software development, we will need to make changes to it. Hence, the process itself becomes an object of study.

The focus of our work is on the second idea. We set out to see how far we could push automation in the TI model. In doing so, we found ourselves concerned with the third idea as well. Specifically, once we began to study the transformation process, we noticed a curious thing: only a small part of the process was being captured by the machine. In particular, the selection and application of a program transformation was only the tip of the problem solving iceberg. Watching expert human developers, it became clear that the sophisticated planning necessary to generate one or more transformations would also have to be represented. Our goals became first to formalize the problem solving steps used to generate transformation applications, and then to build a problem solving system around them. The formalization would include the goals, strategies, selection rationale, and finally the transformations used by expert human developers. The system would include representations for the formal types of knowledge, and machinery for handling human/system interaction and problem solving control. In the remainder of the paper, we discuss the degree to which we were able to meet our goals.

3. A Gist Specification of the Package Router Problem

Our intent is to introduce the Glitter model by showing it in action. To do so, we will need to provide some background material on Gist and the example problem we will use, that of a postal package router.

A transformational development using Glitter, starts with a specification written in the Gist language. The specification is described as a closed-world system, i.e., both the behavior of the process to be implemented, and the environment in which it resides are described. All constraints on the process, including those placed by the environment, are made explicit. As an example, consider a postal routing process for distributing packages into destination bins. Packages arrive at a source station, and then slide through a network of chutes and switches into bins. Figure 2 depicts the network.

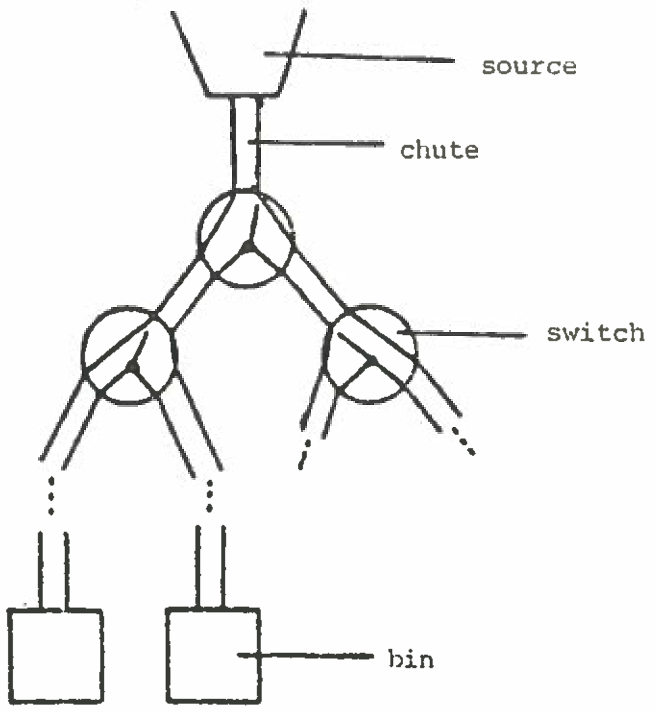

Figure 2

The English description of the problem 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*. 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); switches and chutes have no sensors for determining package destination. However, the presence of a package in a chute, switch or bin can be sensed. The problem is defining a switch controller that will minimize package misrouting.

A Gist specification of the routing process must take several things into consideration. As part of the process, the specification should state that the switches behave in a way that provides correct routing whenever possible, i.e., the final implementation must move the switches to the proper setting at the right time. Because the switches are under program control, they are part of the process specification.

In contrast, the movement of packages through the network is by gravity feed, and is not controllable by the process. However, since package movement is part of the routing environment, its behavior too must be specified. In general, a transformational development requires implementing the controllable portions of the specification so they act in acceptable ways in the uncontrollable environment.

When specifying a process and its environment, the goal is to make a clear and correct statement of behavior, *without* having to provide an algorithm for affecting that behavior. Gist provides certain specification freedoms that allow a specification writer to ignore implementation concerns. In the following sections, we will show how these freedoms can be used to specify portions of the package router.

3.1. The objects

Gist provides a relational model of information, i.e., typed objects and relations between them. For instance, we can use the following to specify objects and relations in the postal package router problem:

```
type package;
type location has subtypes (source, chute, switch, bin);

relation LOCATED_AT(package, location)
relation DESTINATION(package, bin)
relation SWITCH_SETTING(switch, chute)
```

Information may be retrieved via predicates and expressions. Note that the special symbol * can be used to return a particular relation argument as shown in the second example below[4]. Also, relational attributes of objects can be retrieved by the syntactic shorthand "object:attribute", as shown in the third example.

A package in the domain:	*a* package
The location of package p:	LOCATED_AT(p, *)
The location of package p:	p:LOCATED_AT
Is package p at its destination?	p:LOCATED_AT = p:DESTINATION

It is during development that concerns about data access paths, statistical distribution of operations, size, etc., are taken into account to select a physical data representation.

A Gist specification allows information to be extracted from a past state without concern for how it might be made available in the current state.

Has this package ever been at that switch?	package:LOCATED_AT = switch *as of ever*

[4]Data Base retrieval in Gist is non-deterministic (see 3.3 for an example). Hence the * does a non-deterministic match. In this case, the package router specification, in a section we have omitted in this paper, constrains a package to be at a single location at any one time, i.e., the * matches uniquely.

Gist allows information to be specified as an invariant on other data. For instance, we might define the predicate relation EMPTY_SWITCH on the LOCATED_AT relation[5]:

Define a relation that holds when a switch is empty.

```
relation EMPTY_SWITCH(switch)
         definition not E package || package:LOCATED_AT = switch
```

By stating this invariant globally, we make it available throughout the specification. Further, no maintenance of the invariant is required; it is during development that code will be introduced to either maintain it explicitly, or rederive it when necessary.

3.2. The Actions

Activity in a domain is modeled by the creation and destruction of objects, and the insertion and deletion of relations among objects. A change to the domain causes a transition to a new state. A Gist specification denotes one or more sequences of states and transitions. Each such sequence is called a behavior. A development is the implementation of a particular behavior.

Create a new package:	*create* package
Route package p to bin b:	*insert* DESTINATION(b, p)
Update the location of p to loc2:	*update* p:LOCATED_AT *to* loc2

Demons are Gist's means of providing data-triggered activity. Demons allow the specification of asynchronous actions that trigger on a state change. A demon includes a trigger and a response. The former consists of a predicate that recognizes state transitions. The latter is executed when the trigger predicate becomes true.

Note the arrival of a package at a bin:

```
demon note_arrival(package, bin)
   trigger: package:LOCATED_AT = bin
   response: update bin:CONTENTS to +1
end-demon
```

3.3. The constraints

Gist allows a non-deterministic choice to be made from among a set of objects or actions.

[5] We will use E to stand for the existential quantifier.

Any package at a switch: *a* package || package:LOCATED_AT = *a* switch

Set the switch s (non-deterministically) to an outlet:

insert s:SWITCH_SETTING ← s:SWITCH_OUTLET

Non-determinism may also be introduced to model portions of the environment. In the example below, the non-determinism reflects the choice of either triggering or not triggering the demon on each state transition.

Create packages at random times:

```
demon create_package
  trigger: Random
  response: create package
end-demon
```

Gist constraints can be used to set limitations on the environment, or prune the set of non-deterministic choices. An example of the former is a constraint on the topology of the network:

```
constraint UNIQUE-ENTRY
  always required: forall bin || LOCATION_ON_ROUTE_TO_BIN(source, bin)
end-constraint
```

An example of a constraint on a non-deterministic choice point will be discussed in detail in the next section.

This section has given a brief overview of Gist syntax and semantics. The interested reader should consult [12] for further details.

4. An Introduction to the Glitter Model

In this section we will introduce the major components of Glitter by looking at a portion of the transformational implementation of the Package Router specification. To review briefly, the objects in the router domain are packages, chutes, switches, and bins. The operations are putting a package into the input chute, moving it from one location to the next, and changing the setting of a switch. Perhaps the heart of the Gist specification of the package router is the constraint MUST_SET_SWITCH_WHEN_HAVE_CHANCE, paraphrased below:

For every switch s, the following is *always prohibited*:

1) a package p is in s, and
2) s is set so that p will become misrouted, and
3) at some earlier time, s was empty and p was the next package due at s

In Gist, this can be represented as

```
constraint MUST_SET_SWITCH_WHEN_HAVE_CHANCE
  always prohibit
      exists p|package, s|switch ||
            p:LOCATED_AT = s and
            SWITCH_SET_WRONG_FOR_PACKAGE(s, p) and
            (   (p = first PACKAGES_DUE_AT_SWITCH(*, s)
                  and
                 SWITCH_IS_EMPTY(s))
              as of everbefore)
end-constraint
```

The body of MUST_SET_SWITCH_WHEN_HAVE_CHANCE contains references to the relations LOCATED_AT, SWITCH_SET_WRONG_FOR_PACKAGE, PACKAGES_DUE_AT_SWITCH, and SWITCH_IS_EMPTY; each is defined elsewhere in the router specification. Our interest here will be with the derived relation PACKAGES_DUE_AT_SWITCH.[6] As we can see, the constraint uses the relation to reference the first of the sequence of packages due at a switch. The definition of this sequence is given below:

```
Specification PACKAGE-ROUTER
  ...

  constraint MUST_SET_SWITCH_WHEN_HAVE_CHANCE ... end-constraint

  relation PACKAGES_DUE_AT_SWITCH(packages|sequence of package, switch)
       definition packages =
          {p|package ||
             LOCATION_ON_ROUTE_TO_BIN(switch, p:destination) and
             not (p:LOCATED_AT = switch as of everbefore) and
             not MISROUTED(p)}
        ordered by *:LOCATED_AT = source
  ...

end-specification
```

Abstractly, the sequence is defined as a set *ordered by* an event. An element of the package set must have the following characteristics:

[6] In Volume II of [7], the complete development of the package router is given including the constraint above. Because the development of the constraint is both lengthy and complex, we have chosen the simpler example of PACKAGES_DUE_AT_SWITCH, one which we believe still adequately illustrates our points.

1) for p to get to its destination, it must go through the switch
2) p has never been in the switch
3) p is not currently misrouted

The ordering on the set is by each package's entrance to the router, i.e., p1 is ordered before p2 if p1 entered before p2. We will give away the punchline early by saying that the final implementation of the sequence will be a linked list with functions for adding and deleting packages. That is, after applying appropriate transformations to the specification, we will replace the definition of PACKAGES_DUE_AT_SWITCH above with an array of linked lists (one for each switch), and two lisp functions, add-package and delete-package, for maintaining the lists.

Unfortunately, to get to this rather straightforward implementation, we must replace all derived relations in the definition of PACKAGES_DUE_AT_SWITCH with new explicit ones, we must do a fair amount of code movement and hard reasoning to insure that add-package and delete-package end up in places we have control over, and finally we must do the significant amount of subgoaling and simplification that always attends a transformational development. This results in over 30 program transformations being applied. Figure 3 shows a portion of the transformation sequence, in stylized form.

T1: Define new relation ROUTER_NETWORK*(location, location)

T2: Define code to 1) compute the transitive closure of the router network,
and 2) explicitly store each connection as a ROUTER_NETWORK* relation

T3: Replace each reference to LOCATION_ON_ROUTE_TO_BIN with reference
to ROUTER_NETWORK*

T4: Remove LOCATION_ON_ROUTE_TO_BIN from program

...

T10: Simplify expression (not not p => p)

T11: Simplify expression (p and p => p)

T12: Define new relation PACKAGE_LIST(packages, switch)

T13: Define maintenance code for PACKAGE_LIST: add-package, delete-package

T14: Insert maintenance code at point where package destination is created

T15: Verify package entrance to router is equivalent to package creation

T16: Verify no change to PACKAGES_DUE_AT_SWITCH between package creation and entrance

T17: Move maintenance code from creation code to entry code

...

T30: Define array of linked-lists, one for each switch

T31: Replace relation access with linked-list access

Figure 3

The major points to make about the above transformation sequence are that a) it is unstructured, and b) design decisions are left implicit. Hence, it is hard to generate, understand, and modify. What is clearly missing are the higher level goals and strategies that organize steps into coherent wholes, and the criteria used to choose one transformation over other viable candidates. For example, T1 through T4 are all focused on removing the LOCATION_ON_ROUTE_TO_BIN relation. This in turn is part of a bigger goal of removing all derived relations from the definition of PACKAGES_DUE_AT_SWITCH. This in turn is part of the goal of implementing PACKAGES_DUE_AT_SWITCH, which is finally part of implementing the package router specification. The same type of goal structure can be drawn for each of the other 27 steps above.

Glitter attempts to formalize the goals, strategies and design decisions left implicit in transformational implementations. When successful, it gains two advantages: 1) it allows automation to proceed to a much greater extent than working at the primitive transformation level, and 2) it documents much more of the transformational process, namely the planning space. To see this, we will now turn to the development of PACKAGES_DUE_AT_SWITCH using Glitter. The user starts by telling Glitter that he or she wishes to work on the relation PACKAGES_DUE_AT_SWITCH:

User: Develop PACKAGES_DUE_AT_SWITCH

Glitter encorporates a language for stating transformational goals. The syntax of the language is simply the goal keyword (e.g., Develop) and a set of typed arguments (e.g., a Gist construct). The language has evolved in two ways: 1) top down by studying the ways expert transformationalists develop programs, and 2) bottom up by studying existing transformational developments, and attempting to infer the goal structure (see also [7]). It is important to note that a Glitter goal is a representation of a goal to be achieved *and not* a function call. A goal derives its semantics from an attached *achievement condition*, a piece of lisp code that monitors for its successful achievement. Thus, the language permits goals to be stated that may be impossible to achieve in certain situations. This is not viewed as an error, but simply as a dead-end solution path.

In our example, the user has posted a Develop goal with an argument of the relation PACKAGES_DUE_AT_SWITCH. The achievement condition for Develop will monitor the relation PACKAGES_DUE_AT_SWITCH, and signal when it has been successfully implemented. We might now ask how an expert would tackle the development of PACKAGES_DUE_AT_SWITCH. There are two general strategies he or she might employ: 1) explicitly store the sequence for each switch, and define code to maintain each sequence as packages move around, or 2) try to generate the sequence on demand from available information. In Glitter, we have developed a framework for representing and cataloging such strategies. This framework is built around what we call a *method*. A method has a goal slot, a filter slot, and an action slot. We fill the goal slot with the goal we wish to achieve (e.g., Develop X), the filter slot with predicates that will check for situations when the method is not appropriate, and the action slot with one or more operations useful for achieving the goal. In this case, we need two methods, one called MAINTAIN and one called REDERIVE. Each will have their goal slot filled with a pattern that matches the type of goal they can achieve, i.e.,

Goal: Develop DR|derived-relation

Each action slot will call for a new goal to be posted:

```
Method MAINTAIN
    Goal: Develop DR|derived-relation
    Action-1: Maintain DR
End Method

Method REDERIVE
    Goal: Develop DR|derived-relation
    Action-1: Rederive DR
End Method
```

In Glitter, the posting of a goal causes all methods to check their goal slots for a match. All such matches are collected together in a *candidate set*. This is the first bit of automation provided by Glitter: it does an indexed retrieval of the available strategies and tactics. The shift here is away from the user choosing *how* to transform something, and towards the user stating *what* he or she wants done.

Given the two competing strategies, what might we expect an expert to do next? There are certain criteria for deciding if either is even possible, and which will lead to the better solution. Among these criteria are a) is there enough information around to rederive, b) is the information too extensive to store explicitly, c) how often is the information updated, d) how often is it retrieved? The answers to these questions will lead the expert to choose one strategy over the other. In Glitter, we provide a framework for representing and cataloging this type of selection information. The framework is based on what we call a *selection rule*. The basic form is that of an IF-THEN rule where the antecedent portion checks to see if a certain feature is present in the current state, and the consequence portion adds a positive or negative weight to one or more methods in the candidate set. After all selection rules have had a chance to add their votes, the system selects the method with the best overall score. Kant provides a similar mechanism in her Libra system [11].

The evaluation of the left hand side of a selection rule is often beyond the means of the system. In these cases, the user is asked to supply information. For example, one selection rule that is appropriate for eliminating MAINTAIN as a candidate is as follows:

```
SelectionRule TooBig?
  IF the method MAINTAIN is a viable candidate, and
       DR represents a (practically) infinite set
  THEN eliminate MAINTAIN as a viable candidate
End SelectionRule
```

Currently the system can evaluate the second clause above in only limited cases. In general, the system will ask the user to supply a truth value to any clause of a selection rule that it cannot evaluate. Thus we will get

Glitter (SelectionRule question): PACKAGES_DUE_AT_SWITCH represents a (practically) infinite set?

User: no

This dialog will continue as other selection rules are tried. The final outcome, given the information supplied by the user, will be the selection of MAINTAIN as the method to apply.

We must stop here to make two points. First, the user is often asked some very tough questions during evaluation of selection rules. One might complain that this is too much effort to expect from the user. However, the questions asked are not any different than the questions that must be answered implicitly in the TI model. The only difference here is that we explicitly represent them and document their answers.

Second, there is a degree of automation and organization that is added. For instance, some mundane types of analysis are currently carried out by the system, e.g., counting the number of references to a relation. Questions are "indexed" so that they are asked at the right moment, and only then if they will shed some discriminatory light on selection (i.e., while the method is still a viable candidate). Of course even with all of the above, it still may be too much of a burden on the user. This is more of a general statement on the transformational paradigm and the state-or-the-art in theorem proving. Our argument is that as more powerful analysis routines are added to the system, they will find a waiting home in the selection rule framework, and their effect will be immediately felt by the user by increased automation.

Moving back to our example, assume that the MAINTAIN method is chosen. Glitter records this as a choice point. It records the entire candidate set, the selection rules that applied, and any information supplied by the user. Later, if the MAINTAIN method proves to be a bad choice, the user may move back to this point, examine the other candidate methods (i.e., REDERIVE), ask to see the record of the selection process, and choose another method (spawn an alternative solution path) if necessary. To allow this, Glitter retains two types of information ([5] describes how both types of information are represented in the Hearsay III implementation of Glitter). First, a record is made of the design-decision tree. Figure 4 shows the tree generated from the 51 steps necessary to develop PACKAGES_DUE_AT_SWITCH (we are currently only at step 1). It is unary tree because no alternative decisions have yet been explored. The dotted arc and node represent the later selection of REDERIVE as the method to achieve our initial Develop goal.

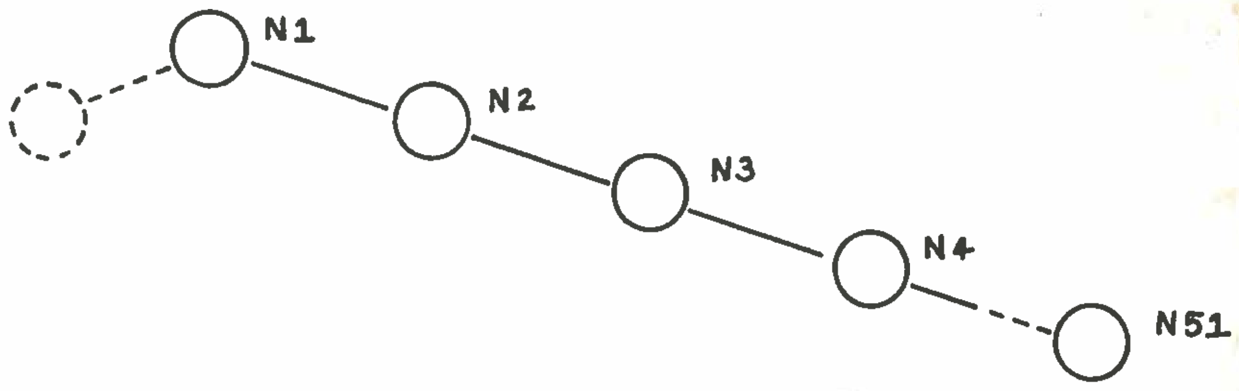

Figure 4

The second type of information the system records is the current state of problem solving within each node in figure 4. To show this, nodes N1 and N2 are blown up in figure 5.

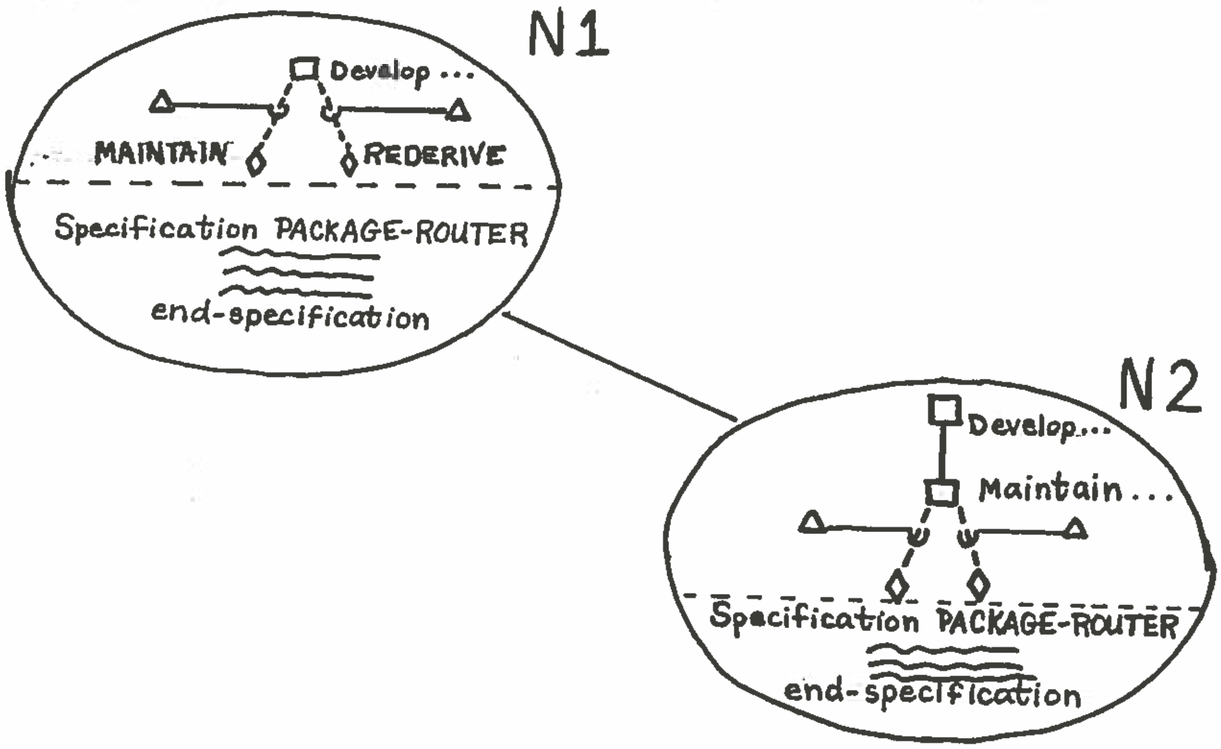

Figure 5

In node N1, the goal Develop (shown as a square) has two candidate methods MAINTAIN and REDERIVE (shown as diamonds). Two selection rules (shown as triangles) have weighted the two methods. Also contained in the node is the entire state of the package router program in Gist.

Node N2 in figure 5 represents the outcome of selecting MAINTAIN as the method to apply. The selection of the method causes a new goal to be posted:

Glitter: Maintain PACKAGES_DUE_AT_SWITCH

In N2, this is represented as a subgoal under the Develop goal, which in turn has two candidate methods with their own selection rules. Since the transition from N1 to N2 was brought about by a planning as opposed to transformation step, the program state in N2 is exactly that of N1. In fact, all development action to this point has been carried out in the planning space; no changes have been made to the Gist code. We must go through 3 more levels of subgoals before we reach the first transformation step T1 in figure 3.

Having introduced the system, we will end this example here, and move to a development problem that better illustrates the low level steps found in a TI development. Before doing so, we summarize the automation found in the Glitter development of PACKAGES_DUE_AT_SWITCH. The final number of non-transformation (i.e., planning) steps used to generate the 31 transformations in figure 3 (and achieve the Develop goal) was 20. Out of the total 51 steps, 45 were produced by the system. Included in the 45 were all of the 31 transformations.

5. Another Example

The last section followed the first few steps of a development to show the major components of Glitter. In this section we will look at an example that better illustrates the many low level steps found in a TI development. Automating these steps is one of our prime concerns.

To carry the example to the necessary depth in a reasonable number of pages, we will show explicitly only the goals; we will paraphrase the methods and selection rules used. We will also omit the selection process when it is uninteresting.

The example deals with one of the data structures of the package router specification: the sequence of packages that have been at the source, i.e., have entered the package router. This sequence is represented by the relation ALL_PACKAGES. There is only one reference in the specification to ALL_PACKAGES, that found in the demon that takes care of packages entering the router, i.e., RELEASE-PACKAGE-INTO-NETWORK. Among other things, the demon will hold up a package if its destination is not the same as the last package to enter. This is to prevent bunching.

```
demon RELEASE-PACKAGE-INTO-NETWORK(newp|package)
  trigger: newp:LOCATED_AT = the source
  response:
     if (previous|package ||
               previous immediately before newp  wrt ALL_PACKAGES):DESTINATION
                      neq newp:DESTINATION
       then invoke WAIT();
     ...
end-demon

relation ALL_PACKAGES(package_seq | sequence of package)
   definition package_seq) = ...
```

Suppose that the user turns his or her attention to optimizing the relation ALL_PACKAGES. A goal is posted to let Glitter know that ALL_PACKAGES is under scrutiny:

User: Optimize ALL_PACKAGES

One method for optimizing a sequence tries to replace the sequence with zero or more individual variables. This method is useful when only part of a sequence is needed. One of Glitter's selection rules determines that since there is only one reference to ALL_PACKAGES (something the system can determine automatically), and that reference uses a relative position (again, something the system can determine automatically), the likelihood of the method succeeding is high. Once chosen, the method sets out to replace the reference to ALL_PACKAGES in RELEASE-PACKAGE-INTO-NETWORK with a reference to a new variable (represented as a relation). The following subgoal is generated:

Glitter: Fold (previous|package || ...)

That is, replace the expression with a reference to a newly created *derived* relation. There is a single method for achieving this goal. Its first action is to get rid of all ties to the local context; we can't fold the expression into a global relation if we must have ties to local variables, e.g., newp. Hence, a new subgoal is posted:

Glitter: Globalize (previous|package || ...)

This goal is achieved when the expression contains no local references. One method that is indexed to this goal attempts to replace a reference to a local variable with a reference to a relation (others attempt to replace a variable with its value). Since all relations are global, this will achieve the Globalize goal. Choosing the method causes a new goal to be posted:

Glitter: Reformulate newp in (previous|package || ...) as <relation-reference-to $>

The achievement condition attached to the Reformulate goal is a pattern-matcher. Each time the Gist construct in its first argument changes, it checks to see if it matches the second argument; the $ will match any relation. In this case, as soon as the variable newp is transformed into any type of relation reference, the goal is marked as achieved. One of the methods that is a candidate for achieving this transformation embodies the following piece of knowledge:

> If you are trying to replace a variable reference with a reference to a relation, find a relation that defines a sequence of the same type as the variable. It may be the case that the object referenced by the variable is also part of the sequence.

One selection rule defined for this method checks to see if the variable reference is part of a larger expression involving a sequence. In this case it is, so the method is recommended and chosen. This leads to the goal

Glitter: Reformulate newp in (previous|package || ...) as <relation-reference-to ALL_PACKAGES>

The system knows about various ways to reference a sequence element, e.g., *first, last, nth*. Each of these methods will be a candidate here. The system must rely on the user to make a choice. Selection rules prompt the user as follows:

Glitter (selection rule question): can newp in (previous|package || ...)
be replaced with *last* ALL_PACKAGES(*)?

User: yes.

...

At this point the system applies a correctness-preserving program transformation to replace newp with *last* ALL_PACKAGES(*). We now have

```
demon RELEASE-PACKAGE-INTO-NETWORK)(newp|package)
  trigger: newp:LOCATED_AT = the source
  response:
     if (previous|package ||
                 previous immediately before  last ALL_PACKAGES(*)
                     wrt ALL_PACKAGES):DESTINATION
                                 neq newp:DESTINATION
        then invoke WAIT();
        ...
end-demon
```

We are now finished with one portion of the optimization plan for ALL_PACKAGES. To achieve the top goal, Optimize ALL_PACKAGES, will require another twenty-nine steps. The final outcome will be as follows:

```
relation LAST_PACKAGE(package);

demon RELEASE-PACKAGE-INTO-NETWORK(newp|package)
  trigger: newp:LOCATED_AT = the source
  response:
     if LAST_PACKAGE(*):DESTINATION neq newp:DESTINATION
        then invoke WAIT();

     update LAST_PACKAGE(*) to newp;
     ...
end-demon
```

We want to make three points about this example. First, it shows the automation leverage provided by Glitter. The user was responsible for initially posting the Optimize goal, and for answering questions about the correctness of replacing the variable newp with an element of ALL_PACKAGES. In the overall problem solving needed to achieve the Optimize goal, these were the only two times the user was involved in the entire thirty-five step process. Thus, we achieve a greater than ten-to-one automation lever. Further, the steps automated are just the right type, i.e., the mundane jittering steps so ubiquitous in a transformational development.

Second, the type of techniques needed to optimize ALL_PACKAGES represent much hard experience in building transformational developments, i.e., it is expert knowledge. Interning this knowledge in a catalog allows us to reuse it again and again. Hence, Glitter begins to codify an expert's knowledge about the transformational process. While cataloging program transformations is not new (see for instance [2]), cataloging the problem solving knowledge necessary to generate transformation steps is new. And as we have argued, this is exactly where the power lies in a transformational system such as TI.

Third, Glitter documents the problem solving process that went in to optimizing the sequence. We now know that newp was replaced with *last* ALL_PACKAGES(*) because that is a relation reference, and a relation is global, and a global reference is ok but a local one is not when we are trying to fold an expression into a relation, and that was what we were trying to do so that we could replace a reference to ALL_PACKAGES with a variable, which in turn will allow us to throw away the portions of the sequence we are not interested in, which finally is an optimization of the sequence. Is this information useful? When it comes to maintenance, we answer with an emphatic yes. In the TI model, changes to software are made at the specification level. The question is, given a specification change how much of the original development can be salvaged. Although we have just begun to explore this question in detail, it is obvious that the more we know about the problem solving steps, the easier it is to reuse the old development. Related to maintenance, there is the question of user exploration. Glitter keeps a record of all choice points reached during a development. This is an *active* record in that the user may move to a choice point and make an alternative decision (as shown in Figure 4). The alternate choice becomes part of the history just as the initial choice.

6. Summary and Further Work

Experience with the Transformational Implementation model described by Balzer [1] pointed out a major weakness: the model was under mechanized. It lacked a representation of goals, strategies and design decisions. It lacked automation. Our research goal was to remedy this problem by providing a new system that automatically found and applied a significant portion of the transformations in a TI development. Along the way, we found that we also had to come to grips with the formal representation of the problem solving components missing from TI.

There are several ways we can assess Glitter's success in achieving our goal. We kept statistics on the two largest examples attempted using Glitter: the package router and a small text editor. >From a conservative standpoint, the system averaged a one-to-three ratio between user goals and transformation applications. This ignores the intermediate problems solving steps generated by Glitter. If we take these into account, as we argue we should, the system provided better than a one-to-ten ratio between user steps and system steps. Further, this cannot help but get better. Many of the user steps are required because of missing system knowledge. As this knowledge is added, we expect reliance on the user to decrease.

While automation was our major concern, we can now see that the problem solving record -- the goals, strategies, design decisions -- generated by Glitter is also of importance. One of our current projects is to use this record to help in maintenance. In particular, we are attempting to classify the types of changes a maintainer might make to a specification, and use these as a guide in salvaging portions of the original development. We find that deciding whether a high level goal or method is still usable is a potentially solvable problem. Deciding whether an isolated transformation is still usable is, on the other hand, often intractable.

Finally, we noted in section 1 the assumption that a Gist specification would be provided to Glitter by some other process. We have begun to study that specification-construction process more formally. Specifically, the KATE project [9] is an attempt to reuse Glitter in the development of specifications. Goals, methods, selection rules, and transformations now are centered on specification design as opposed to program design. Among other things, we hope to capture and document the problem solving steps that lead to a specification such as the Package Router.

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.

The construction of the Glitter system was supported by National Science Foundation grant MCS-7918792. Current work on the Glitter system is supported by National Science Foundation grant DCR-8312578.

References

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

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

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

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

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

(6) Fickas, S. "Automatic Goal-Directed Program Transformation," in *Proc. 1st National Conference on AI*, 1980

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

(8) Fickas, S., Laursen, D., Laursen, J., "Knowledge-based Software Specification," presented at the Workshop on Knowledge Based Design, Rutgers Univ., 1984

(9) Green, C., Westfold, S. Knowledge-Based Programming Self Applied, Machine Intelligence 10, pp. 339-360

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

(11) Kant, E., Barstow, D., "The refinement paradigm: The interaction of coding and efficiency knowledge in program synthesis," *IEEE Trans. Softw. Eng.*, **7**(5) (1981)

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

(13) McCune, B., Building program models incrementally from informal descriptions, PhD. Thesis, Computer Science Dept., Stanford University (1979)

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

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

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

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