Automatic Goal-Directed Program Transformation

Stephen Fickas

USC/Information Sciences Institute*
Marina del Rey, CA 90291

1. INTRODUCTION

This paper focuses on a major problem faced by the user of a semi-automatic, transformation-based program-development system: management of low level details. I will argue that it is feasible to take some of this burden off of the user by automating portions of the development sequence. A prototype system is introduced which employs knowledge of the transformation domain in achieving a given program goal state. It is assumed that such a system will run in a real environment containing a large library of both generalized low level and specialized high level transformations.

2. THE TI APPROACH TO PROGRAM DEVELOPMENT

The research discussed here is part of a larger context of program development through Transformational Implementation (or TI) [1, 2]. Briefly, the TI approach to programming involves refining and optimizing a program specification written in a high level specification language (Currently, the GIST program specification language [8] is being used for this purpose) to a particular base language (eg. LISP). Refinement and optimization are carried out by applying transformations to program fragments. This process is semi-automatic in that a programmer must both choose the transformation to apply and the context in which to apply it; the TI system ensures that the left hand side (LHS) of the transformation is applicable and actually applies the transformation. The TI system provides a facility for reverting to some previous point in the development sequence from which point the programmer can explore various alternative lines of reasoning.

3. CONCEPTUAL TRANSFORMATIONS AND JITTERING TRANSFORMATIONS

In using the TI system, programmers generally employ only a small number of high level "conceptual" transformations, ones that produce a large refinement or optimization. Examples are changing a control structure from iterative to recursive, merging a number of loops into one, maintaining a set incrementally, or making non-determinism explicit. Typically these transformations have complex effects on the program; they may even have to interact with the user.

Although only a relatively small number of conceptual transformations are employed in a typical TI development, the final sequence is generally quite lengthy. Because the applicability conditions of a conceptual transformation may be quite specialized, usually with a number of properties to prove, much of the development sequence is made up of lower level transformations which massage the program into states where the set of conceptual transformations can be applied. I call these preparatory steps "jittering" steps. Examples of quite low level jittering steps include commuting two adjacent statements or unfolding nested blocks. More difficult jittering steps include moving a statement within a block from the kth position to the first position, merging two statements (eg. conditionals,loops) into one, or making two loop generators equivalent.

4. AN AUTOMATIC JITTERING SYSTEM

Requiring the programmer to carry out the Jittering process detracts from his performance in several ways: it consumes a large portion of his time and effort; it disrupts his high level planning by forcing him to attend to a myriad of details. There is then strong motivation for automating some or all of the Jittering process. The following sections will discuss the types of mechanisms used to actually implement such a system (henceforth known as the Jitterer).

4.1. BLACK BOX DESCRIPTION

The Jitterer is initially invoked whenever the TI system is unable to match the LHS of a transformation *Tk* selected by the user. The Jitterer's inputs are 1) the current program state *C*, 2) a goal state *G* corresponding to the mismatched LHS of *Tk*, and 3) a library of transformations *L* to use in the jittering process. The Jitterer's return value is either a failure message or a program state *S* matching *G* and a sequence of instantiated transformations from *L* which when started in *C* will result in *S*. If the Jitterer is successful, then *Tk* will be removed from its suspended state and applied as specified in state *S*.

If *G* is a conjunction of sub-goals then currently a simple STRIPS like approach is employed in solving each in some determined order. This approach can be both inefficient and unbale to find solutions for certain jittering problems. Improving this process, possibly using some of the techniques proposed by problem solvers in other domains (see Sacerdoti [9] for a survey), remains a high priority item for future research.

4.2. THE GOAL LANGUAGE

Contained in the TI system is a subsystem called the Differencer [5]. The Differencer takes as input a current state pattern and a goal pattern, and returns a list of difference-descriptions between the two (an empty list for an exact match). Each element of the list is a description taken at a particular level of detail, and is written in a goal language I call GL1. For example,

*This research was supported by Defense Advanced Research Projects Agency contract DAHC15 72 C 0308. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. Government, or any other person or agency connected with them.

suppose that the Differencer was passed "if P then FOO" as a current pattern, and "if Q then $A" as a goal pattern (The notation $X stands for a variable pattern matching a single statement). The output of the Differencer would be the following three descriptions in GL1: CHANGE-PREDICATE(P Q); CHANGE-ACTION("if P then FOO" "if Q then $A"), ie. change from one conditional to another *without* going out of the current context; PRODUCE-ACTION-AT-POSITION("if Q then $A" J), ie. use any means necessary, including looking at the surrounding context, to produce a conditional matching "if Q then $A" at position J, J being bound by the Differencer. The above three descriptions form a disjunction of goals, each of which gives a slightly wider perspective. Currently the Jitterer attempts to solve the most narrow goal first. If it is successful, then control returns to TI. If not, it attempts to solve the next higher goal and so on until the list is exhausted.

Other goals of GL1 not included in the above example include EMBED, EXTRACT, DELETE, ADD, COMMUTE, DISTRIBUTE, PROVE-PROPERTY, FOLD and UNFOLD. Because GL1 is the language used to both describe pattern differences by the Differencer and describe desired goal states by the Jitterer, it acts as a common interface between the two.

5. JITTERING PLANS

A small number of jittering plans are capable of solving many of the goals of GL1. Suppose we take for example the goal of the previous section CHANGE-PREDICATE(P Q). This is a specific instance of the more general goal CHANGE-PREDICATE(pattern1 pattern2). There are three basic plans for solving this more general goal: 1) embed pattern1 in pattern2, 2) extract pattern1 from pattern2, or 3) first embed pattern1 in something containing pattern2, and then extract pattern2. Since in our case each pattern is a simple variable, we can rule out the second plan. Similar plans exist for the other two goals of the example. In general, jittering plans can be defined for all but the most detailed goals of GL1.

5.1. STRATEGY AND TACTICS

The plans available for solving a particular goal have been organized into a construct I call a STRATEGY. Each of the plans is known as a *Tactic*. Each STRATEGY construct takes the following form (see [4], [7] for similar planning constructs in other domains.):

```
STRATEGY name
   Relevant-goal: goal the strategy matches
   Applicability-condition: used to narrow
                            strategy's scope
   Default-bindings: bind any unbound goal
                     parameters
   Tactic(1) ...
   Tactic(n) ...
```

To illustrate, let us package the three plans described informally for solving the CHANGE-PREDICATE goal into a STRATEGY construct. Suppose that we wanted to rule out working on goals that attempt to change True to False or False to True, and that we expected both of CHANGE-PREDICATE's parameters to be bound. We would get the following STRATEGY header:

```
STRATEGY change-predicate-from-a-to-b
   Relevant-goal: CHANGE-PREDICATE(Pred1 Pred2)
   Applicability-conditions:
      NOT(Pred1=True ∧ Pred2=False) ∧
      NOT(Pred1=False ∧ Pred2=True)
   Default-bindings: none

                [ various tactics ]
```

We now must define the individual jittering plans or *Tactics* which will achieve the CHANGE-PREDICATE goal. Each *Tactic* construct is composed of a set of constraints which further limit its applicability and an *Action* for achieving the matching goal. Our three plans become formally

```
STRATEGY change-predicate-from-a-to-b
   ...
   Tactic(1) embed
      Applicability-condition: none
      Action: POST(EMBED(Pred1 Pred2))

   Tactic(2) extract
      Applicability-condition: NOT(VARIABLE(Pred1))
      Action: POST(EXTRACT(Pred2 Pred1))

   Tactic(3) embed-and-then-extract
      Applicability-condition: none
      Action:
        SEQ(POST(EMBED(Pred1
                       (#ANY-PRED * Pred2)))
            POST(EXTRACT(Pred2
                         (#ANY-PRED Pred1 Pred2]
```

Here, POST(G) says mark the goal G as a sub-goal to be achieved. #ANY-PRED will match to either AND or OR. SEQUENCE(A1 A2 ... An) says execute the list of actions sequentially; if Ai is of the form POST(G), then do not move on to Ai+1 until G has been achieved. There exist similar functions for executing a sequence of actions in parallel and in disjunctive form. In general, an *Action* can specify an arbitrarily ordered list of actions to take to achieve the STRATEGY'S goal.

5.2. BACKWARD CHAINING

The transformations available for jittering, along with the *Tactics* introduced through the STRATEGY construct, define the methods available for solving a particular goal. Transformations are applied in backward chaining fashion: once a transformation's RHS matches a goal G, variables are bound and the LHS is instantiated. The Differencer is then called in to see if a match exists between the current state and the instantiated LHS. If so, then G is marked as achieved by the application of the transformation. If there is a mismatch, then the disjunction of sub-goals produced by the Differencer will be marked as awaiting achievement.

6. THE JITTERING SCHEDULER

Whenever a new goal is posted (marked to be achieved), the Jitterer attaches to it a list of methods (transformations and *Tactics*) which might solve it. It is at this point that the Scheduler is called in. The Scheduler first must decide among all active goals (ones with at least one untried method) which to work on next. Once a goal is chosen, it must decide which of the untried methods to employ. A set of domain dependent metrics which help the Scheduler make an intelligent decision in both cases has been identified.

6.1. Choosing among competing goals

Length of path: the types of problems presented to the Jitterer by TI do not generally involve a large (over 10) number of transformation steps. Hence, as a path (sequence of

transformations) grows over a fixed threshold, the desirability of continuing it decreases.

Conflict with high level motivation: if the Scheduler is able to determine the user's current high level goal, it may be able to rule out certain goal states as un-productive. For example, if it is known that the user is trying to optimize his control structure by merging a number of loops into one, it may be unwise to try to achieve a sub-goal which produces still another loop. Such a sub-goal would be given low priority.

Ease of achieving: A rough estimate is made of the cost to continue the goal by taking the minimum cost of the untried methods attached to it. This is a rough estimate because there is no easy way to compute exactly the final cost associated with any method, or in fact that any method will lead to a solution.

6.2. Choosing among competing methods

Ease of application: a rough static estimate of how difficult the method may be to apply. In the case of a transformation, how complex is the LHS (eg. how many properties must be proven)? In the case of a *Tactic*, how many sub-goals must be achieved?

User assistance: some methods call for the user to supply needed information. The preference is to avoid bothering the user as much as possible.

Side effects: What undesirable actions will a method take besides the desired one (a qualitative judgment)? For instance, a method which unfolds a large procedure body to produce a certain type of goal pattern is seen as having a large side effect, ie. it tends to "flatten" the functional structure of the program. On the other hand, a method which simply changes a current statement into something matching a goal pattern has very little side-effect; it does nothing to disturb the surrounding context. Prefer small side effects over large ones.

Tactic ordering rules: a STRATEGY writer may provide rules for ordering *Tactics*. These rules take the form "if *Condition* then *Ordering*", where *Condition* can refer to any piece of knowledge known to the STRATEGY at match time and *Ordering* takes the form "Try *Tactic*(J) before *Tactic*(K)" or "Try *Tactic*(N) last". The Scheduler makes use of this information when choosing among competing *tactics* for a particular goal.

7. CONCLUSION

The purpose of this paper has been to present a prototype Jitterer with enough domain knowledge to deal *competently* with the types of jittering problems typically encountered in a TI environment. The prototype Jitterer described is currently being implemented in the Hearsay-III knowledge representation language [3] and represents a preliminary system. Future systems will deal much more with *performance* issues (see for example the next section).

8. FURTHER RESEARCH

There are generally many ways of achieving a given jittering goal. The metrics of section 6 give some help in ordering them. Even so, in some cases the Jitterer will produce a solution not acceptable to the user. A simple minded approach (and the one currently employed) is to keep presenting solutions until the user is satisfied. A better approach is to allow the user to specify what he didn't like about a particular solution and allow this information to guide the search for subsequent solutions. In fact, the user may not want to wait until an incorrect solution has been presented, but give jittering guidance when the Jitterer is initially invoked (Feather [6] proposes such a guidance mechanism for a fold/unfold type transformation system). Still another approach may be to "delay" jittering until more high level contextual information can be obtained. Both user guidance and delayed reasoning are being actively studied for inclusion in future jittering systems.

Acknowledgments

I would like to thank Bob Balzer, Martin Feather, Phil London and Dave Wile for their contributions to this work. Lee Erman and Neil Goldman have provided willing and able assistance to the Hearsay III implementation effort.

9. REFERENCES

1. Balzer, R., Goldman, N., and Wile, D. On the Transformational Implementation Approach to programming. Second International Conference on Software Engineering, October, 1976.

2. Balzer, R. TI: An example. Research Report RR-79-79, Information Sciences Institute, 1979.

3. Balzer, R., Erman, L., London, P., Williams, C. Hearsay-III: A Domain-Independent Framework for Expert Systems. First National Conference on Artificial Intelligence, 1980.

4. Bulgas-Rozas, J. *GOAL: A Goal Oriented Command Language for Interactive Proff Construction.* Ph.D. Th., Computer Science Dept., Standford University, 1979.

5. Chiu, W. Structure Comparison and Semantic Interpretation of Differences. First National Conference on Artificial Intelligence, , 1980.

6. Feather, M. *A System For Developing Programs by Transformation.* Ph.D. Th., Dept. of Artificial Intelligence, University of Edinburgh, 1979.

7. Friedland, P. *Knowledge-based Hierarchical Planning in Molecular Genetics.* Ph.D. Th., Computer Science Dept., Stanford University, 1979.

8. Goldman, N., and Wile, D. A Data Base specification. International Conference on the Entity-Relational Approach to Systems Analysis and Design, UCLA, 1979.

9. Sacerdoti, E. Problem Solving Tactics. Technical Note 189, SRI, July 1979.