Heuristic control of design-directed program transformations

by CHRISTINA L. JETTE
University of Washington
Seattle, Washington

INTRODUCTION

This paper describes a class of semantic source-to-source program transformations called design-directed program transformations (DDPT) for use in a transformational implementation (TI) approach to programming. A methodology is developed for applying such transformations based on symbolic evaluation and experimental computation of programs. A DDPT is a cognitive model of source-to-source transformations; it knows what it is trying to accomplish and contains a strategy of how to accomplish it. A DDPT is more than a syntactic pattern replacement rule; it is a semantic program transformation which is intuitively closer to one that a good programmer would invoke in transforming his program.

DDPTs are a natural extension to TI systems. The importance of a TI approach to programming has been made earlier, most notably by Balzer et. al. In a TI environment, the user interactively constructs and modifies a program by applying “correctness-preserving” transformations. Viewed as a programming methodology, this approach strives to relieve the user from worrying about the details of the actual implementation, thereby transforming his role to one of “designer” and “optimizer,” delegating the role of “implementer” to the TI system.

To date, TI systems provide only syntactic transformations. A syntactic transformation can usually be represented by a production-like pattern replacement rule of the form “LHS-pattern⇒RHS-replacement-pattern.” (LHS is left-hand side, RHS is right-hand side.) The user selects a particular rule. The TI system tries to match the LHS-pattern with a portion of the program to be transformed (called the target program). If successful, the RHS-replacement-pattern is substituted for the matched portion of the program.

A DDPT is not so easily described as a syntactic replacement rule because the LHS and RHS patterns are not “one-to-one” with the statements in a programming language, i.e., they contain user-supplied descriptions of what the program fragment is doing such as “update,” “put,” etc. Further, the RHS-replacement-pattern cannot always be specified a priori. It is instead dynamically generated based on the interaction of the specific target program and the specific DDPT. Consequently, a DDPT is defined by 1) an input pattern whose language is semantically rich and 2) a procedure which derives instances of the RHS-replacement-pattern. This pattern instance encodes the underlying strategy of the transformation. In the next section, we give a detailed example of a DDPT definition called BYPASS-LOOP.

What distinguishes the DDPT method from the more syntactic approach is 1) the goal-directedness of a specific DDPT (e.g., “bypass” a looping computation if possible), 2) the nature of the input pattern (e.g., pattern elements contain descriptive annotations such as “update,” “put,” etc.) and 3) the synthesis of replacement program fragments which when substituted into the program, achieve the overall goal of the transformation (e.g., bypass a loop for special cases).

Considering only syntactic transformations leads to several operational difficulties. They include 1) Size—An enormous number of transformations exist. How can their numbers be reduced? 2) Selection—How are transformations selected/accessed? 3) Control—How are transformations applied, i.e., in what order should they be tried and how are they invoked? In this paper we show how these problems can be alleviated in some instances using DDPTs.

Summary

In the next section, we present several examples of design-directed program transformations, and compare the design-directed paradigm to a more traditional syntax based paradigm. In the third section we relate this work to others and in the fourth section we give our conclusions.

DDPT EXAMPLES AND COMPARISONS

In this section, we define the steps of the transformation process. Based on these steps, and using detailed examples for illustration, we compare the syntax based pattern-replacement rule paradigm with the design-directed paradigm. The process of manipulating a program can be factored into four steps. They are selection, matching, substitution and replacement. A transformation “rule” is selected (either
automatically or by the user), and matched to a program fragment. If the match is successful, the variable portions of the matched part are substituted for the variable portions of the replacement pattern. Finally, the replacement pattern replaces the matched target program fragment. In the next two examples, we show how the DDPT approach extends each of these steps by making them more “computational” in nature.

**BYPASSLOOP DDPT**

This example illustrates how the DDPT method extends the matching and substitution processes. We first look at the syntactic approach.

Consider the sequence of syntactic transformation rules necessary to transform the program fragment

\[(a) \text{while } i \leq n \text{ do } (b[i] := b[i] + p \times (i-1); i := i+1)\]

into the program fragment

\[(b) \text{if } p = 0 \text{ then } i := n + 1 \text{ else while } i \leq n \text{ do } (b[i] := b[i] + p \times (i-1); i := i+1).\]

This transformation takes advantage of the special case for which the loop does no “significant” processing, i.e., for “\(p = 0\)” Figure 1 defines six transformation rules, T1 through T6, to transform (a) into (b).

In this approach, T1 is applied to the target program (a) by matching the LHS-pattern of T1 to (a). The result of this match is a list of (argument,value) bindings, e.g.,

\[(A : \langle b[i] := b[i] + p \times (i-1) \rangle)\]

Next, the value for A is substituted in the RHS-replacement-pattern. Note that the predicate “\(p = 0\)” which is bound to \(R\) in the RHS-replacement-pattern must somehow be specified by the user. The RHS-replacement-pattern replaces the matched portion, producing a new program fragment to which T2 will be applied, etc. The program fragment (b) above is the result of applying T1 through T6 to (a).

The problems inherent with this method are:

1. The overall goal to “bypass the loop for the special case that the no computation path is taken,” is completely obscured by the details of invoking the correct sequence of applicable transformations.
2. Six transformation rules are invoked in this case, but these six are not necessarily the only sequence of syntactic rules which would transform (a) into (b). How do we select the applicable ones, and how do we know which ones are available for selection?
3. The user is completely responsible for the selection of a correct sequence—the system merely carries out each selection by matching the appropriate code segment and accomplishing a straightforward replacement.
4. The user is also responsible for discovering properties of the program which might be derivable automatically by a more intelligent system, such as the predicate “\(p = 0\)” and the action “\(i := n + 1\)” in the example above.

In comparison, the design-directed approach has a single DDPT called BYPASSLOOP, whose strategy is to bypass a looping computation for the special cases in which the loop does no “significant” processing. We call this strategy the abstract design of the DDPT. It is abstract in that it does not specify the “details” of the target program to which it applies. It does, however, specify the necessary constraints of the transformation. The abstract design specifies that the target loop has a “no computation” path, but it does not specify the specific action sequences of that loop. This strategy is encoded as the procedure which generates the replacement program fragment for a DDPT. A DDPT is defined by giving: 1) An input pattern, and 2) A replacement fragment constructor procedure which constructs an incompletely specified replacement program fragment based on the given target program and the underlying strategy or goal of the transformation. It contains both concrete, or known actions and predicates and abstract or unspecified actions and predicates. The concrete parts are derived from the matched input pattern; the abstract parts are exactly those portions of the yet-to-be-transformed program which must be synthesized to achieve the overall goal of the transformation.

For BYPASSLOOP, the input pattern is:

\[\text{do; (transient-updates | non-transient-updates); od}\]

where “\(\text{do; \ldots ; od}\)” is a pattern which denotes a looping computation, “\((\text{transient-updates | non-transient-updates})\)” is an alternation pattern which denotes one of two specializations of “update.” (A transient variable is one with a non-repetitive value structure, e.g., an non-array. Hence a transient-update is an assignment to a transient variable, e.g., \(i := i + 1\).) Note that the input pattern for BPL is considerably more abstract than the patterns T1-T6 in Figure 1.

The BPL replacement fragment constructor procedure may informally be stated as:

1. Let \(M\) be the matched portion of the target program and the input pattern. Let \(TU\) be the transient variables, and NTU be the non-transient variables. Create a conditional statement such that:
   (a) The else branch is \(M\)
   (b) The true branch is \(TU := F(TU)\),
   (c) The predicate condition is \(r\) \(\Rightarrow \text{if } r \text{ then } TU = F(TU) \text{ else } M\).
2. Define \(r\) to be exactly that abstract predicate which is true for the bypass path to be taken. That is, \(P[M] \Rightarrow P \text{ and } r(TU := F(TU)) \Rightarrow Q\).
3. Define \(TU := F(TU)\) to be those update actions which may be expressed as non-looping computations. For all \(x\) in \(TU\), \(F(x)\) is the value \(x\) would have had on exit from the loop if the loop had executed.
RULES:

T1: P(A)Q => P(if R then A else A)Q

T2: P(if R then A else B; C)Q => P(if R then A; C else B; C)Q

T3: Partial evaluation, i.e., symbolic evaluation of the result of substituting known values for variables, actual parameters for formal parameters in procedures, etc.

T4: P(x:=x)Q => P(noop)Q

T5: P(while R do if S then A else B)Q => P(if S then (while R do A) else (while R do B))Q when S(A)S and S(B)S.

T6: Computing linear relationships and values of variables.

TRANSFORMATIONS:

let target program fragment = while i<=n do (b[i]:=b[i]+p*(i-l);i:=i+l);

=> T1 while i<=n do (if p=0 then b[i]:=b[i] + p*(i-l) else b[i]:=b[i] + p*(i-l); i:=i+l)

=> T2 while i<=n do (if p=0 then (b[i]:=b[i] + p*(i-l);i:=i+l) else (b[i]:=b[i] + p*(i-l);i:=i+l)

=> T3 while i<=n do (if p=0 then (b[i]:=b[i] + p*(i-l);i:=i+l) else (b[i]:=b[i] + p*(i-l);i:=i+l))

=> T4 while i<=n do (if p=0 then i:=i+l else b[i]:=b[i] + p*(i-l);i:=i+l)

=> T5 if p=0 then while i<=n do i:=i+l else while i<=n do (b[i]:=b[i] + p*(i-l);i:=i+l)

=> T6 if p=0 then i:=n+1 else while i<=n do (b[i]:=b[i] + p*(i-l);i:=i+l)

Figure 1—The transformation process using syntactic pattern-replacement rules.

Figure 2 shows the DDPT method applied to BYPASSLOOP. The first step matches the input pattern to the target program fragment, "while i<=n do...". The second step invokes the BPL replacement fragment constructor procedure above. This results in an incompletely-specified program fragment whose abstract elements r and TUr := F(TU) are constrained by the BPL strategy. Finally in Step 3, these abstract portions are synthesized with p=0 for r and i=n+1 for TUr := F(TU). (The details of the synthesis procedure are beyond the scope of this paper.)

The advantages of this approach are:

1. The overall goal (to bypass the loop) directs the transformation process.
2. Once BYPASSLOOP is selected, any "low-level" analysis is carried out automatically (e.g., partition variable assignments into types).
3. The system derives properties such as the predicate p=0, instead of requiring the user to specify them.
4. The input pattern and replacement fragment construc-
input pattern: "do; (transient-updates | non-transient-updates); od"

target program: while i ≤ n do (b[i]:=b[i]+p*(i-1); i:=i+1)

1. MATCH BPL input pattern to target program. Analyze statements in loop. Partition variable assignments into transient and non-transient updates. (See section 2.1.)

=> <do; ••• ;od : while i<=n do ••• >
<transient-updates : i:=i+1>
<non-transient-updates : b[j]:=b[j]+p*(j-1), 1<=j<=n>

2. INVOKE BPL replacement fragment constructor procedure.
(a) Bind M to value of <do;...;od>
(b) Bind TU to {i}
(c) Bind NTU to {b}
(d) CREATE-CONDITIONAL(pred = "r", truepart = "TU:=F(TU)", elsepart = M)

=> "if r then TU:=F(TU) else M"

3. SYNTHESIZE abstract parts "r", and "TU:=F(TU)"
(a) Compute F(TU), the final values of x in TU.

=> i:=n+1

(b) Compute predicate "r". Let x_i denote the i-th value of x. For each non-transient variable x, for each iteration i, construct a predicate r_i = AE(x_0 = x_i). (AE is abstract evaluation [BIGG77a].) Let r = r_1 and r_2 and ... and r_n.

=> AE(b_0[1]=b[1]) and ... and AE(b_0[n]=b[n])
=> AE(b_0[1]=b_0[1]+0) and ... and AE(b_0[n]=b_0[n]+p*(n-1))
=> T and p*l=0 and ... and p*(n-1)=0
=> p=0

(c) Return fully instantiated program fragment.

=> "if p=0 then i:=n+1 else while i<=n do ..."

Figure 2—DDPT method applied to BYPASSLOOP.

tor procedure of the transformation applies to a large class of target program fragments.

Cancel DDPT

A general selection procedure is yet another important aspect of our system which is illustrated in this example. We are experimenting with automating this step of the transformation process by using the program annotations supplied by the programmer for clarity. A database of related annotations is maintained. For example, "loop," a syntactic program control structure element, and "update," a descriptive action verb element, are related by a relation R-BPL. The selection procedure essentially constructs a list of those related elements contained in a target program which are related in the database. Each relation (e.g., R-BPL) is suggestive of one or more DDPTs (e.g., BPL). Hence the selection procedure returns a list of potential transformations to be tried.

From the collection of the Computer History Museum (www.computerhistory.org)
The program in Figure 3 processes characters given in an array text [0:n]. In and out are two integer pointers such that during execution of the main loop,

(a) text[1:out] is the text processed "so far," and
(b) text[1:in] is the output or "saved" text.

The text processing 1) replaces linefeeds by blanks, 2) removes redundant blanks and 3) removes non-alphabetic characters. (This is an example discussed in Reference 1.)

Abstractly, the main loop of the program is:

while moretext do
  (get next character from input;
   put character in output;
   case on character type:
     linefeed: (replace character by blank;
                   if redundant blanks then remove character from output);
     blank: if redundant blanks then remove character from output;
     alpha: noop;
     else: remove character from output);

In Figure 3 we "fill in the details" of this abstract program; the abstract operations above are left as annotations.

An applicable DDPT is the CANCEL transformation. Intuitively the strategy of this transformation is to rearrange the action sequences so as not to have to "undo" or "cancel" any previous action. In this case, CANCEL will eliminate action sequences such as ["put": . . . ; "remove"] by replacing the "puts" and "removes" with "noops", i.e., ["put": . . . ; "remove"]=>["noop": . . . ; "noop"]. Applying CANCEL to the abstract program above yields the transformed abstract program:

while moretext do
  (get next character from input;
   noop:
   case on character type:
     linefeed: if not redundant blanks then put blank in output;
     blank: if not redundant blanks then put character in output;
     alpha: put character in output;
     else: noop)

Those portions in italics are program parts affected by the transformation. The complete program is shown in Figure 4. In this version, note that the "put" is done only when needed: the "replace" and "remove" actions have been eliminated.

How can one use the program's annotations in Figure 3

```
procedure processtext(var text:array[0:n] of char);
var ch:char; in,out:integer;
begin
  in:=out:=0; text[0]:=" "; /** initialize **/
  while out<n do
    begin
      /** get next character from input **/
      out:=out+1;
      ch:=text[out];
      /** case on character types **/
      case ch in
        linefeed: begin
          /** replace character by blank **/
          ch:=' ';
          /** test for redundant blanks **/
          if text[in-1]=' ' then
            begin
              /** remove character from output **/
              in:=in-1;
            end;
          end;
        space: begin
          /** test for redundant blanks **/
          if text[in-1]=' ' then
            begin
              /** remove character from output **/
              in:=in-1;
            end;
        else: begin /** noop **/ end;
        end; /** case **/
    end; /** while out<n **/
end; /** procedure processtext **/
```

Figure 3—Program to process characters.

```
procedure processtext(var text:array[0:n] of char);
var ch:char; in,out:integer;
begin
  in:=out:=0; text[0]:=" "; /** initialize **/
  while out<n do
    begin
      /** get next character from input **/
      out:=out+1;
      ch:=text[out];
      /** case on character types **/
      case ch in
        linefeed: begin
          /** replace character by blank **/
          ch:=' ';
          /** test for redundant blanks **/
          if text[in-1]=' ' then
            begin
              /** remove character from output **/
              in:=in-1;
            end;
        space: begin
          /** test for redundant blanks **/
          if text[in-1]=' ' then
            begin
              /** put character in output **/
              in:=in+1;
              text[in]:=ch;
            end;
        else: begin /** noop **/ end;
        end; /** case **/
    end; /** while out<n **/
end; /** procedure processtext **/
```

Figure 4—Processtext modified by DDPT CANCEL.

From the collection of the Computer History Museum (www.computerhistory.org)
to select the CANCEL DDPT? If the program contains a
"put" followed by a "remove," then there is a chance that
the CANCEL transformation is applicable. A database of
relationships is maintained for potentially related annota-
tions. The system searches the program, constructing a list
of these related elements. For this example, there are several
instances of a relationship named R-CANCEL. In the first
abstract program above, there are three paths containing the
sequence [put: ...; remove]. The pair (put, remove) is
contained in the database under the relation R-CANCEL.
The system collects three instances:

R-CANCEL:
{(PUT001,REMOVE001),(PUT001,REMOVE002),
 (PUT001, REMOVE003)}

The presence of these related annotations in the program
causes the corresponding DDPT, CANCEL, to be selected.
Later, these annotations are used to locate the specific in-
stances of these actions in the match input pattern step of
the transformation process. The CANCEL replacement
fragment constructor procedure analyzes the target pro-
gram, reorders and eliminates the offending action se-
quen ces, and produces an incompletely specified replace-
ment program fragment (as with BPL). The synthesizer fills
in those remaining abstract portions producing the program
shown in Figure 4.

In addition to automating more of the transformation pro-
cess, the advantages to having the transformation system aid
in the selection part include

1. Elimination of the problem of "pointing" to the places
within the program text where the transformation is to
be applied—these places are pointed to by the anno-
tations and remembered by the system.
2. The program is manipulated at the semantic level, e.g.,
manipulate the verbs "put" and "remove," rather than
at the syntactic level, e.g., manipulate "text[in]: =ch" and
"in:=in—1."

Other DDPTs

Many program manipulations fall naturally into the design-
directed paradigm, because of the ability to analyze the
"matched" portion of the program in order to synthesize
replacement program fragments—the replacement fragment
constructor procedure built into each DDPT can be quite
general.

In this section, we list a number of DDPTs of interest.
The reader is referred to Reference 9 for more details.

FLAG-MONITOR—Rep 1 ec a rbitrary boolean test with an
equivalent boolean flag variable which "monitors" the
original condition.

DO-EITHER—Generates (synthesizes) "special case" pro-
gram fragments from a specification of the "general case."

GENERALIZE—Synthesizes "general case" program frag-
ments from a sequence of "special cases."

EXTEND—Data structure extension 13 (A form of GEN-
ERALIZATION.)

RELATION TO OTHER RESEARCH

As noted earlier, most other transformation systems are
syntax-based and two "rule-types" have emerged. One is
the production-like syntactic pattern replacement rules
shown earlier; 1,8,10,12,13 the other is a system of rules which
manipulate recursive programs. 6,7,11,14 The restrictions im-
posed by these methods have been pointed out in the second
section. Mostly the problems center on control and the de-
gree of complexity allowed in transformations.

The major differences between our rules and their syntax-
based counterparts rests in the degree of flexibility in speci-
fying the "patterns" associated with the transformation
process. Our matching mechanism allows for more general
pattern matching; the substitution and replacement mecha-
nism is more computational in nature, and we provide a
mechanism for automatic selection based on the program's
annotations. The syntactic rules are both useful and neces-
sary in a TI system; we simply extend the type of rules
available to the user.

Design-directed transformations extend the work of Bigger-
staff and Johnson, 3,4 in which automatic program synthe-
sis is viewed as a process directed by known abstract pro-
gram designs.

CONCLUSIONS

The design-directed approach to program transformations
is a natural extension to other, more syntactic-oriented ap-
proaches. DDPTs are shown to have more general selection,
matching and replacement procedures than their syntax-
based counterparts. The advantages of this generality are

• Higher degree of automation.
• Each single transformation has a larger scope (it affects
a larger target program fragment).
• Transformations are intuitively closer to ones a good
programmer would select in transforming his program.
• Reduction in the number of transformation rules at the
user level.
• Makes use of system's ability to analyze and reason
about programs (e.g., using symbolic evaluation and
program synthesis).
• Each transformation applies to a large class of target
programs.

We are currently implementing these transformations and
several others in LISP on a DEC 2020, at the University of
Washington.

ACKNOWLEDGMENT

The author would like to thank Dr. Ted J. Biggerstaff for
his influence, guidance and support in this research. Notable
contributions to portions of this work are attributed to his
continuing interest. The author is also indebted to Professor
David L. Johnson for his continued support and suggestions.
Finally, thanks is due to the referees whose comments greatly improved the content of this paper.

REFERENCES
