Applications of the TAMPR Transformation System

TJ Harmer and PJ McParland
School of Information and Software Engineering
University of Ulster at Jordanstown
Newtownabbey BT37 0QB
United Kingdom.

JM Boyle
Technology Division
Argonne National Laboratory
Argonne IL 60439
USA.

November 1, 1998

Abstract

In this paper we present an overview of the uses of the TAMPR transformation system and present experience with using transformation in industrial applications. TAMPR is a *fully automatic*, rewrite-rule based program transformation system. From its initial implementation in 1970, TAMPR has evolved into a powerful tool for generating correct and efficient programs from specifications. The specification and transformation community has long claimed that program transformation and specification provides a powerful, labour-saving means of developing efficient and correct programs. What is the evidence for this statement? We present experience with using TAMPR transformation in applications which range from efficient implementation of numerical mathematical algorithms to inter-language transformation and the Year 2000 problem.

1 Introduction

The field of program specification and transformation makes the claim that better programs can be produced more cheaply with computer assistance than by hand. The specification-transformation methodology involves *deriving* an executable program from a specification by means of correctness-preserving *program transformations*. Because each transformation preserves correctness, such a derivation will produce a program that is just as correct as its specification, but usually much more efficient.

Thus, the specification-transformation methodology is a means of producing verified programs. The process is attractive not only because it saves the expensive labour involved in programming, but also because it greatly reduces the cost of proof. Instead of requiring a large and complex proof of correctness that is specific to the program being developed, the specification-transformation methodology breaks the proof into pieces that are both of manageable size and sufficiently general that they can be immediately reused for other programs.

This has been the promise of program specification and transformation, a promise that has remained unfulfilled for a quarter century. How close is it to being realized? We provide evidence that the TAMPR (Transformation Assisted Multiple Program Realization) System [3, 7, 4], is beginning to fulfill major portions of the promise of program specification and transformation.

We will first give an overview of TAMPR and TAMPR-style transformation rules. Then we consider a range of example areas briefly, to give an overview of the range of application areas and the achievements in those areas, and consider the area of reverse engineering of COBOL programs and the YEAR 2000 problem in such programs in detail. Finally we conclude with observations about our experience of the commercial use of transformation and whether transformation will gain industrial respectability.

2 TAMPR Transformation

The idea and basic notation for TAMPR transformations came from Chomsky's transformational grammar, developed in the late 1950s for use in describing natural languages [10]. However, Chomsky's approach to transformational grammar was theoretical rather than practical. To make automated program transformation possible, TAMPR extended Chomsky's idea in three important ways:

- TAMPR introduced the idea that the rules of transformational grammar should be *intra-grammatical*; that is, the result of applying a transformation rule should be guaranteed to be again a sentence (program) in the same language as was the sentence to which the rule applied. Given a set of transformation rules that are intra-grammatical, it is easy and natural for them to apply repeatedly, the input of one rule being the result of applying another.

- TAMPR added a very simple *control language* for describing how to apply transformation rules. This control language makes possible fully automatic application of transformation rules.

- TAMPR provided a *computer implementation* of transformational grammar. Without a computer implementation, it is difficult to explore large applications of transformational grammar.

2.1 Transformations and Derivations

A TAMPR transformation is a rewrite rule, consisting of a pattern and a replacement defined using the syntax of a wide–spectrum language, the *subject language*, in which the programs being transformed are written. A typical TAMPR transformation is

```
.sd.
   NOT ( <expr>"1" AND <expr>"2" )
==>
   NOT ( <expr>"1" ) OR NOT ( <expr>"2" )
.sc.
```

Here the non–terminal symbol `<expr>` is from the subject language grammar and may be thought of informally as an expression. In this transformation

- the pattern text follows `.sd.` ("structural description") and precedes the arrow (`==>`);

- the replacement text follows the arrow and precedes `.sc.` ("structural change"); and

- `<expr>"1"` in the replacement stands for whatever is matched by `<expr>"1"` in the pattern, etc.

Application of this transformation converts a statement such as

`NOT(a < b AND Flag)` $\longrightarrow$ `NOT(a < b) OR NOT(Flag)`

Repeated application will also convert an expression such as

`NOT((a < b AND c) AND Flag)` $\longrightarrow$ `(NOT(a < b) OR NOT(c)) OR NOT(Flag)`.

When a transformation is applied to a program, *every* fragment of the program that matches the pattern text is substituted by the corresponding text created from the replacement.

A group of related transformations can be grouped together to create a transformation sequence. Thus, a sequence of transformations to perform simplification according to de Morgan's Laws is

```
.transform *.{
<expr> {
.sd.
   NOT ( <expr>"1" AND <expr>"2" )
==>
   NOT ( <expr>"1" )  OR  NOT ( <expr>"2" )
.sc.
.sd.
   NOT ( <expr>"1" OR <expr>"2" )
==>
   NOT ( <expr>"1" ) AND NOT ( <expr>"2" )
.sc.
.sd.
   NOT ( NOT <expr>"1" )
==>
   <expr>"1"
.sc.
}
}
```

The order of the transformations in the sequence indicates the order in which matching should be attempted. The transformation sequence is defined to be applicable in a particular program context, `<expr>` in this case, which is called the *dominating symbol*. The pattern and replacement of each transformation must (and are verified to) be a syntactically valid form that can be derived from the dominating symbol (`<expr>` in this example).

For most transformations the transformation sequence is applied to exhaustion, i.e., upon termination no transformations in the sequence can apply anywhere in the subject program. Such an application strategy is defined by the transformation traversal definition, `.transform *.`. Other application strategies are available to the transformation programmer if exhaustive application is not required; the interested reader is referred to [1]. A complete definition of a transformation sequence, such as the preceding one, may be given a name and stored for subsequent use by a transformational programmer.

A number of transformation sequences can be grouped to create a transformational *derivation*. A derivation defines the transformation sequences to be applied, by stating the name of each transformation sequence, and the order in which these transformations should be applied. The first transformation sequence is applied and when it cannot be applied further the second is applied and then the third and so on. The preceding transformation sequence for de Morgan's Laws is a simple example of one of the simplifications that might be performed in general purpose boolean expression simplification.

2.2 Correctness of Transformations

Transformations manipulate syntactic objects in a program; they are not string manipulation rules. A transformation's pattern and replacement are syntactically correct statements in the subject language. The pattern of a transformation cannot match a syntactically incorrect program, and the replacement of a transformation cannot introduce a syntactically incorrect form into a program. The transformational process is always one of manipulating syntactically correct programs.

In practice, most transformations we write are no more complex than the preceding examples; the power of transformations comes from the cumulative effect of, perhaps, thousands of applications of a few dozen transformations, each application performing some simple, local change to a program fragment. Because transformations are applied entirely automatically, the number of applications is of little significance to the developer or user of the tool.

What *is* significant is the simplicity of each transformation; to see that an application of a simple transformation preserves the meaning of a program is usually trivial. (However, because transformations are formal objects, formal proofs can be used to guarantee the correctness of more complex transformations [16].) Simple transformations make reasoning about the rule easier and facilitate informal or formal proofs of transformation termination.

If each of several transformations preserves meaning, then their combined application also preserves meaning. Thus, an implementation derived by correctness–preserving transformations is known to be as correct as the specification from which it is derived.

In writing transformations one attempts to identify an appropriate algebra for the forms being manipulated. In the preceding example, these forms are Boolean expressions and one can draw on logic, specifically de Morgan's Laws, for the required algebra. An algebra that specifies the relationship between one form and another facilitates the development of transformations by providing a simple model for the rules; moreover, an algebra simplifies verification of transformations because the algebraic law can be proved correct separately, and then used as a lemma in the proof of the transformation. Often, a transformation is little more than a recasting of the algebraic law, as in the example of de Morgan's Laws.

The exhaustive application of transformations results in programs with particular, desired properties–in the preceding example, simplified Boolean expressions. The application of sequences of such transformations results in programs having a number of such properties. Each transformation sequence can use the properties established by prior sets to reduce the number of transformational cases that it must consider; for example, transformations applied after the Boolean simplifications discussed earlier can assume that Boolean expressions are in simplified form.

The approach of identifying special properties of a program also facilitates reusing derivations. A derivation is documented by stating the particular properties of input programs that it assumes and by stating the properties that hold for the corresponding output programs following the application of the derivation.

The TAMPR system may be summarized as providing

- A restricted repertoire of constructs for expressing transformations;
- A declarative semantics for transformations;
- Application of transformations to exhaustion;
- An emphasis on *sequences of canonical forms*;
- Completely automatic operation; and
- The ability to effortlessly "replay" the application of transformations.

3 Deriving Efficient Numerical Mathematical Algorithm Implementations

The first major industrial application of TAMPR was to derive the programs included in the LINPACK package of programs for solving systems of linear equations [11]. The goal of the LINPACK project was to provide "certified" software for solving linear systems—software that had been thoroughly systematized and tested. The resulting LINPACK package consists of 48 Fortran subroutines plus additional test programs, and has been used at sites all over the world for 20 years.

Most LINPACK routines are available in four versions: complex single- and double-precision arithmetic, and real single- and double-precision arithmetic. The possibility to benefit from using program transformation is obvious: write only one of these versions, the complex single-precision version, by hand, let this version play the role of a specification, and derive the other three versions from it automatically by program transformation.

TAMPR was used to perform these derivations. In addition, it was also used to format (pretty-print) the resulting LINPACK Fortran programs, using indentation to display their logical structure (even though Fortran 66 is a notoriously unstructured language). Use of TAMPR to produce LINPACK thus reduced labour, assured uniformity among the programs, improved their readability, and enhanced their correctness.

4 Efficient Implementations from Algorithmic Specifications

The LINPACK work involved the refinement of programs at a (relatively) low level—hand-crafted implementations of the algorithms were transformed in a systematic way to a number of alternative forms. This posed the question as

to whether transformation was more useful with higher level algorithmic forms were the need to improve efficiency was important. The TAMPR transformation system itself provided an ideal candidate for experimentation with higher level algorithmic specifications.

4.1 Deriving efficient Implementation of TAMPR Components

The specifications for the TAMPR transformer and other components are written in pure functional Lisp (functional Scheme). These specifications make full use of data abstraction, permitting the algorithms to be expressed in terms of parse trees, etc., rather than Lisp $cons$, car , and cdr. We use TAMPR to derive its own implementations in C or Fortran (or Algol or Pascal or Java) from this high-level specification. The derivation includes a number of steps to improve the efficiency of the derived implementation; for example, it performs extensive unfolding and simplification of data abstractions.

The flexibility inherent in program specification and transformation is well illustrated by the fact that TAMPR can derive both sequential and parallel implementations from the *same* specification [4]. The derived sequential TAMPR (compiled C) implementation is approximately 25% faster than the equivalent compiled (Franz) Lisp specification.

The transformation system and its components are maintained and enhanced exclusively by transformation. The source is Lisp although it has not been processed by a lisp interpreter in more than ten years.

The success in deriving highly efficient implementations of algorithmic specifications led us to experiment in other area where need for efficiency is of crucial importance.

4.2 Telecommunication Software

A major telecommunication company[1] develops mobile communication systems using a Pascal-like programming language to describe processes and a finite state machine (defined by a decision table) to describe when processes will execute. The process and state machine serve as a system description that is given to contract programmers to refine to a executable program. A significant task in this development is the balancing of process execution time because the host architecture runs only a limited operating system.

Using transformation we were able to automate the development process by generating appropriate control code that realizes the state machine and define rules to implement the process context switching required by the host architecture.

4.3 High-Level Specification of Numerical Mathematical Algorithms

Numerical mathematical algorithm pose an interesting problem area. Many problems have clear easy-to-follow algorithmic specifications and yet implementation of such algorithms are rarely clear or easy-to-follow. The problem is the high computational requirements of such algorithms require that the natural implementation of such algorithms must be refined to achieve acceptable execution performance.

In a derivation of a program to solve hyperbolic partial differential equations (PDEs) [9] [8] we illustrated that we can compete with the hand-crafted implementations.

Our particular Hyperbolic PDE problem was a practical problem of interest to a colleague— HPDEs arise in fluid flow problems involving *shock*, such as supersonic flow, traffic flow, and the breaking of waves on the beach. We started from a higher-order, pure-functional specification for a cellular-automaton algorithm for solving hyperbolic PDEs. Again, the specification made no concessions to efficiency and made full use of data abstraction. Version tailored for a sequential machine, a vector parallel computer and an MIMD computer could be derived from the same specification.

The TAMPR-derived implementations (in Fortran) are highly efficient: that tailored for the CRAY X-MP vector computer is faster than the handwritten program prepared by the inventors of this algorithm, and that for sequential computers (Sun) is again faster than the handwritten version.

[1] We are not permitted to identify the company because the company insists that we are bound by a secrecy clause.

A more significant example of the power of transformation was the derivation of an efficient implementation of a parallel eigenvalue solver (POT) [6, 13, 14]. For this derivation, we expressed the specification in the ML language. The derivation targets the AMT DAP-510 SIMD parallel array processor. DAP programs are efficient only when most of the program can be executed using *array* operations; such operations apply the same arithmetic or logical operation to up to 1024 elements of a vector or rectangular array simultaneously The derivation uses an *array algebra* to generate efficient DAP code from the specification [14] [12]. Again, the TAMPR-derived implementation in DAP Fortran is as fast as the handwritten program.

The previous examples illustrate the traditional rôle of transformation—the refinement of a high-level form to a low-level one. To illustrate the techniques involved in transformation we consider a rather *untraditional* use of transformation—the reverse engineering of legacy programs.

5 Reverse Engineering

The use of the TAMPR system for forward engineering relied on the use of transformations to migrate a specification through a series of canonical forms towards specific implementation form(s). A refinement of this approach can be used to carry out the reverse engineering of existing implementations. With reverse engineering the objective is to extract an abstract representation of an implementation which is unstructured and undocumented. This example application is explained in more detail than previous applications of TAMPR since it requires less domain knowledge to understand the approach taken and it is an good example of the transformational approach.

Many organizations have old programs which are important to the operation of their business. These programs may have been developed in the 1960s or 1970s and constantly updated by many different programmers. Accurate documentation may not exist. However, in many cases these programs cannot easily be replaced by modern systems because embedded in the old programs is important business processes which are recorded only in the source code. In order to replace or to maintain these old programs they must be made more understandable. A TAMPR reverse engineering tool has been developed to reorganize COBOL control structure in an attempt to improve the understandability and the maintainability of the programs, and permit such programs to be the subject of further re-engineering; for example, to facilitate translation to an alternative system or programming language.

Unless an organization enforces rigid coding standards then a program can evolve to become particularly difficult to understand; this is particular true when maintenance is carried out in a piecemeal way and no major reorganization is attempted because such a reorganization would require significant system testing. For example consider the following implementation of Prime numbers[2] in figure 1 which is an exemplar of the control forms present in legacy COBOL.

The obvious way to attempt to make this prime number code more understandable is to replace `IF .. THEN .. GO TO` statements by appropriate loop constructs. Whereas this might work in this specific case it would not work in general since to restructure a COBOL program it is necessary to understand and to make explicit the semantics of the COBOL programming language. Ignoring the semantics of COBOL may result in a program which does not preserve its original meaning.

Thus the first set of transformation derivations applied seek to make the implicit paragraph invocation rules of COBOL explicit. Example COBOL paragraphs from figure 1 include `MAIN`, `START1` and `CALCULATION`. Transformations to simplify COBOL programs must include the following knowledge about paragraph invocation and termination from the COBOL execution model.

1. A paragraph can be executed by a `PERFORM` statement (e.g., `PERFORM GET-A-NUM` at line 190). Such execution transfers control to the first statement of the paragraph, and control returns to the statement following the `PERFORM` statement provided the paragraph terminates by completing (rather than by transfer).

2. A sequence of paragraphs can be executed by a `PERFORM ... THRU ...` statement (e.g., the statement `PERFORM START1 THRU FINISH` at line 160). Such execution transfers control to the first statement of

[2] We are indebted to a COBOL consultant for providing this challenging example.

```
            IDENTIFICATION DIVISION.
            PROGRAM-ID.        PRIME-N.
            AUTHOR.      J.SMITH.
            ENVIRONMENT DIVISION.
            DATA DIVISION.
            WORKING-STORAGE SECTION.
            01  NUM                 PIC 99999.
            01  QUOTIENT            PIC 99999.
            01  REST                PIC 99999.
            01  DIVISION            PIC 99999.
            01  DIVISOR             PIC 99999.
            01  DIVI2               PIC 99999.
00          PROCEDURE DIVISION.
01          PRIME-CALCULATION SECTION.
02          MAIN-PARA.
03              PERFORM PRINT-INSTRUCTIONS.
04              PERFORM START1 THRU FINISH.
05              STOP RUN.
06          PRINT-INSTRUCTIONS.
07              DISPLAY "TYPE A NUMBER WHEN PROMPTED".
08              DISPLAY "TYPE 0 (ZERO) TO END PROGRAM".
09          START1.
10              PERFORM NUMBER-READ.
11              IF NUM EQUAL 0 GO TO FINISH.
12              IF NUM EQUAL 2 OR 3 GO TO PRIME-N.
13              MOVE 2 TO DIVISOR.
14          CALCULATION.
15              DIVIDE NUM BY DIVISOR GIVING QUOTIENT.
16              COMPUTE REST = NUM - (DIVISOR * QUOTIENT).
17              IF REST EQUAL 0 GO TO NO-PRIME-N.
18              COMPUTE DIVI2 = (DIVISOR*DIVISOR).
19              IF DIVISOR = 2 MOVE 1 TO DIVISOR.
20              IF DIVI2 < NUM
21                 ADD 2 TO DIVISOR
22                 GO TO CALCULATION.
23          PRIME-N.
24              DISPLAY NUM " IS A PRIME NUM".
25              GO TO START1.
26          NO-PRIME-N.
28              DISPLAY NUM " IS NOT A PRIME NUM".
29              GO TO START1.
30          FINISH.
31              EXIT.
32          NUMBER-READ SECTION.
33          GET-A-NUM.
34              DISPLAY "PLEASE TYPE A NUM".
35              ACCEPT NUM NOT ON EXCEPTION GO TO FINISH.
36          ACCEPT-EXCEPTION.
37              DISPLAY "NUMBER EXCEPTION!".
38              GO TO GET-A-NUM.
39          FINISH.
40              EXIT.
```

Figure 1: An Example COBOL Program for Prime Numbers

the first–named paragraph, and control returns to the statement following the PERFORM statement provided the last–named paragraph terminates by completing (rather than by transfer).

3. A paragraph can be executed by a GO TO statement that references the name of the paragraph (e.g., GO TO CALCULATION at line 310). Such execution causes transfers control to the named paragraph, with no possibility of return to the statement following the GO TO.

4. A paragraph can be executed as a result of *fall–through*. Such execution occurs when the paragraph textually preceding the paragraph terminates by completing (rather than by transfer), and that paragraph was executed neither by a single–paragraph PERFORM nor as the last–named paragraph in a PERFORM ... THRU ... statement. In figure 1, fall–through occurs from paragraph START1 to paragraph CALCULATION when none of the if–conditions in START1 are true.

In a typical COBOL program a single paragraph may be executed using any or all of the four invocation methods. Thus, understanding the flow of control in a COBOL program that uses a mixture of the four invocation methods is difficult and requires a detailed study of large parts of the program code to obtain the context required for understanding.

A number of canonical forms are used to simplify the COBOL semantics and program structure into a form which can be more readily processed. Figure 2 shows a restructured version of the Prime Number example in which paragraphs have been replaced by functions and all paragraph invocations consist of function calls. Each function has been annotated with code to identify which function is executed when the current function's execution terminates. This code is inserted to make the semantics of COBOL clear so that it is clear that at the end of a function (i.e. paragraph) execution control can progress to the next textual function (i.e. fall-thru) or return to the calling function (i.e. a PERFORM statement).

Although the program in figure 2 has a form in which the implicit semantics of COBOL have been made explicit it is neither elegant nor maintainable. Removing this control information is achieved by using partial evaluation. For each instance of a function invocation a function is created consisting of the original code with the control code replaced by a return statement or a call to the fall-thru function as required by the function invocation. This approach may produce duplicate code but program transformations are used to 'fold' out duplicate code into single functions with associated function calls. The ability to partially evaluate syntactic constructs across a program is one of the techniques which program transformation system can use to specialize an implementation or, as in this case, to simplify a notation.

It is appropriate to make this prime number code more understandable by replacing IF .. THEN .. GO TO statements by appropriate loop constructs. These transformations employ the usual transformation technique of folding and unfolding. Remaining *goto* statements (except those implementing loops) are unfolded and the structure of the resulting code is improved. Markers are left around the original COBOL paragraphs so they can be recreated later as functions. Finally, proper loop constructs are inserted.

In figure 3 the restructured form of the Prime number example is presented. The structure of the program is much simpler than the original and it is therefore easier to understand and to maintain. This notation notation can be translated back into COBOL or into an alternative programming language such as C or Fortran (or Pascal or Java).

This COBOL restructuring tool was applied to several large industrial COBOL programs with significant success.

6 TAMPR as a Year 2000 Analysis and Conversion Tool

Many organizations have systems which they rely on to conduct their business which will not interpret the year 2000 correctly. In the majority of cases the designers of these systems chose to use a two digit field to represent a year. Thus the year 2000 will be represented in such systems as '00' and the system will assume that it represents the year 1900. The designers of systems dating from the 1970s used two digit year fields as a space saving strategy in an era when computer memory was very expensive and the lifetime of a computer system was assumed to be short. Designers in the 1980s (and even 1990s) could not claim either of these mitigating factors for the choice of two digit year fields. By the 1980s the use of two digit year fields was the de facto standard and the expense of changing existing systems and data seemed unnecessary. Today the software industry is paying the price of this lack of investment and foresight in the 1980s and early 1990s.

```
program PRIME_N();
   PRIME_N.PRIME_CALCULATION.MAIN_PARA("PRIME_N.NUMBER_READ.FINISH");
end;
procedure PRIME_N.PRIME_CALCULATION.MAIN_PARA(end-point: String);
   PRIME_N.PRIME_CALCULATION.PRINT_INSTRUCTIONS(PRIME_N.PRIME_CALCULATION.PRINT_INSTRUCTIONS);
   PRIME_N.PRIME_CALCULATION.START1(PRIME_N.PRIME_CALCULATION.FINISH);
   cobol_stop ($not_defined);
   if (end-point=="PRIME_N.PRIME_CALCULATION.MAIN_PARA") then
      return;
   end else
   end;
end;
procedure PRIME_N.PRIME_CALCULATION.PRINT_INSTRUCTIONS (end-pont: String);
   ......
end;
procedure PRIME_N.PRIME_CALCULATION.START1 ($end_para_name$);
   PRIME_N.NUMBER_READ.GET_A_NUM (PRIME_N.NUMBER_READ.FINISH);
   if (NUM .eq. 0) then
      PRIME_N.PRIME_CALCULATION.FINISH (end-point);
      return;
   end else
   end;
   if (NUM .eq. 2 .or. 3) then
      PRIME_N.PRIME_CALCULATION.PRIME_N (end-point);
      return;
   end else
   end;
   move (2, DIVISOR);
   if (end-point=="PRIME_N.PRIME_CALCULATION.START1") then
      return;
   end else
      PRIME_N.PRIME_CALCULATION.CALCULATION end-point);
   end;
end;
procedure PRIME_N.PRIME_CALCULATION.CALCULATION ($end_para_name$);
   divide (NUM, DIVISOR, QUOTIENT);
   compute (REST, NUM - (DIVISOR * QUOTIENT));
   if (REST .eq. 0) then
      PRIME_N.PRIME_CALCULATION.NO_PRIME_N (end-point);
      return;
   end else
   end;
   compute (DIVI2, DIVISOR * DIVISOR);
   if (DIVISOR .eq. 2) then
      move (1, DIVISOR);
   end else
   end;
   if (DIVI2 .lt. NUM) then
      add (2, DIVISOR);
      PRIME_N.PRIME_CALCULATION.CALCULATION (end-point);
      return;
   end else
   end;
   if (end-point: String == PRIME_N.PRIME_CALCULATION.CALCULATION) then
      return;
   end else
      PRIME_N.PRIME_CALCULATION.PRIME_N (end-point: String);
   end;
end;
procedure PRIME_N.PRIME_CALCULATION.PRIME_N (end-point: String);
   ......
end;
procedure PRIME_N.PRIME_CALCULATION.NO_PRIME_N (end-point: String);
   ......
end;
procedure PRIME_N.PRIME_CALCULATION.FINISH (end-point: String);
   ......
end;
procedure PRIME_N.NUMBER_READ.GET_A_NUM (end-point: String);
   ......
end;
procedure PRIME_N.NUMBER_READ.ACCEPT_EXCEPTION (end-point: String);
   ......
end;
procedure PRIME_N.NUMBER_READ.FINISH (end-point: String);
   ......
end;
```

Figure 2: Explicit COBOL Semantics

```
program PRIME_N();
   PRIME_N.PRIME_CALCULATION.MAIN_PARA();
end;
procedure PRIME_N.PRIME_CALCULATION.MAIN_PARA();
   PRIME_N.PRIME_CALCULATION.PRINT_INSTRUCTIONS();
   PRIME_N.PRIME_CALCULATION.START1.FINISH();
end;
procedure PRIME_N.PRIME_CALCULATION.PRINT_INSTRUCTIONS();
   display (values ("TYPE A NUMBER WHEN PROMPTED"));
   display (values ("TYPE 0 (ZERO) TO END PROGRAM"));
end;
procedure PRIME_N.PRIME_CALCULATION.START1.FINISH();
   PRIME_N.NUMBER_READ.GET_A_NUM.FINISH ();
   while (NUM != 0) do
      if (NUM .eq. 2 .or. 3) then
         PRIME_N.PRIME_CALCULATION.PRIME_N ();
      end else
         move (2, DIVISOR);
         PRIME_N.PRIME_CALCULATION.CALCULATION();
      end;
      PRIME_N.NUMBER_READ.GET_A_NUM.FINISH ();
   end;
end;
procedure PRIME_N.PRIME_CALCULATION.CALCULATION ();
   while TRUE do
      divide (NUM, DIVISOR, QUOTIENT);
      compute (REST, NUM - (DIVISOR * QUOTIENT));
      if (REST == 0) then
          PRIME_N.PRIME_CALCULATION.NO_PRIME_N ();
          return;
      end else
        compute (DIVI2, DIVISOR * DIVISOR);
        if (DIVISOR == 2) then
          move (1, DIVISOR);
        end;
        if (DIVI2 < NUM) then
          add (2, DIVISOR);
        end else
          PRIME_N.PRIME_CALCULATION.PRIME_N ();
          return
        end;
      end;
   end;
end;
procedure PRIME_N.PRIME_CALCULATION.PRIME_N ();
   display (values (NUM , " IS A PRIME NUM"));
end;
procedure PRIME_N.PRIME_CALCULATION.NO_PRIME_N ();
   display (values (NUM , " IS NOT A PRIME NUM"));
end;
procedure PRIME_N.NUMBER_READ.GET_A_NUM.FINISH ();
   display (values ("PLEASE TYPE A NUM"));
   while ( accept (NUM , $not_defined) == AcceptException) do
      PRIME_N.NUMBER_READ.ACCEPT_EXCEPTION();
      display (values ("PLEASE TYPE A NUM"));
   end;
end;
procedure PRIME_N.NUMBER_READ.ACCEPT_EXCEPTION ();
   display ("NUMBER EXCEPTION!");
end;
```

Figure 3: The Restructured Prime Numbers Example

The traditional approach to refining an existing 3GL system to make it year 2000 compliant is to use a syntactic analysis tool to identify the program code which has been 'infected' and then to modify the affected lines of code manually. Several tools exist to help developers with this process and a few even suggest to a developer the updates. Program transformation systems with their ability to recognize patterns of program code can be used to automate this process.

This field offered the possibility of being able to demonstrate to industrial organizations the benefits of using a program transformation based approach, in fact the Refine program transformation system [ref] has already been utilized in this way. The year 2000 problem differs from the reverse engineering problem in the scale and scope of the changes which are required. Reverse engineering or program restructuring requires a major revision of the format, style and , possibly, notation in which a system is represented. The transformations concentrate on refining the overall structure of a program so that individual statements or data definitions are not significantly altered. A few statement types such as `goto` statements may be replaced by structured programming constructs such as while loops.

The transformations necessary to automatically make a program year 2000 compliant are detailed and fine grain, concerned with particular statement types and the data that they manipulate. In fact it is not necessary to process those parts of a system that do not contain date manipulation code. Thus although the problems of reverse engineering and year 2000 solution automation appear to be similar there are significant differences.

6.1 Date Identification and Reporting

The first requirement for a year 2000 tool is an analysis tool to identify the lines of code in a program which have been 'infected', i.e. lines of code or data definitions that reference or declare two digit year fields. The tool should also summarize this information in a report which can be used to estimate the extent of the problem for a particular program or group of programs. Many tools exist to complete this analysis process but most are based on a syntactic analysis of the code or even text string processing. The approach described in this section is based on the use of program transformations to parse and analyze a target program for year 2000 infection.

Many existing tools base the analysis of programs on a set of textual patterns for identifiers which would store two digit year fields. Example patterns would include '*date*', '*year*', '*day*' and many others which may depend upon the coding standards or style of variable names in a particular organization. This approach then relies on knowledgeable developers to confirm the list of identified date infected variables. In general it is impossible *automatically* to identify all date fields that are in a program.

A date field might be located by transformations searching for particular text patterns (like the patterns listed above) and/or by particular data type definition. However, such analysis is likely to miss dates that are named without date reference or of a non-standard type definition.

Most organizations which were consulted were content with this approach although an alternative approach was to devise a transformation tool that would concentrate on the ways in which dates enter programs. There are (at least) three ways for COBOL.

- As fields of input files;
- As fields of linkage records; and
- As pre-defined system variables.

Once the input dates are identified transformations can perform an impact analysis and identify date infections in other data in a program. This approach would have been less dependent upon the availability of a knowledgeable developer.

The adopted approach was to use a simple text analysis tool to list likely date infected data items and to present this list to a knowledgeable developer. Once the expert confirmed the list of date infected data items this information was used in a full date recognition analysis. This analysis was implemented using a program transformation tool. The actual analysis is a form of impact analysis to identify the lines of code which are infected based on a set of data items being data infected. The set of data items is dynamically updated during the impact analysis as new data items are identified as being date infected.

```
 IDENTIFICATION DIVISION .
 PROGRAM-ID . C2BDATE .
 ENVIRONMENT DIVISION .
 CONFIGURATION SECTION .
 SOURCE-COMPUTER . PRIME .
 OBJECT-COMPUTER . PRIME .
 DATA DIVISION .
 WORKING-STORAGE SECTION .
* YEAR2000 MED-DATE IS INFECTED
  01 MED-DATE .
  03 DD PICTURE 99 .
  03 MM PICTURE 99 .
* YEAR2000 YY IS INFECTED
  03 YY PICTURE 99 .
* YEAR2000 DT-ARR IS INFECTED
  01 DT-ARR .
* YEAR2000 D-V IS INFECTED
  03 D-V OCCURS 8 PICTURE X .
* YEAR2000 HH IS INFECTED
  01 HH .
* YEAR2000 H IS INFECTED
  03 H OCCURS 2 PICTURE 9 .
* YEAR2000 PP-PP IS INFECTED
  01 PP-PP .
* YEAR2000 PP IS INFECTED
  03 PP OCCURS 3 PICTURE 99 .
  77 I PICTURE 9 VALUE ZERO .
  77 I-M PICTURE 9 VALUE ZERO .
  77 M PICTURE 9 VALUE ZERO .
  77 J PICTURE 9 VALUE ZERO .
  77 K PICTURE 9 VALUE ZERO .
* YEAR2000 N-YY IS INFECTED
  77 N-YY PICTURE 99 VALUE ZERO .
* YEAR2000 LEAPS IS INFECTED
...
* YEAR2000 NOOUT IS INFECTED
  01 NOOUT PICTURE S9 ( 8 ) COMP .
 PROCEDURE DIVISION USING OLD-DATE , NOOUT .
 BEEGOOD .
* YEAR2000 DATE ACCESS USING THE FOLLOWING INFECTED DATA ITEMS
* YEAR2000 N-YY LEAPS
  MOVE 0 TO I I-M M J K N-YY LEAPS .
* YEAR2000 DATE ACCESS USING THE FOLLOWING INFECTED DATA ITEMS
* YEAR2000 N-LEAPS DEC-CHEC FEB-1 FEB-2
  MOVE 0 TO N-LEAPS DEC-CHEC FEB-1 FEB-2 .
* YEAR2000 DATE ACCESS USING THE FOLLOWING INFECTED DATA ITEMS
* YEAR2000 FEB-C N-DATE
  MOVE 0 TO SLASH-CT FEB-C N-DATE .
* YEAR2000 DATE ACCESS USING THE FOLLOWING INFECTED DATA ITEMS
* YEAR2000 DT-ARR OLD-DATE
  MOVE OLD-DATE TO DT-ARR .
* YEAR2000 DATE ACCESS USING THE FOLLOWING INFECTED DATA ITEMS
* YEAR2000 DT-ARR
  INSPECT DT-ARR REPLACING ALL SPACES BY "/" .
* YEAR2000 DATE COMPARISON USING THE FOLLOWING INFECTED DATA ITEMS
* YEAR2000 D-V
  IF D-V ( 1 ) = "/"
* YEAR2000 DATE ACCESS USING THE FOLLOWING INFECTED
* DATA ITEMS YEAR2000 NOOUT
    MOVE -1 TO NOOUT
    GO TO R-EX.
* YEAR2000 DATE ACCESS USING THE FOLLOWING INFECTED DATA ITEMS
* YEAR2000 DT-ARR
  INSPECT DT-ARR TALLYING SLASH-CT FOR ALL "/" .
* YEAR2000 DATE COMPARISON USING THE FOLLOWING INFECTED DATA ITEMS
* YEAR2000 D-V
  IF D-V ( 8 ) = "/"
  GO TO A10 .
  IF SLASH-CT NOT = 2
  MOVE 0 TO SLASH-CT
* YEAR2000 DATE ACCESS USING THE FOLLOWING INFECTED DATA ITEMS
* YEAR2000 NOOUT
  MOVE - 1 TO NOOUT
  GO TO R-EX .
  ...
```

An additional tool processed such comments to produce summary reports detailing program size, number of definitions processed, number of dates identified etc.

6.2 Automated Conversion

A prototype conversion tool used the annotated source program outlined above to automatically convert infected lines of code to work with four digit year fields. The transformation rules for the conversion are similar in form to the one shown above. Most of the changes are in the data declarations and the occasional statement. However, programmers have developed many obscure date based algorithms which are not easy to recognize. Thus a first version of the

conversion tool considered only the obvious cases. In order to identify the more obscure cases the tool would need to be used on several projects with the transformation set being expanded to recognize each new date usage case. A comprehensive set of conversion transformations would evolve. However, it was envisaged that there would be a number of cases of date usage which it would be better to leave an expert to correct manually. These cases would be marked.

This conversion part of the year 2000 tool was never completed because of the difficulty in finding a commercial partner.

7 Conclusions

In this paper we have described the uses of TAMPR transformation systems and documented the effective use of transformation in an industrial setting. Yet, we would not claim that transformation is widely accepted in industry or that transformation is recognized as a viable technology. A few companies have been able to use transformation in niche industrial applications but rarely does transformation feature in their sales information.

We will give a number of broad comments from our industrial experience.

- It is difficult to convince organizations of the value of transformation and there is a need for additional commercial demonstrations to prove the maturity of the technology.
- Transformation are difficult to write and require significant ability and we have not found many companies that will invest the time and effort in staff training.
- Organizations want to acquire ready-made solutions that address particular company needs. It is our experience that we could sell the TAMPR transformation engine and a black-box collection of transformations—say as a reverse engineering tool, for example—but not transformation as a technology.
- Organization are unwilling to use transformation because it does not fit easily within a traditional software development environment and software development process.

References

[1] Boyle J.M., Harmer T.J. and Winter V.W. (1996). The TAMPR Program Transformation System, Proceedings of the Durham Transformation Workshop, 1-2 April.

[2] Boyle J.M. and Muralidharan M.N (1984). Program re-usability through program transformation, IEEE Trans. Software Eng., 10:574–88.

[3] James M. Boyle, *A Transformational Component for Programming Language Grammar*, Report ANL-7690, Argonne National Laboratory, Argonne, Illinois, July 1970.

[4] James M. Boyle, *Abstract Programming and Program Transformation—An Approach to Reusing Programs*, Software Reusability, Volume I, Ted J. Biggerstaff and Alan J. Perlis, eds., ACM Press (Addison-Wesley Publishing Company), New York, 1989, pp. 361–413.

[5] James M. Boyle, *Automatic, Self-adaptive Control of Unfold Transformations*, PROCOMET '94, IFIP Working Conference on Programming Concepts, Methods and Calculi, San Miniato, Italy, June 6-10, 1994, North-Holland/Elsevier, 1994, pp. 83–103.

[6] James M. Boyle, Maurice Clint, Stephen Fitzpatrick, and Terence J. Harmer, *The Construction of Numerical Mathematical Software for the AMT DAP by Program Transformation*, Parallel Processing: CONPAR 92—VAPP V, Second Joint International Conference on Vector and Parallel Processing, Lyon, France, 1-4 September 1992, L. Bouge, M. Cosnard, Y. Robert and D. Trystam, Eds., LNCS 634, Springer-Verlag, Berlin 1992, pp. 761–767.

[7] James M. Boyle, Kenneth W. Dritz, Monagur M. Muralidharan, and Roy Taylor, *Deriving Sequential and Parallel Programs from Pure LISP Specifications by Program Transformation* , Program Specification and Transformation, Proceedings of the IFIP TC2/WG2.1 Working Conference on Program Specification and Transformation, Bad-Tölz, Germany, 15-17 April, 1986, L.G.L.T. Meertens, ed., North-Holland, Amsterdam, 1987, pp. 1–19.

[8] James M. Boyle and Terence J. Harmer, *Functional Specifications for Mathematical Computations*, Constructing Programs from Specifications, Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs from Specifications, Pacific Grove, CA, USA, 13-16 May, 1991, B. Möller, Ed., North-Holland, Amsterdam, 1991, pp. 205–224.

[9] James M. Boyle and Terence J. Harmer, *A Practical Functional Program for the CRAY X-MP*, Journal of Functional Programming, Vol. 2, No. 1, Jan. 1992, pp. 81–126.

[10] Noam Chomsky, *Three Models for the Description of Language*, IRE Transactions on Information Theory, Vol. IT-2, 1956, 113–124; Reprinted (with corrections) in R. D. Luce, R. Bush, E. Galanter eds., *Readings in Mathematical Psychology, Vol. II*, John Wiley and Sons, New York, 1965.

[11] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart, *LINPACK User's Guide*, SIAM (Society of Industrial and Applied Mathematics), 1979, Philadelphia, PA.

[12] Stephen Boyle, *The Specification of Array-Based Algorithms and the Automated Derivation of Parallel Implementations through Program Transformation*. Phd thesis, The Queen's University of Belfast, 1994.

[13] Stephen Fitzpatrick, Terence J. Harmer, and James M. Boyle, *Efficient Implementations of General Sparse Matrices on Parallel Computers*, Proceedings of CONPAR 94—VAPP VI, Linz, Austria, 5-9 September 1994, Buchberger, B. and Volkert, J. Eds., LNCS 854, Springer Verlag, pp. 148–159.

[14] Stephen Fitzpatrick, Terence J. Harmer, Alan Stewart, Maurice Clint, and James M. Boyle, *The Automated Transformation of Abstract Specifications of Numerical Algorithms into Efficient Array Processor Implementations*, to appear in Science of Computer Programming.

[15] Victor L. Winter and James M. Boyle, *Proving Refinement Transformations Using Extended Denotational Semantics*, Proceedings of the Durham Program Transformation Workshop, Durham, U.K., 1-2 April 1996.

[16] Winter, V.W. and Boyle, J.M. (1996). Proving Refinement Transformations for Deriving High-Assurance Software, Proceedings of the IEEE High-Assurance Systems Engineering Workshop, Niagara on the Lake, Ontario, Canada, Oct. 22, 1996.