The Use of Transformations in "The Maintainer's Assistant"

M. Ward, F.W. Calliss and M. Munro

Centre for Software Maintenance
School of Engineering and Applied Science
University of Durham
Durham, DH1 3LE
England

Abstract

The Maintainer's Assistant is a code analysis tool aimed at helping the maintenance programmer to understand and modify a given program. Program transformation techniques are employed by the Maintainer's Assistant both to derive a specification from a section of code and to transform a section of code into a logically equivalent form. The general structure of the tool is described and two examples of the application of program transformations are given.

1 Introduction

Software maintenance is the most costly stage of the software lifecycle, often consuming 50%–90% of a project's total budget. The work of maintenance programmers is hampered by their lack of tools. Much of their work involves code analysis of one form or another, whether it is finding an obscure bug, attempting to understand a piece of code prior to modification or enhancement, or analysing the possible effects of a modification to the code. Some work has been done in this area with the development of such tools as: interactive cross referencers [17], data flow analysers [6, 19, 22], call graph generators [21] and redocumentation tools [11, 12]. All of these tools help the maintenance programmer in reducing the amount of time spent on code analysis activities, but the task of understanding the program, i.e. determining what *it does*, remains the domain of the maintenance programmer. The Maintainer's Assistant project [7] was conceived in order to address this problem.

In his DPhil Thesis [23] M. Ward presents a theory of program equivalence and a program transformation technique based on the infinitary logic language $L_{\omega_1 \omega}$ [3]. These transformations have been shown capable of deriving a specification for a given program [24] as well as the more conventional program manipulation activities associated with program transformations [8].

The method used to prove the transformations in [23] is based on "weakest preconditions", of Dijkstra [9]. For a given program S and condition on the final state R the weakest precondition $\mathsf{WP(S,R)}$ is the weakest condition on the initial state such that the program will terminate in a state satisfying condition R. It is possible to express the weakest precondition of any program or specification as a single formula in infinitary logic (an extension of first order logic which allows infinitely long formulae). It is shown in [23] that two programs S_1 and S_2 are equivalent if and only if the corresponding weakest preconditions $\mathsf{WP}(\mathsf{S}_1,\mathsf{R})$ and $\mathsf{WP}(\mathsf{S}_2,\mathsf{R})$ are equivalent formulas for any formula R. This reduces the problem of proving two programs to be equivalent to a problem in logic to which all the standard methods and techniques of theorem proving can be applied. The success of this technique may be judged by the large number of useful transformations which have been proved and the wide range of problems to which they can be applied.

Program transformations are used in program development [4, 5, 13, 14]; however their methods cannot cope with general specifications or with transforming programs into specifications. The system employed by the Maintainer's Assistant uses a small "kernel language" that has a simple mathematical semantics, associating a function with each program. This function maps each allowed initial state to the set of possible final states. Two programs are said to be equivalent if their associated functions are identical. The kernel consists of a few programming constructs; other constructs are added by defining them in terms of the kernel. In this way a complete programming language is built up.

1.1 The Proof Method

As an example of the proof method, the simple transformation known as assignment merging will be proved. Suppose we have the sequence: x:=x*2; x:=x+1 where we want to merge the two assignments. The weakest precondition is:

$$\begin{aligned} \mathsf{WP}(x{:=}x*2;\ x{:=}x+1,\ R) &\Leftrightarrow \mathsf{WP}(x{:=}x*2,\ \mathsf{WP}(x{:=}x+1,\ R)) \\ &\Leftrightarrow \mathsf{WP}(x{:=}x*2,\ R[x+1/x]) \end{aligned}$$

where $R[x+1/x]$ is R with x replaced by $x+1$.

$$\begin{aligned} &\Leftrightarrow R[x+1/x][x*2/x] \\ &\Leftrightarrow R[(x*2)+1/x] \\ &\Leftrightarrow \mathsf{WP}(x{:=}(x*2)+1,\ R) \end{aligned}$$

This has shown that x:=x*2; x:=x+1 is equivalent to x:=x*2+1.

An example of the application of program transformations to a program maintenance problem can be found in [24] which analyses a published program (from [10]) written in such a way that the structure and effect of the program are very hard to discern. Various transformations are applied to the program in order to reveal its structure and enable its effect to be summarised as a specification.

2 An Overview of The Maintainer's Assistant

Figure 1 shows a diagrammatic representation of the Maintainer's Assistant. The boxes labelled "Front Ends" and "Source Code Representation" are not being discussed in this paper. The representation of the code in the Maintainer's Assistant is the actual code plus some extra information that is needed by the transformations. The front ends are used to generate this extra information. The interested reader is referred to [23, 24] for a detailed account of what extra information is needed. This paper will concentrate of the *use* of these transformation in the Maintainer's Assistant.

Using the Maintainer's Assistant a maintenance programmer can do one of two things with a piece of code: examine it or modify it. All modifications of the program are performed via the structure editor, while all examinations of the code are done via the browser. The following subsections describe the different ways in which the Maintainer's Assistant allows a maintenance programmer to view and manipulate a program.

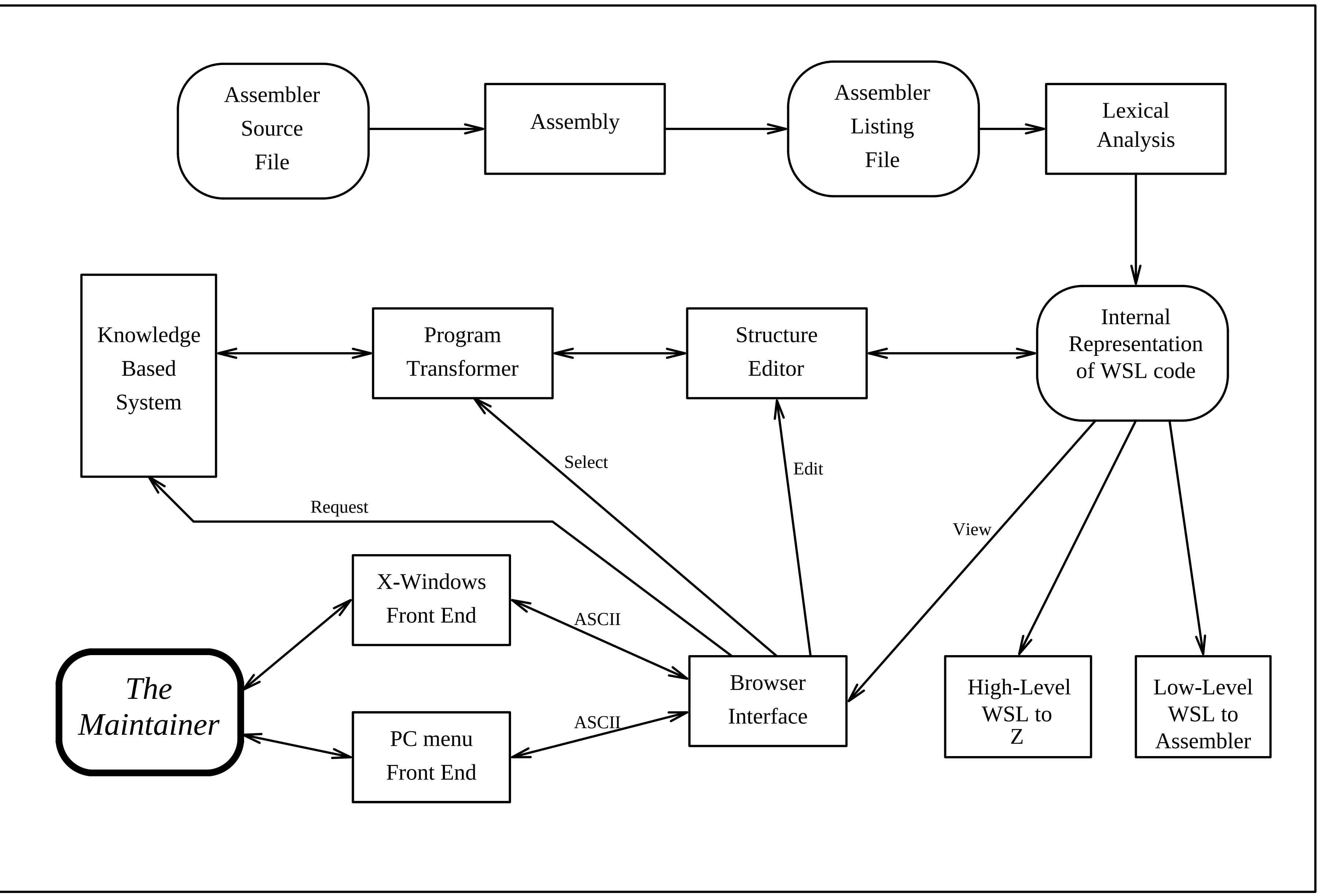

Figure 1: The Maintainer's Assistant

2.1 The Browser

The browser is used by the maintenance programmer to read the program; the output of the browser is displayed on a VDU. In its simplest form the browser displays the program text, but it can also be used to help the maintenance programmer understand that piece of text. This is done by the programmer sending the browser a scaling instruction, which is a command either to replace some program text by its specification (a 'contract' instruction), or to replace a specification with the appropriate program text (an 'expand' instruction).

When maintenance programmers read a piece of text, it is common for them to misread it (i.e. without noticing the absence (or presence) of some statements, e.g. a missing **return** statement from a function), and this can have serious repercussions.

Ward has shown developed transformations that are capable of deriving the specification for a piece of program text [24]. The browser automates part of this transformation process; hence the maintenance programmer can be sure that he is looking at an up–to–date specification as it is generated from the actual code. The theory of program equivalence presented in [23] ensures that the specification received by the maintenance programmer is an accurate portrayal of the code that it represents.

The browser is the only means by which a maintenance programmer can view a program with the Maintainer's Assistant.

The Maintainer's Assistant allows a maintenance programmer to manipulate the source code in three ways:

- by directly modifying the source using editing commands,
- by selecting a particular transformation from the library of transformations,
- by requesting the IKBS to search for a sequence of transformations which will achieve a given effect.

The "editing commands" form of program manipulation is the only method that allows a program to be transformed into a program that is not logically equivalent to the original, whereas the other two methods transform a program into a structurally different, but semantically equivalent program. The following three subsections describe the different program manipulation methods, starting with the IKBS.

2.2 The Knowledge–Base

The maintenance programmer sends 'request instructions' to the knowledge–base to transform a given piece of program text into a logically equivalent piece. The program is analysed by the Maintainer's Assistant and a suitable set of transformations are automatically derived. 'Select transformation' instructions are then sent to the transformation library where the appropriate transformation is performed on the program text.

2.3 The Transformation Library

Some transformations are too complicated to be derived automatically. In these circumstances, the maintenance programmer can explicitly select suitable transformations. This is done by sending a 'select transformation' to the transformation library.

When the transformation library receives a 'select transformation' instruction the appropriate transformation is recalled from the library together with the "applicability conditions" (the conditions on the program which must be satisfied for the transformation to guarantee equivalence). These conditions may range from trivial restrictions on the type of statement (e.g. a transformation that can only be applied to an if statement) to more complex requirements. Some applicability conditions may not be automatically derivable; for example, they could depend on the structure of the data to which the program will be applied for instance. The system will then ask the programmer for an "assurance" that the condition does indeed hold in this case, the correctness of the transformation will then be dependent on the correctness of this assertion and this fact will be recorded in the documentation accompanying the program. Once the system is satisfied (or has been assured) that the transformation is applicable, the transformation is applied to the selected piece of the program text by sending a sequence of edit commands to the structure editor.

2.4 The Structure Editor

The Structure Editor closely resembles a normal editor and is the only means that the Maintainer's Assistant provides for manipulating the source code's internal representation. Commands are entered and the program text is modified in accordance with these instructions. The structure editor can receive edit commands directly from the maintenance programmer or from the transformation library.

The structure editor is used to supply the transformation library with details about the source code, a prerequisite for the application of transformations.

3 Examples of the use of Transformations

3.1 Loop Inversion

In addition to the usual while loops we will be using loops of the form: do S od where S is the body of the loop. When a statement exit(n) is encountered (where n is an integer constant) then the n^{th} do-loop enclosing the statement will be terminated with execution continuing after the loop. exit(1) is written exit and exit(0) is equivalent to skip, i.e. a null statement. Such loops have been used in program transformations in [1] and [2]. Peterson et. al. [20] show how these statements can be used to remove goto statements from a program without increasing the size of the text or introducing flag variables or extra tests.

A "proper sequence" is defined as a sequence of statements which contains no exit statements which would leave a loop enclosing the sequence. Suppose S_1 is a proper sequence and S_2 is any statement. Then the following programs are equivalent:

```
do S1;
   S2 od
```

```
S1;
do S2;
   S1 od
```

This is often described as "moving the statement S_1 to the beginning/end of the loop": it is a special case of the "loop inversion" transformation proved in [23].

This transformation may be invoked in the Maintainer's Assistant by selecting the loop and sending the appropriate command to the transformation library. However it can also be invoked by requesting the IKBS to find a sequence of transformations which achieve a given effect. The "effect" required can be indicated in two other ways by means of editing commands:

- Selecting S_1 and moving it outside the loop.
- Selecting S_1 and moving it to the end of the loop.

In either case the transformation library will check that the appropriate applicability conditions hold (in this case that S_1 is indeed a proper sequence).

This transformation can be used in the reverse direction for removing duplicate copies of a sequence and in the forward direction for converting a loop of the form:

```
do S1;
   if B then exit fi;
   S2 od
```

where S_1 and S_2 are proper sequences to the while loop:

```
S1;
while ~B do
    S2;
    S1 od
```

3.2 Exponentiation Program

For the final example we show how the development and improvement of a simple algorithm can be carried out entirely using program transformations. The algorithm is for positive integer exponentiation with the following specification:

PROG $\equiv$ $z{:=}x^n$.

This specification is implemented using a loop with invariant $P \equiv z.x^n = x_0^{n_0}$ (where x_0 and n_0 are the initial values of x and n respectively) and variant n. The terminating condition is $n=0$ when we have $z.1 = x_0^{n_0}$ as required. The loop takes the form:

```
z:=1; { P is now set up }
while n≠0 do
    { Reduce n while maintaining P } od
```

Using the fact that $x^n = x.x^{n-1}$ we get the following implementation:

```
z:=1;
while n≠0 do
    ⟨z,n⟩:=⟨x.z, n-1⟩ od
```

One way to make this more efficient is to reduce n by more than one in the body of the loop. For example, if n is even we can use the fact that $x^n = x^{2(n/2)}$ and square x and halve n when n is even:

```
z:=1;
while n≠0 do
    if even(n) then ⟨x,n⟩:=⟨x²,n/2⟩
               else ⟨z,n⟩:=⟨x.z, n-1⟩ fi od
```

Now apply the transformation "entire loop unrolling" (from [23]) to insert a (modified) copy of the loop after the statement ⟨x,n⟩:=⟨x^2,n/2⟩:

```
z:=1;
while n≠0 do
    if even(n) then ⟨x,n⟩:=⟨x²,n/2⟩;
                    while even(n) ∧ n≠0 do
                        if even(n) then ⟨x,n⟩:=⟨x²,n/2⟩;
                                   else ⟨z,n⟩:=⟨x.z, n-1⟩ fi od
               else ⟨z,n⟩:=⟨x.z, n-1⟩ fi od
```

This can be simplified (since even(n) is true within the loop) to give:

```
z:=1;
while n≠0 do
    if even(n) then ⟨x,n⟩:=⟨x²,n/2⟩;
                    while even(n) ∧ n≠0 do
                        ⟨x,n⟩:=⟨x²,n/2⟩ od
               else ⟨z,n⟩:=⟨x.z, n-1⟩ fi od
```

Since n>0 holds within the body of the outer loop and is preserved by the body of the inner loop it must be invariant over the inner loop. Thus the test n≠0 can be removed from the condition on the inner loop. Next, apply "loop unrolling" (from [23]) to insert a copy of the *body* of the outer loop after the inner loop. On termination of the inner loop we must have odd(n) (since we cannot have n=0) so we can simplify the inserted statement to get:

```
z:=1;
while n≠0 do
    if even(n) then ⟨x,n⟩:=⟨x²,n/2⟩;
                    while even(n) do
                        ⟨x,n⟩:=⟨x²,n/2⟩ od;
                    ⟨z,n⟩:=⟨x.z, n-1⟩
               else ⟨z,n⟩:=⟨x.z, n-1⟩ fi od
```

We can "roll up" the statement before the inner loop (the converse transformation to loop unrolling in [23]) and simplify to get the final version:

```
z:=1;
while n≠0 do
    while even(n) do
        ⟨x,n⟩:=⟨x²,n/2⟩ od;
    ⟨z,n⟩:=⟨z.x,n-1⟩ od
```

4 Current Status

The first prototype of the tool is currently being implemented on an IBM RT 6150 workstation. The core of the tool is being implemented in Common Lisp, the user communicates with this core via an X-Windows WIMP interface. A front end for Assembler language is under construction: this will read in an assembler listing file and translate it into low-level WSL (Wide-Spectrum Language) code. WSL is a language we have developed which covers the whole spectrum of operations, from the low-level operations used in Assembler code, to high-level abstract specifications. The language has been designed to have a simple semantics and be easy to manipulate. The aim is to use transformations to turn the low-level WSL code into equivalent high-level code and specifications which will be easier to understand and maintain.

Much of the structure editor and parts of the program transformer have already been implemented, it is hoped that ultimately most of the system will itself be written in WSL and that therefore the system can be used to maintain its own source code.

5 Conclusion

Proponents of program transformation systems have seen their value largely as an aid to program development [5], however the Ward theory of program transformations [23] has proved valuable in attacking the problems faced by maintenance programmers as well as development programmers. This has led to the development of the Maintainer's Assistant as an interactive system for maintaining programs which is based on program transformations. The transformations are used to derive the specification of a section of a program, to present the program in different but equivalent forms as an aid to program analysis, and for general restructuring functions. The Maintainers Assistant also includes powerful editing functions and a mechanism for recording the editing history.

6 Acknowledgements

The authors would like to express their appreciation to Barry Cornelius and David Bittlestone for helpful comments made about an earlier draft of this paper.

This work was supported by a grant from the Alvey Directorate. F.W. Calliss is supported by a Science and Engineering Research Council Studentship.

References

[1] Arsac, J., “Syntactic Source to Source Program Transformations and Program Manipulation”, *Communications of the ACM*, vol. 22, no. 1, pp. 43–54, January 1979.

[2] Arsac, J., “Transformation of Recursive Procedures”, in [18], pp .211–265, 1982.

[3] Back, R.J.R., *Correctness Preserving Program Refinements*, Mathematical Centre Tracts 131, Mathematisch Centrum, 1980.

[4] Bauer, F.L., “Programming as an Evolutionary Process”, in [15], pp. 153–182, 1976.

[5] Bauer, F.L., “Program Development By Stepwise Transformations – the Project CIP”, in [16], pp. 237–266, 1979.

[6] Calliss, F.W. and Cornelius, B.J., “Dynamic Data Flow Analysis of C Programs”, in *Proceedings of the Twenty First Hawaii International Conference on System Sciences – 1988*, Volume II, Software Track, IEEE Computer Society Press, pp. 518–523, January 1988.

[7] Calliss, F.W., Khalil, M.M., Munro, M. and Ward, M., “A Knowledge–Based System for Software Maintenance”, to appear in *Proceedings of the Conference on Software Maintenance – 1988.*

[8] Cornelius, B.J. and Kirby, G.H., “A Programming Technique for Recursive Procedures”,”, BIT vol. 16, 1976, pp. 125–132.

[9] Dijkstra, E.W., *A Discipline of Programming*, Prentice–Hall International, 1972.

[10] Fenton, M., *Developing in DataFlex, Book 2, Reports and other outputs*, B.E.M. Microsystems, 1986.

[11] Fletton, N.T. and Munro, M., “Redocumenting Software Systems using Hypertext Technology”, to appear in *Proceedings of the Conference on Software Maintenance – 1988.*

[12] Foster, J. and Munro, M., “A Documentation Method Based on Cross–Referencing”, in *Proceedings of the Conference on Software Maintenance – 1987*, IEEE Computer Society Press, pp. 181–185, September 1987.

[13] Griffiths, M., “Program Production by Successive Transformation”, in [15] pp. 125-152, 1979.

[14] Griffiths, M., “Development of the Schorr–Waite Algorithm”, in [16] pp. 464–471, 1979.

[15] Bauer, F.L. and Samelson, K., (Eds), *Language Hierarchies and Interfaces*, Lecture Notes in Computer Science, Volume 46, Springer–Verlag, 1976.

[16] Goos, G. and Hartmanis, H., (Eds), “Program Construction”, Lecture Notes in Computer Science, Volume 69., Springer–Verlag, 1979.

[17] Munro, M. and Robson, D.J., “An Interactive Cross Reference Tool for use in Software Maintenance”, in *Proceedings of the Twentieth Hawaii International Conference on System Sciences – 1987*, Volume II, Software Track, Western Periodicals Company, pp. 64–70, January 1987.

[18] Neel, D. (Ed), *Tools and Notations for Program Construction*, Cambridge University Press, 1982.

[19] Osterweil, L.J. and Fosdick, L.D., “DAVE – A Validation Error Detection and Documentation System for FORTRAN Programs”, *Software – Practice and Experience*, vol. 6, no. 4, pp. 473–486, October–December 1976.

[20] Peterson ,W.W., Kasami, T., and Tokura, N., "On the Capabilities of While, Repeat and Exit Statements", *Communications of the ACM*, vol. 16, no. 8, pp. 503–512, August 1973.

[21] Ryder, B.G., "Constructing the Call Graph of a Program", *IEEE Transactions on Software Engineering*, vol. SE–5, no. 3, pp. 216–226, May 1979.

[22] Ryder, B.G., "An Application of Static Program Analysis to Software Maintenance", in *Proceedings of the Twentieth Hawaii International Conference on System Sciences – 1987*, Volume II, Software Track, Western Periodicals Company, pp. 171–179, January 1987.

[23] Ward, M., *Proving Program Refinements and Transformations*, D.Phil Thesis, Oxford University, 1988.

[24] Ward, M., "Transforming a Program into a Specification", University of Durham, Computer Science Technical Report 88/1. Also being reviewed for publication.