A Knowledge–Based System for Software Maintenance

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

Center for Software Maintenance
University of Durham
Durham, England DH1 3LE

Abstract

A description of an Intelligent, Knowledge-Based maintenance tool, being developed by the Centre for Software Maintenance at the University of Durham is described. The tool is intended to help reduce the amount of time spent on analysing code. Code analysis is performed when a programmer is familiarising himself with a piece of code, and when the effects of a proposed modification of the code is being assessed.

1 The Problem

There has been much research in recent years on the problems of program and system development, but very little work has been done on the problems of maintaining developed programs. The early promises of researchers in formal methods and structured programming, that the maintenance problem would "simply disappear" as their methods are developed and applied, have not been realised for several fundamental reasons:

- There exists a large quantity of what might be termed as "geriatric code". This code has often been heavily modified to make it perform many more functions than was originally intended. This code represents a considerable investment on the part of the owner and cannot therefore simply be discarded and rewritten using modern techniques.
- For much of this software there is little or no documentation. The documentation that does exist is unlikely to have been maintained along with the code and is therefore useless.
- The original designers and coders may long since have moved on to pastures new.
- The requirements of the customer are constantly changing, and the environment under which the software is run is also changing (from adding another terminal to a time-sharing system to porting the whole system onto different hardware with a different operating system). This means that any large system will be constantly being modified to fit the changed requirements and environment as well as fixing bugs. Thus even the hypothetical "bug-free" program will require maintenance and even the most structured program can be reduced to a tangled mess by a series of unstructured modifications and patches.

Much of the work which has been directed at the problems of program maintenance has been directed towards "restructuring" unstructured programs. Restructuring is the process of taking a program text, shuffling the pieces of code around to produce a logically equivalent program which conforms to some given programming criteria. This only tackles one small part of the maintenance task and even then can present new problems [5]. By far the largest part of the maintenance programmers job involves program *analysis*–the study of the source code in order to try and work out what the program is supposed to do and which other pieces of code are affected and how. When a program modification is proposed then further analysis of the modified program is required to show that the modification has the desired effect without introducing further bugs. For many large programs the chance of introducing another bug can be surprisingly high.

2 The Research Project

In response to these problems, the Centre for Software Maintenance has organised a research project, funded by the UK Alvey research initiative, with the aim of building a program analysis tool for use during the maintenance of a large system written in C, given only the source code. This tool will aid the maintenance programmer in understanding an initially unfamiliar system in as short a time as possible. Understanding a system has several aspects:

- Being able to describe the specification of a piece of code in a clear and concise way.
- Discovering the flow of information through the program (especially into and out of those modules which are to be modified).
- Discovering the flow of control–this includes a call graph of the subroutines for each module of interest.
- Untangling the structure to reveal which pieces of code form the bodies of loops and discovering under what conditions they are executed.
- Determining the use and scope of each variable name. Some work has already been carried out in this area at Durham with the development of interactive cross referencers [6, 19].
- Using the above information to determine the effects of a proposed modification (for instance to determine which modules will be affected via the control and data flow from a modified module).

2.1 The Knowledge Bases

Much of the knowledge is based on the concept of program plans. These are the basic building blocks from which algorithms are constructed and are familiar cliches to most programmers. For example a mergesort algorithm can be viewed as a composition of a plan for recursion on a binary tree, a plan for splitting a sequence in two, a plan for sorting pairs of numbers, and a plan for merging sorted lists. There are also collections of specialised plans appropriate to each programming language. For example, scanning a file in COBOL, the use of formulas with side effects in C and the use of functions which return their error status as the result and their actual effect as a side effect in both C and PASCAL. One approach to program comprehension is to transform the program into a composition of recognised plans.

In figure 1, we see a schematic diagram of the proposed tool. The expert system is seen to take knowledge from several sources. We will discuss each of these in turn.

2.1.1 Maintenance Knowledge

This will be knowledge about how maintenance programmers do their work. This knowledge will have to be elicited from expert maintainers using knowledge elicitation techniques similar to those described by Letovsky and Soloway [16, 17, 18] in their work on delocalised plans.

This knowledge will provide the bulk of the system's heuristic knowledge that will dictate the weighting patterns on searches through the expert system.

2.1.2 Program Plans

When dealing with program plans we divide them into two different categories:

General Program Plans
This will consist of a small set of plans that show commonly occurring activities in computer programs. The danger with plan based tools is that the number of plans needed is so large, that for computing purposes it can be regarded as infinite. The plan library alone could take up all of the available on-line storage and still want more [22, 26].For this reason the proposed tool will make use of plans, but will not be dependent on them. Plans will be used to help guide a particular search through the knowledge base.

Program Class Knowledge
This form of knowledge combines a group of specialist plans, known as program class plans, with a specialised set of heuristic rules on how to use the plans.

The program class plans are a group of plans that are common to a particular type of program. This project is concerned with maintaining a C compiler, so the set of program class plans would be a set of plans that depict a set of activities that are typically found in a compiler.

In order to use this plans effectively a specialised set of heuristic rules needs to be developed. In order to develop this set of rules we will have to perform some knowledge elicitation work on expert compiler writers. We are fortunate, in that the Centre for Software Maintenance has four ex-compiler writers as members so we have easy access to the required experts.

2.1.3 Program Specific Knowledge

This includes the internal representation of source code together with knowledge gleaned from this using the code analysis tools already developed. The proposed tool will know what information is wanted in a given situation and so is able to restrict the amount of information that is displayed. The maintenance programmer will analyse the program by interrogating the tool's knowledge–base.

2.1.4 Documentation

The item labelled documentation tool is not part of the present project, but is an idea for a follow on project dealing with binding sections of documentation to sections of the code.

2.1.5 Other Tools

The internal representation is in a particularly suitable form for interpretive execution including debugging using dynamic data flow analysis (see for example [6]) and other tools. Adding such tools to the system could form the basis of another project.

2.2 Internal Representation

The precise representation of the program in the analyser is very important because we need to be able to extract and analyse the structure of the program and make simple transformations of the structure easily. For example, variable names and procedure calls will be linked to their definitions so that the call graph can be rapidly examined and general module interconnections deduced.

2.3 Functional Aspects

The system must be interactive and be able to give suggestions as to where the maintenance programmer should be directing his attention in the mass of code available. It should present the code in a way which is easy to read (this will involve automatic indentation, appropriate highlighting and local restructuring).

To analyse the "higher-level" structure of the program, (for instance, the modules which are too big to fit on the screen) we need a "summariser" which collapses a section of code into a high- level specification or statement. Also the ability to link a comment to a section of code which can then be replaced by the comment.

The system is envisaged as containing the following main parts:

1. A catalogue of proven transformations together with the conditions which need to be tested before they can be applied and hints as to the situations in which they will be useful.
2. An interactive "structure editor" which acts directly on the internal structure of the program to edit the program, check the conditions for a particular transformation and apply transformations automatically.
3. An "intelligent" system which searches for a suitable sequence of transformations to achieve a desired effect (eg "Can you move this piece of code to here?", "Can you eliminate the two copies of this statement?", "Please transform this recursive procedure into an equivalent iterative form").

3 Theoretical Foundation

The proposed system will be based on a formal system developed by Ward [23, 24] in which it is possible to prove that two versions of a program are equivalent. The formal system is independent of any particular programming language and allows the inclusion of arbitrary specifications as statements in a program. Hence it can be used to prove that a program is equivalent to a given specification.

Program transformations are used in program development in [3, 4, 10, 11]. However their methods cannot cope with general specifications or with transforming programs into specifications. Our system uses a small "kernel language" which consists of specification statement as the only primative statement together with sequential composition, the if statement, a nondeterministic choice statement, a while loop and simple recursive procedures. This language has a simple mathematical semantics which associates 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.

Also associated with any program is a logical formula which encapsulates the initial conditions under which the program is guaranteed to terminate in a state which satisfies a given final conditions. These formulae are called "weakest preconditions" in [7]. Their value lies in the fact proved in [2] for iterative programs and extended in [23] to recursive programs that two programs are equivalent if and only if they have equivalent weakest preconditions. Thus proving the transformation of a program to a different form amounts to proving the equivalence of two formulae—for which all the apparatus of mathematical logic is available.

Once a set of simple transformations has been proved they can be used to derive other transformations by composition. Also the language can be extended to include new programming constructs by means of "definitional transformations" which for each program using the new constructs give an equivalent program defined in terms of simpler constructs. We can then prove the equivalence of two programs using the new constructs by proving that their definitional transformations are equivalent. This method is used in [23] to introduce the exit from the middle of a loop, recursive procedures with parameters and local variables, functions and expressions with side effects.

These transformations are used in [24] which takes a published program (from [8]) which was written in such a way that the structure and effect of the program are very hard to discern. Various transformations to the program to reveal its structure and enable its effect to be summarised as a specification.

3.1 Preventive Maintenance

This theoretical underpinning gives us the confidence to apply transformations to a program which is *not* (as yet) understood, as is shown in [20, 25]. In general, maintenance programmers are reluctant to make large changes to the source code of any program they are maintaining—least of all one which has yet to be understood fully, preferring instead to make additions and patches. The result is that over a period of time programs become longer, more fragmented and more difficult to follow with their structure obscured by an accumulation of changes. This can eventually give rise to the feeling "Don't touch that code! It may never work again!!". Our system will not only halt this process of fragmentation but will actually *reverse* it; so that a program which has become obscured through poor programming standards coupled with accumulated modifications can be gradually "cleaned up" over a period of time starting with the most critical areas (these will be the sections of code which are most frequently modified). The system will also enable the piecemeal redocumentation of a program which has little or no documentation or for which the documentation is out of date. This process—called "preventive maintenance"—can remove the need for periodically scrapping a whole system and re-writing it from scratch: which seems to be inevitable under current maintenance approaches.

It is important to have the ability to make large scale changes in the structure of a program without affecting its function so that as the function changes the structure can change to reflect that function. Also transforming a section of code into an equivalent form is an important tool for program comprehension, as is replacing sections of code by equivalent single "specification" statements. We aim to consider programs much more as manipulable objects which exist in different forms and which do well-defined things.

The language independence means that many of the results we develop can be applied to programs written in different programming languages and also enables the transformation of a program written in one high-level programming language to a different one.

4 Conclusion

The era of the automatic maintainer is many years away. This project is confining itself to the more realistic goal of a maintainer's assistant. The proposed tool will concentrate in the area of code analysis, this is because code analysis is perceived as the most time consuming activity in maintenance, Petzold [1] indicated that in a breakdown of maintenance tasks code analysis was seen to take up 47% of the time.

The lack of success of plan based systems such as PROUST [14, 15] and the Programmer's Apprentice [21] has led us to consider different solutions to the problem of understanding code. From some preliminary studies, the use of program transformations seems to have great potential in the area of program comprehension as well as program development.

Acknowledgement

The authors would like to express their appreciation to Barry Cornelius for helpful comments made on earlier drafts 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] *First Software Maintenance Workshop Notes*, 8th-9th September 1987 eds. Munro, M. and Calliss, F.W. Centre for Software Maintenance, University of Durham, Durham.

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

[3] Bauer, F. L. "Programming as an Evolutionary Process", in [12], pp.153-182, 1976.

[4] Bauer, F.L. "Program Development By Stepwise Transformations - the Project CIP", in [13], pp.237-266 1979.

[5] Calliss, F.W., "Problems With Automatic Restructurerers", *SIGPLAN Notices*, vol. 23, no. 3, pp.13-21, March 1988.

[6] Calliss, F.W. and Cornelius, B.J., "Dynamic Data Flow Analysis of C Programs", in Proceedings of the 21st Hawaii International Conference on SS, IEEE Computer Society Press, 1988.

[7] Dijkstra, E. *A Discipline of Programming*, Prentice–Hall Int, New York 1972.

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

[9] Foster, J. and Munro, M., "A Documentation Method Based on Cross-Referencing", in Proceedings Conference on Software Maintenance-1987, Austin, Texas IEEE Computer Society Press, Washington D.C., pp.181-185, September 1987.

[10] Griffiths, M. "Program Production by Successive Transformation", in [12], pp.125-152 1979.

[11] Griffiths, M. "Development of the Schorr-Waite Algorithm", in [13], pp.464-471 1979.

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

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

[14] Johnson, W.L. and Soloway, E., "PROUST", *Byte*, vol.10, no.4, April 1985, pp.179-190.

[15] Johnson, W.L. and Soloway, E., "PROUST: Knowledge-Based Program Understanding", in Proceedings Conference on Software Maintenance-1985, IEEE Computer Society Press, Washington DC., November 1985, pp.369-380. Also in *Readings in Artificial Intelligence and Software Engineering.*

[16] Letovsky, S., "Cognitive Processes in Program Comprehension", in Proceedings of the Conference on Empirical Studies of Programmers, 1986.

[17] Letovsky, S. and Soloway, E., "Strategies for Documenting Delocalized Plans", in Proceedings of the Conference on Software Maintenance– 1985, IEEE Computer Society Press, Washington DC., pp.144-151, November 1985.

[18] Letovsky, S. and Soloway, E., "Delocalized Plans and Program Comprehension", *IEEE Software*, vol.3, no.3, pp.41-49, May 1986.

[19] Munro, M. and Robson, D., "An Interactive Cross Reference Tool for use in Software Maintenance", in Proceedings of the 20th Hawaii International Conference on System Sciences, Vol.II, Software, ed. Shriver, B.D., Western Periodicals Company, California. pp.64-70.

[20] Munro, M. and Ward, M., "Intelligent Program Analysis Tools for Maintaining Software", Alvey Directorate.

[21] Rich, C., “A Formal Representation for Plans in the Programmer’s Apprentice”, in Proceedings of the Seventh International Joint Conference on Artificial Intelligence, IJCAI, Vancouver, August 24^{th}-28^{th} 1981, pp.1044-1053. Also in *Readings in Artificial Intelligence and Software Engineering.*

[22] Seviora, R.E., “Knowledge-Based Program Debugging Systems”, *IEEE Software*, vol.4, no.3, pp.20-32, May 1987.

[23] Ward, M., *Proving Program Refinements and Transformations*, D.Phil Thesis, Oxford University, 1986.

[24] Ward, M., “Transforming a Program into a Specification”, Comnputer Science Technical Report, 88/1, University of Durham, England. Being reviewed for publication.

[25] Ward, M. Calliss, F.W. and Munro, M., “The Use of Transformations in “The Maintainer’s Assistant”” Comnputer Science Technical Report, 88/9, University of Durham, England. Being reviewed for publication.

[26] Waters, R.C., “The Programmer’s Apprentice: A Session with KBEmacs”, *IEEE Transactions on Software Engineering*, vol.11, no.11, pp.1296-1320, November 1981. Also in *Readings in Artificial Intelligence and Software Engineering.*