N. Williams−Preston

An Experiment in Reverse Engineering Using Algebraic Specifications

Research Report LSV−97−10, Nov. 1997

Laboratoire Spécification et Vérification

ENS CACHAN

CNRS CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

Ecole Normale Supérieure de Cachan
61, avenue du Président Wilson
94235 Cachan Cedex France

Research Report LSV-97-10, Lab. Spécification et Vérification, CNRS & ENS de Cachan, France, Nov. 1997

An Experiment in Reverse Engineering Using Algebraic Specifications*

Nicky Williams Preston
LSV (Laboratoire Spécification et Vérification) URA 2236 CNRS,
Ecole Normale Supérieure de Cachan,
61 Av. Pdt Wilson,
94235 CACHAN Cedex
France
Nicky.Williams-Preston@lsv.ens-cachan.fr

November 27, 1997

Abstract

We propose a methodological framework for reverse engineering using formal methods, in which at least two algebraic specifications are constructed: the low-level "implementation specification" of the source code must be abstracted to obtain the "requirements specification". We explain that, in our view, the most interesting abstraction step is to link a specialized function to a more general function by means of the appropriate pre-condition. We illustrate our approach by reverse engineering a Fortran procedure from a real program used in the control of the French very high voltage electrical power system. The method used consists of successive transformations of the flowchart of this procedure. It succeeds in revealing the abstract operation and abstract data type to be described in the requirements specification, as well as the condition which must be satisfied in order for the procedure to be a correct implementation of this specification.

*This work was financed by Electricité de France through the Contrat d'étude R32/1K2924/ER292, No. OT: R32L02

Extended Abstract

Much of the software currently in use at Electricité de France, though giving entirely satisfactory results, was written 15 or 20 years ago. However, the replacement of these existing systems by new software re-developed from scratch can be expensive, risky and time-consuming.

As a solution to this problem, we propose a methodological framework for reverse engineering Fortran source code using formal methods. The aim is a description of the software which is very precise but easier to understand than the source code itself and which brings out the aspects which are most important when studying potential re-use. In particular, we wish to discover any assumptions about the domain of application of particular pieces of source code.

We propose the construction of at least two algebraic specifications of the Fortran code. First of all, the low-level "implementation specification" is drafted. This must be sufficiently close to the source code to inspire complete confidence that it is a faithful representation. However, by virtue of its structure, by making explicit certain coding conventions, by representing convoluted control structures by equivalent ones which are easier to understand, the implementation specification should be easier to read and understand than the source code itself.

The "requirements specification" is then drafted from scratch to describe the code in a far more abstract way. It should be comprehensible to an electrical engineer who is not a software specialist.

The resulting formal description is precise and rigorous. Moreover, the implementation specification can be enriched to show how it corresponds to the requirements specification and it can then be formally proved that the implementation specification satisfies the requirements specification. This enrichment allows us to locate in the source code the implementation of any aspect of the requirements specification which may interest us. Any desired or suspected property of the specifications at any level can also be formally proved.

The construction of specifications at two levels which are then linked by a formal proof is in contrast to other approaches to reverse engineering, in which transformations are carried out on a formal low-level description in order to create the more abstract description. We use the algebraic specification formalism as a rigorous and expressive means to describe and prove what we have discovered in other ways about the code.

We propose to use software tools to restructure and analyze the source code in order to aid the construction of the implementation specification. However, we believe that some human input is essential in the reverse engineering of the trickier parts of Fortran code. It is in studying these parts of the code at least partially by hand that the reverse engineer acquires the understanding which enables him or her to draft a more abstract specification.

As an example, we take a Fortran procedure from a real program used in the control of the French very high voltage electrical power system. Successive transformations of the flowchart of this procedure are first carried out informally. These transformations help us understand that the procedure is in fact a highly

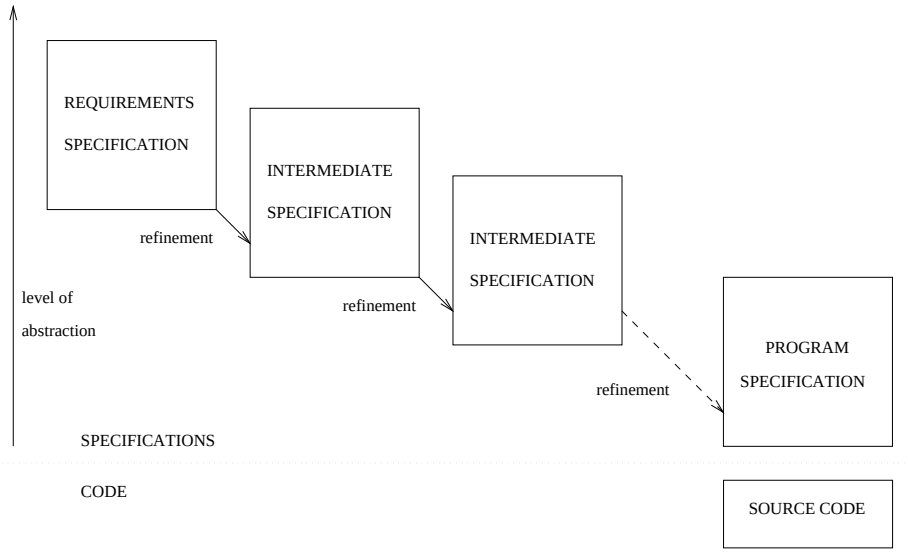
specialized implementation of a classic software operation and thus to draft the formal specifications at two levels. Our approach succeeds in highlighting the condition which the application domain must satisfy in order for the procedure to be a correct implementation of the more general operation.

1 Introduction

Much of the software currently in use at Electricité de France (EDF, the national French electricity generation and distribution company), though giving entirely satisfactory results, was written 15 or 20 years ago. As this software pre-dates the spread of rigorous software engineering practices, and as there have inevitably been changes in the personnel responsible for its maintenance, EDF's knowledge and understanding of some of its own software has become more and more partial. The replacement of existing systems by new software redeveloped from scratch can be expensive, hazardous and time-consuming. This is why it is increasingly felt to be worthwhile investing in better documentation and understanding of existing software.

This paper describes some results of a feasibility study of the reverse engineering of Fortran source code towards algebraic specifications. The aim is a description of the software which is very precise but easier to understand than the source code itself and which brings out the aspects which are most important when studying potential re-use. In particular, we wish to discover any assumptions about the domain of application of particular pieces of source code. These are likely to be implicit, dispersed within the code and not included in any documentation because they seemed so obvious at the time. However, they may no longer be validated by today's technology. Our aim is not to build up a repository of re-usable components but the adaptive re-use of whole systems or sub-systems.

Figure 1: The use of formal methods in software development



2 Formal methods

Algebraic specifications [11, 9] belong to the domain of formal methods, originally conceived for the development of new software. The use of formal methods in software development is illustrated in Figure 1. An initial specification is drafted at a high level of abstraction, we will call this the *requirements* specification: it is expressed in many-sorted first-order logic and so may not be executable. The software to implement this requirements specification is then designed in stepwise fashion by successively drafting new versions of the specification, at less and less abstract levels. Each new specification incorporates some design or implementation decision compared to the previous specification:

- if the latter was non-deterministic in some way, this might be a choice of one of the different possibilities so as to remove the non-determinism;
- or it might be a choice of algorithm;
- or a choice of data representation scheme;
- or the decision to stock an intermediate result so that it doesn't need to be re-calculated;
- etc.

Each new specification is proved to "satisfy" in some sense the specification at the preceding level, and thus, by transitivity, the initial specification. We say that each specification *refines* those at higher levels of abstraction. Refinement can take different forms [8]: one possibility is for every model (i.e. possible implementation) of the first specification to be a model of the second but not vice-versa. Another possibility is for the two specifications to be equivalent (i.e. have the same class of models) or the two may have what appears to be exactly the same *behavior*, i.e. when the *observable* results of any operation are compared, they are always the same.

The final specification, which we will call the *implementation* specification, is very nearly a program, but still expressed in the algebraic specification language. The software can be coded directly from the implementation specification.

We propose to reverse this process in order to carry out reverse engineering in a formal setting. First of all, the implementation specification is drafted. This must be sufficiently close to the source code to inspire complete confidence that the implementation specification is a faithful representation of the source code. It should also be possible to locate objects or operations in the source code by referring to the implementation specification. The implementation specification is thus a description of software entities, such as arrays, and operations on them. However, in our view the implementation specification must not necessarily be a simple transcription of the source code because it should also aid understanding of the source code. By virtue of its structure, by making explicit certain coding conventions, by representing convoluted control structures by equivalent

ones which are easier to understand, the implementation specification should be easier to read and understand than the source code itself.

After the implementation specification, a more abstract specification is written which the implementation specification refines. The process continues until the desired level of abstraction is reached. This is the requirements specification, which should describe electrical networks and properties and be comprehensible to an electrical engineer who is not a software specialist.

By using formal methods in this way, we obtain a very precise and rigorous description of the software whose correctness can be formally proved (apart from the final transition from implementation specification to code). The proofs depend on the links which exist between the specifications at different levels and which allow us to pinpoint in the code the implementation of any aspect of the requirements specification which may interest us. Any desired or suspected property of the specifications at any level can also be formally proved. Moreover, the process we describe provides a framework to guide reverse engineering.

3 Abstraction

How can we abstract the implementation specification?

We must recognize, among the low-level data structures and operations of the code, the abstract data types and the functions that will be described by more abstract specifications. Moreover, in writing the implementation specification, we describe the *effective* functionality of fragments of the source code. Initially these fragments will correspond to source code subroutines and functions. However, it is often the case that the *intended* functionality of a subprogram may only be found by considering it in the different contexts in which it can be called. When hypotheses expressing the conditions of application of the code fragment are added to the effective functionality, we obtain its intended functionality, i.e. its requirements specification. We believe that the most important and valuable abstraction step is to link a very specialized function to a more general and familiar or more easily understood function, by means of the appropriate pre-condition. Our aim is to aid the reverse engineer to find this step.

We believe that the abstraction process is essentially intuitive and depends on an understanding of the source code acquired during the manual part of implementation specification construction. The translation of source code to an algebraic implementation specification can certainly be partially automated and this possibility would be very useful for the most simple code, which makes up a large part of most real applications. However, the trickier parts of the code, loops in particular, will probably need to be studied manually before they can be abstracted, and it is not obvious that, at this low level of abstraction, automatically generated specifications are easier to understand than the (possibly cleaned-up) code itself. Automation implies taking into account all theoretically possible computer programs — it cannot take advantage of the rational approach of the authors of most real tested programs. This is why computer-assisted construction of the implementation specification seems to us to offer

Figure 2: The source code of PRCHEN

```
      SUBROUTINE PRCHEN(L,N1,N2,IPCO,ICCO,NRLE)

      COMMON/COMSOL/SPGM(2),KER,IBT,BF4L,B4TROP
      COMMON/DBUG/BS,B1,B2,B3,B4,B5,B6,B7,B8,
     +      B9,B10,B11,B12,B13,B14,B15
      COMMON/NUFIC/ KFB,KFR,KCR,KS

      DIMENSION IPCO(*),ICCO(*),NRLE(*)

      KCH = IPCO(N1)

      KP = 0

      IF (KCH.LE.0) GOTO 6

4     IF (NRLE(KCH).GE.N2) GOTO 3
        KP  = KCH
        KCH = ICCO(KCH)
        IF (KCH.GT.0) GOTO 4
3     IF (KP.LE.0) GOTO 6
        ICCO(KP) = L
7       ICCO(L) = KCH
      GOTO 1

6     IPCO(N1) = L
      GOTO 7

1     RETURN
      END
```

the right balance between efficiency and a chance to study the code.

We illustrate these ideas with the analysis of a procedure belonging to the Fortran77 program ESTIM, written in 1977 and in continual use for over 15 years. This program is the last link of the data acquisition chain for real-time control of the very high voltage electrical power system (although no interrupts or other such real-time features are present in the source code). Once the grossly erroneous measurements have been replaced by other programs in the chain, ESTIM carries out a more sensitive analysis, based on statistical methods, to detect and correct false measurements of power transits across each line of the electrical network.

4 An example

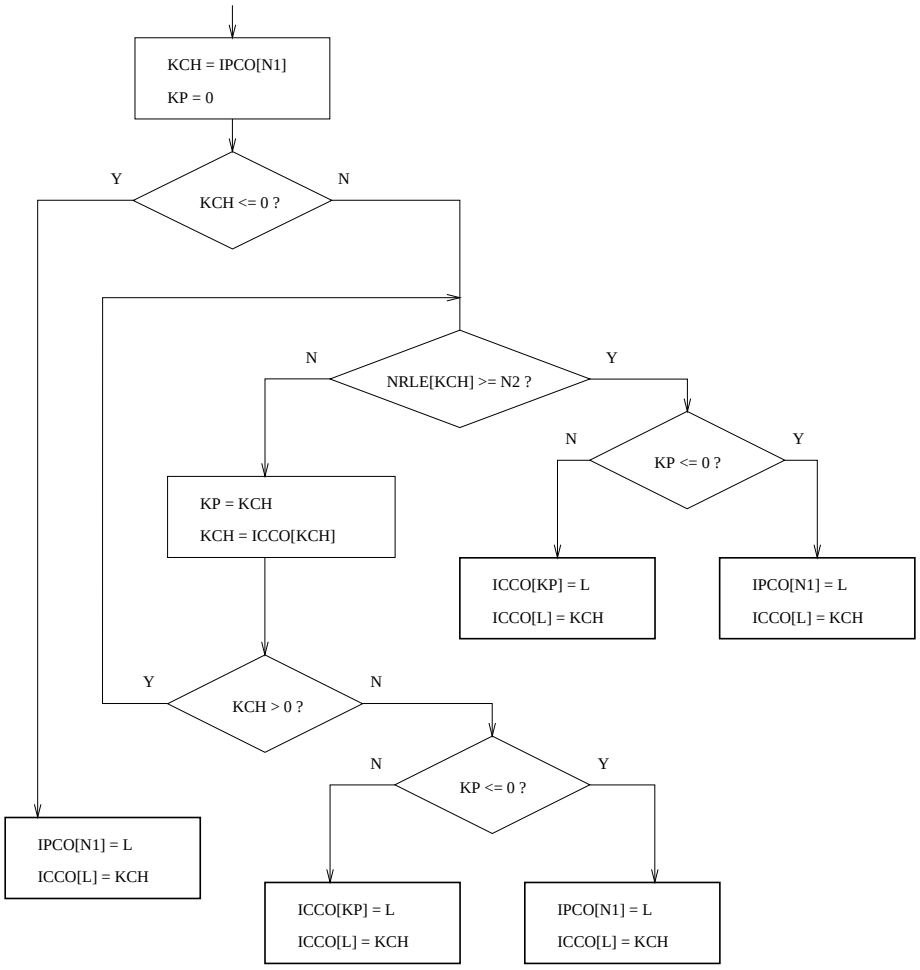
The procedure PRCHEN, shown in Figure 2, is an interesting subject for reverse engineering as, although it is short, it has a complicated structure with several *GOTO*s crossing each other. Although PRCHEN contains a few cryptic comments (not included in Figure 2), its functionality is not at all obvious at first sight and cannot be understood without also understanding the underlying data structure, whose role was completely unknown when we started this analysis.

4.1 Static analysis results

A preliminary step in the construction of the implementation specification is a static analysis of the source code and examination of such documentation and comments as exist. We do not have the space here to describe this process in detail but in the case of the PRCHEN procedure it provides the following information:

- The parameters of PRCHEN are L, N1, N2, IPCO, ICCO and NRLE, of which only IPCO and ICCO are modified.
- No global variables can be modified by PRCHEN.
- KCH and KP are local variables.
- PRCHEN is only called twice and as the two calls are "symmetrical" and independent of each other we can simplify this presentation by only treating one of them: the calling subroutine contains a *DO* loop in which the *DO* variable I is incremented from 1 to NS and PRCHEN is called with its formal parameter L instantiated to I. From comments elsewhere in the program, we know that NS is the number of lines in the network, so L successively takes the values of identification numbers of all the lines, in increasing order.
- The actual parameter corresponding to N1 is NRLO[L], and that corresponding to N2 is NRLE[L]. We know from the analysis of other parts of the program that the global array NRLO stocks for each line the node connected to its start and NRLE the node connected to its end, we will call these nodes the start-node and end-node respectively of the line.
- The actual parameter corresponding to IPCO is the one-dimensional global array IPCO of which we only know that its length is the number of nodes in the network.
- The actual parameter corresponding to ICCO is the one-dimensional global array ICCO of which we only know that its length is the number of lines in the network.
- The last formal parameter, NRLO, is instantiated by the global array NRLO described above.

Figure 3: The initial flowchart



- What is more, the global array IPCO is filled with 0's immediately before the start of the loop containing the call to PRCHEN and within this loop IPCO and ICCO are only modified by PRCHEN. In other words PRCHEN alone fills IPCO and ICCO: between two successive calls to PRCHEN as well as at the end of the calling loop any non-zero value in IPCO and ICCO has been written by PRCHEN.

Figure 4: The simplified flowchart

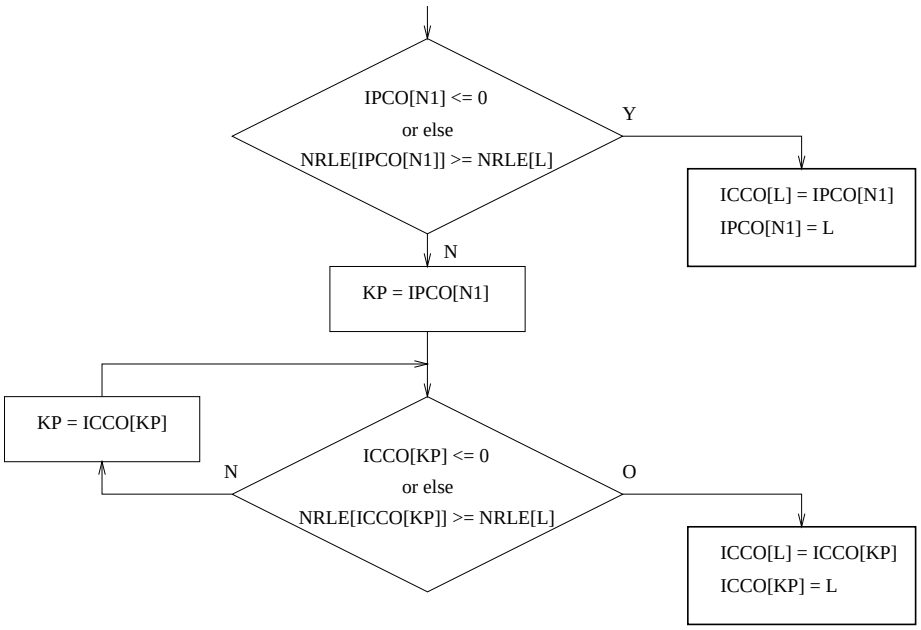
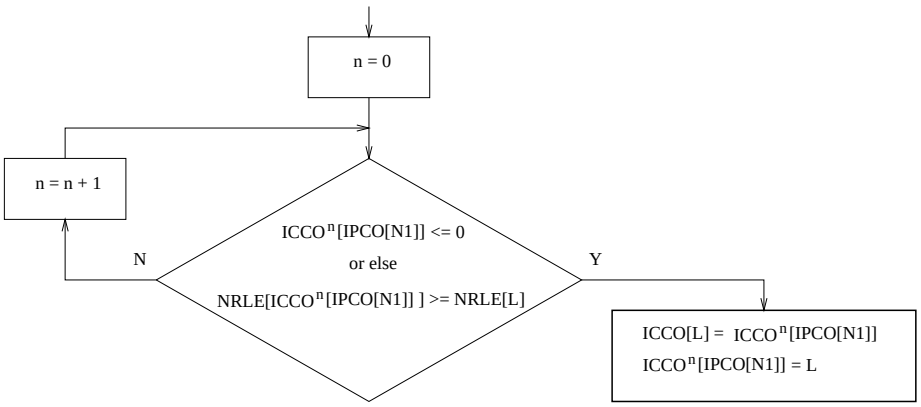


Figure 5: The final flowchart



4.2 Flowchart transformations

The approach adopted to aid axiomatization and comprehension of this subroutine is that of successive transformations of the flowchart. The first flowchart constructed is shown in Figure 3 (the boldly outlined boxes contain assignments immediately preceding exits from the subroutine). In fact, the axiomatization of PRCHEN could be based on this flowchart but the resulting axioms would be as badly structured and incomprehensible as the code. So we start to simplify the flowchart, manually applying transformations which preserve the semantics of the code:

- propagation of the results of assignments and tests;
- unfolding the first iteration of the loop;
- eliminating an unnecessary test and a local variable;
- re-folding the loop differently;
- grouping tests and exits.

Each successive transformation brings further understanding of the code which guides our choice of the next transformation.

These are all familiar transformations, whose correctness is evident and could be proved if necessary, but the automatic restructuring tools that we have tried out on this procedure (Toolpack[10] and FORESYS[1][4]) do not carry them all out. In fact, the sequence of transformations is relatively subtle: the unfolding of the first iteration of the loop is necessary in order to find the unnecessary test and the loop must then be re-folded differently.

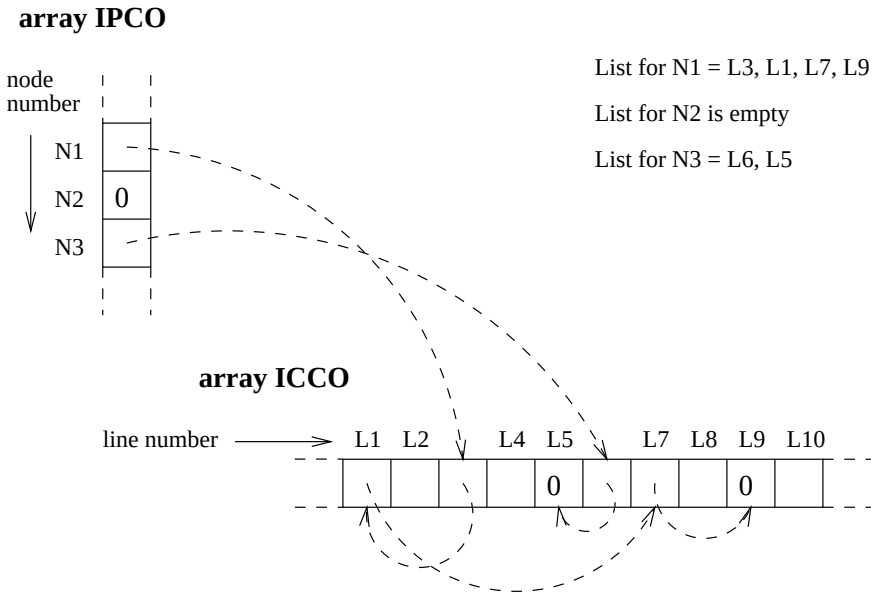
The simplified flowchart is shown in Figure 4. The axioms defining the functionality of PRCHEN in the implementation specification are based on this flowchart. They are a natural transcription of the flowchart into first-order logic, given pre-defined functions corresponding to the reading and assignment of array elements. The loop is represented by a function with a recursive definition.

However, we carry out one more flowchart transformation which is not standard and which amounts to finding a loop invariant. This simplification is not necessary for the specification of PRCHEN at the implementation level. Indeed the resulting flowchart is too far removed from the source code to be used for the implementation specification. However, this simplification enables us to completely understand the subroutine and associated data structure. The requirements specification is not a transcription of the final flowchart: it is the expression, drafted "from scratch", of our understanding of the subroutine's role, as illustrated in Figure 7. It can then be formally proved that this requirements specification is refined by the implementation specification.

So what is this final simplification? The flowchart in Figure 4 has two parts, each with a similar test and exit. To better understand the loop, and consequently the relationship between these two parts, we adopt a special notation.

[1]FORESYS is a trademark of CONNEXITE S.A.

Figure 6: The data structure

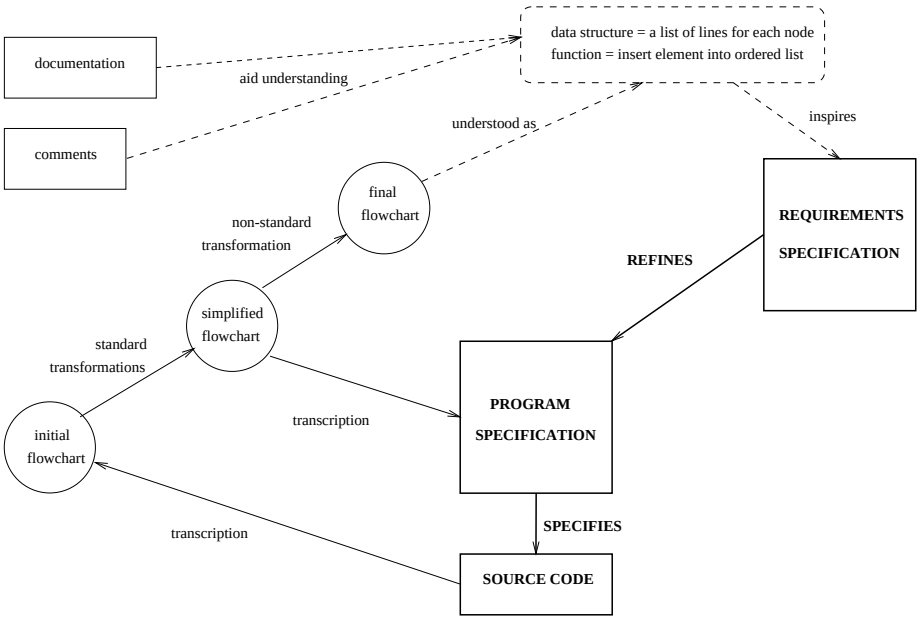


At each successive iteration of the loop, the local variable KP is set to ICCO[KP]. On entry to the loop KP is equal to IPCO[N1], after the first iteration it is set to ICCO[IPCO[N1]], and so on. We therefore define a pseudo-variable n which counts loop iterations and we represent the value of KP in the nth iteration of the loop by $ICCO^{n-1}[IPCO[N1]]$. By allowing IPCO[N1] by itself to be written $ICCO^0[IPCO[N1]]$, we obtain the flowchart of Figure 5.

4.3 The data structure represented by IPCO and ICCO

Examination of the double assignment on exit from the flowchart of Figure 5 reveals that if we call the value denoted by $ICCO^n[IPCO[N1]]$ before the assignments v, then after the assignments the value of $ICCO^n[IPCO[N1]]$ has become L and it is $ICCO^{n+1}[IPCO[N1]]$ which now has the value v. This recalls insertion of an element in the middle of a list and, pursuing this idea, we can see that the data structure is that shown in Figure 6 — for each node, a list is constructed of all the lines which have this node as their start-node. The list of line-numbers is stored in the form of a chained list with zero indicating the end of the list. The array IPCO stocks the beginning of the list for each node and the tails of all the lists are stored in the array ICCO.

Figure 7: The reverse engineering of PRCHEN



4.4 The functionality of PRCHEN

Examination of the two tests in Figure 5 reveals that PRCHEN inserts line L into the list of lines sharing the same start-node, its position in the list being determined by the node-number of its end-node. Each line precedes all those whose end-node has a higher node-number. The relative positions of parallel lines (i.e. connected at each end to the same nodes) is not determined by the code of PRCHEN, but by the order of insertion of the lines. As PRCHEN is in fact successively called for increasing line-numbers, in the case of parallel lines it is the line with the highest line-number which will come first in the list.

4.5 Abstraction

The transformations applied to the flowchart of PRCHEN helped us understand that the role of this subroutine is none other than the classic software function of insertion of an element in an ordered list. However, the implementation is highly specialized, depending not only on the order of insertion of the lines but also on the data structure represented by the arrays IPCO and ICCO. This data structure is a very economical solution to the problem of storing a set of lists of varying lengths, but it depends on the property that each line only appears

once, i.e. is only connected at its start end to one node. This may seem obvious but if we wanted to use this code on some new types of equipment, such as multi-poles (which in some respects can be treated in the same way as classic lines), this condition would no longer be respected.

Further analysis of the program shows that the IPCO/ICCO data structure is later used when calculations have to be carried out on all the lines with the same start-node, in the order in which they are stored in the lists.

We can imagine the most abstract level of specification being the requirement to carry out a calculation on all these lines in this order. At a more concrete level, the specification would explicitly require the construction of a data structure to store the ordered lists of lines. The next most concrete level would specify the representation of lines and nodes as non-zero numbers and use the property that each line only appears once to define the data structure as two one-dimensional arrays. Finally, the construction of the lists by insertion of each line number in turn can be specified and the most concrete specification would be a transcription of the source code.

5 Conclusion

We have not written all the different levels of specification described in the previous paragraph but we did construct specifications of PRCHEN and the calling subroutine at two levels.

The use of flowchart transformations in the reverse engineering of the PRCHEN procedure is illustrated in Figure 7.

The structure of the implementation specification is based on the Fortran data structures: one module describes the global arrays IPCO and ICCO and the operations on them, including the operation PRCHEN. The mere representation of the Fortran code is augmented at this level by an extension to the specification which defines some elementary operations on lists in terms of operations on the arrays IPCO and ICCO. This specification makes explicit various programming conventions, such as the use of the value 0 to indicate the end of the list.

The requirements specification consists of:

- the specification of lines and nodes of a network;
- a classic specification of insertion in an ordered list;
- the specification of the order on all the lines connected to the same start node;
- the specification of the condition that each line only appears once in the set of lists.

The drafting of these specifications and proof that the implementation specification respects the requirements specification makes interesting demands on the expressive power of algebraic specifications and the notion of refinement since an elegant solution would entail the use of sub-sorts [7].

This approach to reverse engineering has been successful in finding and expressing formally the generic function corresponding to the PRCHEN routine and the pre-condition which must be satisfied in order for PRCHEN to be a correct implementation of this function.

6 Related and future work

Maintainer's Assistant [2] is a reverse engineering tool which automates the application of transformations which have been proved to preserve the semantics of the program. Code containing *GOTO*s is first transformed into an action system. The user decides which transformations to apply (helped to a certain extent by the tool), so the approach is similar to that described here. However, the transformations are effected on code translated directly into a Wide Spectrum Language (WSL) with the aim of attaining a sufficiently abstract specification just through proven transformations so that the satisfaction of the requirements specification by the code does not have to be proved.

In the approach developed in the *REDO* project for small sections of code [1] the source code is first translated into a mathematical notation consisting just of equational definitions of functions, function composition and an *IF THEN ELSE* construct. Each *GOTO* is removed by creating a new function to define its continuation. These equations are then rewritten, with the aid of a set of guidelines, in order to simplify all the loops and re-express all the equations in a normal form. The detection of impossible combinations of *IF* branches is also discussed. The authors suggest that the transformation of source code into this normal form can be fully automated. They then propose to simplify this representation "by hand" to obtain a more abstract functional definition of the code fragment.

A similar approach to the previous one was described, in less detail, in [6] which addresses the generation of an abstract description of Cobol source code. The authors propose to start by automatically transforming the source code into structured form, which we feel could in some cases hamper understanding of the code. They recognize that the recursive definitions arising from the automatic treatment of loops may not aid understanding of the code, and suggest manual intervention to find loop invariants.

In [3] the generation of a specification of C source code using symbolic execution is proposed. Loop invariants, or the total number of iterations, must be found by the user and the symbolic execution tool then produces the path conditions and functional definition of each execution path. These are analyzed by hand to simplify them and find an abstract description.

Finally, *AUTOSPEC* [5] is a tool being developed to automate the generation of (low-level) formal specifications directly from C source code. The tool is based on the definition of the strongest postcondition semantics of each C language construct and the specifications are expressed as Hoare triples, with evaluation functions and a notation for textual substitution. Loops are given a specification using a function call exponent i whose value is unknown and which represents

the total number of iterations. The abstraction of the resulting specification is still being worked on.

Apart from the level of automation (and in a feasibility study such as ours, techniques must be tried out by hand first) the obvious difference between our approach and related work is that we do not carry out simplification and abstraction in the specification language. We feel that it may be easier for the user to spot possible transformations in a flowchart than in WSL code or mathematical equations, due to the flowchart's greater visual appeal, at least for the relatively short sections of code that are likely to be analyzed by hand. Moreover, the ideas of abstraction developed by others do not seem to encompass the notion we emphasize of linking one specialized function to another more general function, by means of the right pre-condition.

The continuing exploration of the approach described here, with particular investigation of

- tools and techniques to aid in the construction of the implementation specification;
- the application of the approach described here to other examples of source code;
- the development of a framework for the refinement of algebraic specifications with sub-sorts, which we can use in reverse engineering,

is now the subject of a Ph.D..

7 Acknowledgements

I would like to thank Jérôme Ryckbosch of EDF and Michel Bidoit of the Laboratoire Spécification et Vérification for all their help and support.

References

[1] P. Breuer, K. Lano, and J. Bowen. Understanding programs through formal methods. In H. van Zuylen, editor, *The REDO Compendium of Reverse Engineering for Software Maintenance*, pages 195–223. John Wiley & Sons, 1993.

[2] T. Bull. An introduction to the WSL program transformer. In *Proceedings of the Conference on Software Maintenance, San Diego, 1990*, 1990. updated version available at http://www.dur.ac.uk/CSM/.

[3] A. Cimitile, A. de Lucia, and M. Munro. Qualifying reusable functions using symbolic execution. In *Proc. 2nd Working Conference on Reverse Engineering, Toronto, Canada, 1995*, pages 178–187, Washington, DC, 1995. IEEE Comp. Soc. Press.

[4] For information on FORESYS see http://www.cais.net/s2i/www/general/foresys.html.

[5] G. C. Gannod and B. H. Cheng. Using informal and formal techniques for the reverse engineering of C programs. In *Proc. 3rd Working Conference on Reverse Engineering, Monterey, CA, 1996*, 1996.

[6] P. Hausler, M. Pleszkoch, R. Linger, and A. Hevner. Using function abstraction to understand program behaviour. *IEEE Software Special Issue Maintenance, Reverse Engineering and Design Recovery*, 7(1):55–63, Jan. 1990.

[7] P. D. Mosses. The use of sorts in algebraic specifications. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification, Proc. 8th Workshop on Specification of Abstract Data Types, Dourdan, France, 1991*, volume 655 of *Lecture Notes in Computer Science*, pages 66–91. Springer-Verlag, 1993.

[8] F. Orejas, M. Navarro, and A. Sanchez. Implementation and behavioural equivalence: A survey. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification, Proc. 8th Workshop on Specification of Abstract Data Types, Dourdan, France, 1991*, number 655 in Lecture Notes in Computer Science, pages 93–125, Berlin, 1993. Springer-Verlag.

[9] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 1997. To appear.

[10] Toolpack is free software obtainable at ftp://ftp.germany.eu.net/pub/utils/toolpack.

[11] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788, BW, 1990. Elseiver.