Formal Methods for Legacy Systems

M. P. Ward* K. H. Bennett*

January 6, 1995

Abstract

A method is described for obtaining useful information from legacy code. The approach uses formal proven program transformations, which preserve or refine the semantics of a construct while changing its form. The applicability of a transformation in a particular syntactic context is checked before application. By using an appropriate sequence of transformations, the extracted representation is guaranteed to be equivalent to the code. In this paper, we focus on the results of using this approach in the reverse engineering of medium scale, industrial software, written mostly in languages such as assembler and JOVIAL. Results from both benchmark algorithms and heavily modified, geriatric software are summarised. It is concluded that the approach is viable, for self contained code, and that useful design information may be extracted from legacy systems at economic cost. We conclude that formal methods have an important practical role in the reverse engineering process.

1 Introduction

Legacy software may be defined informally as "software we don't know what to do with, but it's still performing a useful job". The implication is that the preferred solution is to discard the software completely, and start again with a new system. This may not be appropriate in all cases, for example:

(i) The software represents years of accumulated experience, which is unrepresented elsewhere, so discarding the software will also discard this knowledge, however inconveniently it is represented;

(ii) The manual system which was replaced by the software no longer exists, so systems analysis must be undertaken on the software itself;

(iii) The software may actually work well, and its behaviour may be well understood. A new replacement system may perform much more badly, at least in the early days. Hence it may be worth recovering some of the good features of the legacy system;

(iv) A typical large legacy software system has many users, who typically have exploited undocumented "features" and side effects in the software. It may not be acceptable to demand that users undertake a substantial rewrite for no discernable benefit. Therefore, it may be important to retain the interfaces and exact functionality of the legacy code, both explicit and implicit;

(v) Users may prefer an evolutionary rather than a revolutionary approach.

The aims of the work described in this paper are:

(i) To help the expert maintainer in understanding a large legacy system;

(ii) To assist in representing that understanding, using a carefully and formally defined language;

(iii) To automate as far as possible mundane and mechanical tasks, leaving the maintainer to focus on the strategic steps;

*Department of Computer Science University of Durham, Durham, UK

(iv) To ensure that the ultimate representation represents the semantics of the source code exactly.

Our ultimate objective is to recover a formal requirements specification for a legacy system, given only the source code (written in a typical third or second generation language).

It should be stressed that the aim is not to *replace* the expert maintainer, or to "de-skill" the reverse engineering task, but to provide tool support to enhance and magnify human skills.

Given a source program $\mathbf{P}$, the approach is firstly to translate this into an equivalent form $\mathbf{P}'$ expressed in a language WSL, in which all subsequent operations are performed. This wide spectrum language is a key part of our work, as it must facilitate the representation of both low level imperative constructs (e.g. goto's, aliased memory) and also non executable specifications e.g. in first order logic. The WSL language has been designed from the start to be both a powerful programming language and a language which is easy to transform. This is not the case for existing programming languages. All transformations are expressed in terms of WSL.

Thus the user starts with $\mathbf{P}'$, and selects a transformation from a library of pre-proven transformations. This is applied, resulting in an intermediate representation $\mathbf{S}_1$. Subsequent transformations may be applied to transform the software into $\mathbf{S}_2$, $\mathbf{S}_3$, ..., $\mathbf{S}_i$. The end point may depend on the need; for example, the user may wish only to perform simple restructuring, or they may need to extract an abstract specification.

Such an approach clearly lends itself to tool support, though the central work of designing the wide spectrum language and proving the transformations has to be done beforehand.

The main focus of this paper is the results of using this approach.

2 The Method

One of the major results from our research, which the availability of a prototype tool has helped to produce, is the development of a method for reverse engineering using formal transformations. The method is based on the following stages:

1. Establish the reverse engineering environment. This will involve a CASE tool to record results, maintain different versions of code, specifications, and documentation and the links between them; together with a WSL code browser and transformation system.
2. Collect the software to be reverse engineered. This involved finding the current versions of each subsystem and making these available to the CASE tool.
3. Produce a high-level description of the system. This may already be available in the documentation, since the documentation at this level rarely needs to be changed, and is therefore more likely to be up to date. The documentation is supplemented by the results of a cross reference analysis which records the control flow and data dependencies among the subsystems.
4. Translate the source code into WSL. This will usually be an automatic process involving parsing the source files and translating the language structures into equivalent WSL structures.
5. "Inverse Engineering", i.e. reverse engineering through formal transformations. It involves the automatic and manual application of various transformations to restructure the system and express it at increasingly higher levels of abstraction. This is carried out by iterating over the following four steps:
   (a) Restructuring transformations. These include removing **goto** statements, eliminating flags, removing redundant tests, and other optimisations. The effect of this restructuring is to reveal the "true" structure of the program which may be obscured by poor design or subsequent patching and enhancements. This stage is more radical than can be achieved by existing automatic restructuring systems since it takes note of both data flow

and control flow, and includes both syntactic and semantic transformations. We have however had considerable success with automating the simpler restructuring transformations, by implementing heuristics elicited from experienced program transformation users.

(b) Analyse the resulting structures in order to determine suitable higher-level data representations and control structures.

(c) Redocument: record the discoveries made so far and any other useful information about the code and its data structures.

(d) Implement the higher-level data representations and control structures using suitable transformations. A powerful technique we have developed for carrying out these data refinements is to introduce the abstract variables into the program as "ghost" variables (variables whose values are changed, but which do not affect the operation of the program in any way), together with invariants which make explicit the relationship between abstract and concrete variables. Then, one by one, the references to concrete variables are replaced by references to the new abstract variables. Finally, the concrete variables become "ghost" variables and can be removed. See [13] for an example of this process; it is also used extensively in [10]. In general, if our analysis in step 5b is correct then the result of this stage is likely to be in a form suitable for further restructuring.

6. Acceptance test: We now have a high-level specification of the whole system which should go through the usual Q.A. and acceptance tests.

2.1 Theoretical Foundation

This project originated not in software maintenance, but in theoretical research, developing a language in which proofs of equivalence for program transformations could be achieved as easily as possible for a wide range of constructs.

WSL is the "Wide Spectrum Language" used in our program transformation work, which includes low-level programming constructs and high-level abstract specifications within a single language. By working within a single formal language we are able to prove that a program correctly implements a specification, or that a specification correctly captures the behaviour of a program, by means of formal transformations in the language. We don't have to develop transformations between the "programming" and "specification" languages. An added advantage is that different parts of the same program can be expressed at different levels of abstraction, if required.

A *program transformation* is an operation which modifies a program into a different form which has the same external behaviour (it is equivalent under a precisely defined denotational semantics). Since both programs and specifications are part of the same language, transformations can be used to demonstrate that a given program is a correct implementation of a given specification.

A *refinement* is an operation which modifies a program to make its behaviour more defined and/or more deterministic. Typically, the author of a specification will allow some latitude to the implementor, by restricting the initial states for which the specification is defined, or by defining a nondeterministic behaviour (for example, the program is specified to calculate a root of an equation, but is allowed to choose which of several roots it returns). In this case, a typical implementation will be a *refinement* of the specification rather than a strict equivalence. The opposite of refinement is *abstraction*: we say that a specification is an abstraction of a program which implements it. See [6,7] and [1] for a description of refinement.

The syntax and semantics of WSL are described in [9,13] so will not be discussed in detail here. Most of the constructs in WSL, for example **if** statements, **while** loops, procedures and functions, are common to many programming languages. However there are some features relating to the "specification level" of the language which are unusual. Expressions and conditions (formulae) in WSL are taken directly from first order logic: in fact, an infinitary first order logic is used, which allows countably infinite disjunctions and conjunctions, but this is not essential for understanding

this paper. This means that statements in WSL can include existential and universal quantification over infinite sets, and similar (non-executable) operations.

Program transformations may be used to refine a specification to an executable implementation (forward engineering) or to abstract a formal specification from program source code (reverse engineering). The main novel theoretical contribution lies in the use of infinitary logic in both the kernel language and proof meta language to widen the scope of the transformations it is possible to prove. In particular, it has been possible to develop general purpose transformations for loops and for recursive procedures which can be applied without needing loop invariants.

2.2 The FermaT Tool

The initial prototype of the tool was developed as part of an Alvey project at the University of Durham [14]. This work on applying program transformation theory to software maintenance formed the basis for a joint research project (the ReForm project) between the University of Durham, CSM Ltd and IBM UK Ltd. whose aim was to develop a tool which would interactively transform assembly code into high-level language code and **Z** specifications. The prototype has since been completely redeveloped into an industrial-strength tool, called FermaT, which is capable of dealing with medium sized (up to 20,000 lines) modules of source code. Translators have been developed for IBM 370 Assembler and JOVIAL. A COBOL translator is currently under development, and this will enable to tool to assist with migration from Assembler or JOVIAL to COBOL II. Translators for C and ADA are also planned for the future.

A practical system for reverse engineering has to deal with real programs, not laboratory or toy examples. More specifically, the following requirements were identified:

- The tool must cope with the usual programming constructs and their uses (and abuses) including Gotos, global variables, aliasing, recursion, pointers, side effects etc.;
- It is not acceptable to assume that the code has been developed or maintained using structured methods. Real code must be acceptable, and major restructuring may be required before proper reverse engineering can start. This should be carried out automatically (or semi automatically) by the system.
- Transformations in the library must be proven correct, so that the user can employ them with confidence, but also so that the user does not have to undertake such proofs. The transformations need applicability conditions, and these must be mechanically checked by the tool. In this way, all responsibility for correctness lies with the tool—there are no generated "proof obligations" which the user must discharge before correctness can be guaranteed;
- It must be possible to select a sub-component of a large existing system and to guarantee to preserve the interactions of the sub-component with the rest of the system. This permits attention to maintenance 'hot spots', and also permits a piecemeal approach to reverse engineering;
- The correctness of the implementation must be well established.

The main components of the tool are shown in Figure 1. The core of the tool is the library of proven transformations together with the transformation engine. The transformations in the library were proven before the tool was built. They allow a construct in WSL to be recast into another WSL construct while ensuring that the semantics are preserved. The software maintainer using the tool has only to select a transformation and apply it. He or she does not have to do the proof; the system's transformation engine checks that the transformation is applicable.

However, the first stage is to load the source code into the tool, and this is achieved by the source language to WSL translator as a batch job. The equivalent WSL code is stored internally as an abstract syntax tree (together with ancillary information to aid applicability checking). Further details are given in Section 3.2.2.

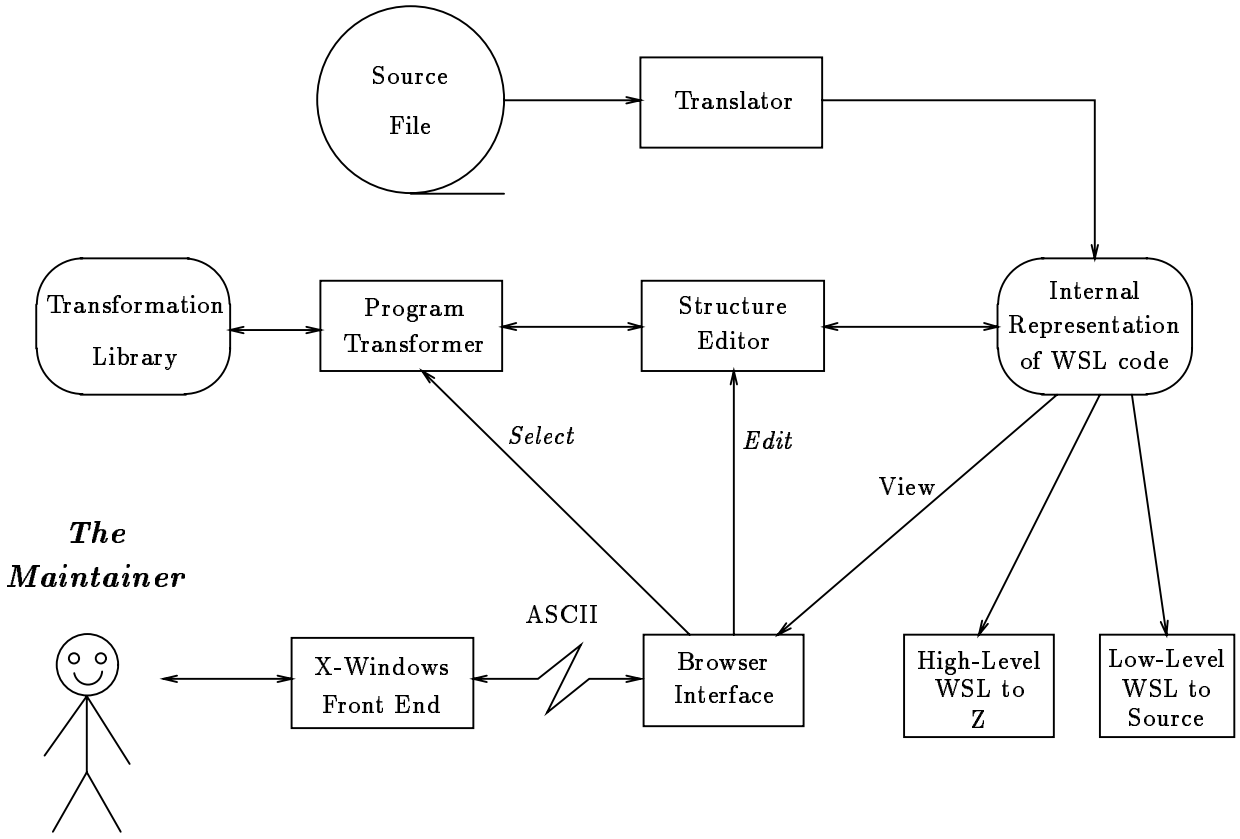

Figure 1: The Architecture of the tool

The system is interactive and incorporates a graphical front end, pretty-printer and browser. This allows the programmer to move through the program, apply transformations, undo changes he or she has made, and in special circumstances, edit the program manually: but always in such a way that it is syntactically correct. The system automatically checks the applicability conditions of a transformation before it is applied, or even presented in one of the menus. This means that the correctness of the resulting transformed program is guaranteed by the system rather than being dependent on the user. A history/future structure is built-in to allow back-tracking and forward-tracking enabling the programmer to change his or her mind. The system stores the results of its analysis of a program fragment as part of the program, so that re-calculation of the analysis is avoided wherever possible.

The interactive approach means that the programmer is always in control of how much or how little the program is changed. Unlike conventional restructuring tools, the programmer drives the process, and is not simply presented with a restructured program he or she still does not understand. The ultimate shape of the program has been determined by the user, not by some remote tool developer. If at any point the user is unhappy with the latest state of the program, he or she can always undo the last (sequence of) transformation(s).

Presenting the programmer with a variety of different but equivalent representations of the program can greatly aid the comprehension process, making best use of human problem solving abilities (visualisation, logical inference, kinetic reasoning etc).

Note that the theoretical foundation work which proves that each transformation in the system preserves the semantics of any applicable program is *essential* if this method is to be applied to practical software maintenance or reverse engineering. It must be possible to work with programs which are poorly (or not at all) understood, and it must be possible to apply many transformations which drastically change the structure of the program (as in the examples in Section 3.1) with a

very high degree of confidence in the correctness of the result.

Finally, the tool is also capable of computing standard complexity metrics for a selected region of the WSL program, and presenting them in graphical form to show changes with time. Currently, McCabe, structural, size, control flow, data flow and branch-loop metrics may be computed [2].

It has been learned through experience that a user often employs a pattern of transformations, and it is easy within the tool to group such transformations into more powerful single transformations. Currently, such 'super transformations' are added by the tool builder, but as the transformations are represented internally in WSL it would not be difficult to allow the user to do this. The system is constructed as a hierarchy of abstract machines, each of which is formally specified. Additionally, much of the tool is written in either WSL or $\mathcal{META}$WSL, an extension of WSL used for representing transformations. This makes it possible for the developers to use the tool in the maintenance of its own source code.

3 Results

3.1 Results with Benchmark Programs

In this section we describe some of the results of applying our program transformation system to several small but challenging example programs. The examples selected pose particular challenges to the approach, because they exhibit a combination of complex control flow and complex data structures, and use data structures for multiple purposes.

3.1.1 A Report Generator

The first example is reverse engineering a simple program with a complex control flow. This was taken from a programming textbook, although it is a classic example of bad programming style! In [13] we translate the program into WSL and use program transformations to reverse-engineer it to an abstract specification. This gave confidence that the approach could be used on a program with a complex control structure, whose operation was not understood before starting the reverse engineering process.

3.1.2 The Schorr-Waite Graph Marking Algorithm

Our next example is a forward engineering example: the Schorr-Waite graph marking algorithm [8]. This has acquired the status of a standard testbed for program verification techniques applied to complex data structures. In [10] we present a complete derivation of the algorithm, starting with a mathematical specification of graph marking. We develop a simple recursive algorithm by applying general purpose transformations, and then apply Schorr and Waite's "pointer switching" technique to develop an efficient iterative algorithm. The central idea behind the pointer switching technique is that when we return from having marked the left subtree of a node we know what the value of l[x] is for the current node (since we just came from there). So while we are marking the left subtree, we can use the array element l[x] to store something else—for instance a pointer to the node we will return to after having marked this node. Similarly, while we are marking the right subtree we can store this pointer in r[x]. In [10] we use program transformations to apply this technique to a number of different algorithms.

The transformation approach proved to be a powerful way to prove the correctness of these challenging algorithms. A correctness proof for the algorithm has to show that:

1. The original graph structure is preserved by the algorithm (although it is temporarily disrupted as the algorithm proceeds);
2. The algorithm achieves the correct result (all reachable nodes are marked).

Most published correctness proofs for the algorithm have to treat these two problems together. The methods involving assertions (and intermittent assertions) require an understanding of the

"total situation" at any point in the execution of the program. In contrast, our approach makes it possible to separate the two requirements, and thereby to apply the same technique to a number of different algorithms. We have derived several marking algorithms which make use of the pointer switching ides and which illustrate the advantages of transformational development using a wide spectrum language:

1. The development divides into four stages: (i) Recursive Algorithm; (ii) Apply the pointer switching idea; (iii) Recursion Removal; and (iv) Restructuring. Each stage uses general-purpose transformations with no complicated invariants or induction proofs;
2. The method easily scales up to larger programs: for example, the hybrid algorithm presented in [10] is much more complex than the simple algorithms, yet our development uses the same transformations and involves no new ideas or proofs.

3.1.3 Topological Sorting

Our third example is a reverse engineering problem. In [4], Knuth and Szwarcfiter present an algorithm for determining all the embeddings of a given partial order into a total order. The algorithm is highly unstructured with complex control flow combined with complex data structures. In [11] we restructure and analyse the algorithm using program transformations. The analysis of the algorithm breaks down into several stages, culminating in a formal specification of topological sorting.

1. Restructure to remove some of the control-flow complexity;
2. Recast as an iterative procedure, "abstracting away" the error cases so that recursion introduction can be applied easily;
3. Apply the recursion introduction transformation;
4. Restructure the resulting recursive procedure;
5. Add abstract variables to the program and update them in parallel with the actual (concrete) variables;
6. Replace references to concrete variables by equivalent references to abstract variables;
7. Remove the concrete variables to give an abstract program;
8. Show that the abstract program is a refinement of the specification of topological sorting.

3.1.4 Polynomials in Several Variables

The previous example exhibited a combination of control flow complexity with highly complex data structures. Our fourth example (also a reverse-engineering example) is a program with complex data structures (trees implemented as four-way linked structures) and highly complex control flow. Algorithm 2.3.3.A from Knuth's "Fundamental Algorithms" [5] (P.357) is an algorithm for the addition of polynomials represented using four-directional links. Knuth describes this as having "a complicated structure with excessively unrestrained **goto** statements" and goes on to say "I hope someday to see the algorithm cleaned up without loss of its efficiency". In [12] we use program transformations to manipulate the program, using semantics-preserving operations, into an equivalent high-level specification.

3.1.5 Conclusions

These and other case studies have demonstrated that the transformational approach can be applied to both forward and reverse engineering problems. During this time, a method and strategy for reverse engineering using formal transformations has been developed and successfully tested. Certain features of the case studies indicate that the approach will scale up to industrial-scale software. This is addressed in the next section.

3.2 Results with Industrial-Scale Software

3.2.1 IBM 370 Assembler

Experiments have been undertaken on modules of Assembler taken from real application programs. The majority have been between 500 and 2,000 lines but some have been up to 20,000 lines. These experiments have shown that programs which have been transformed using the tool can be expressed in a form which subjectively is much easier to understand than the original. This applies particularly to real programs which have been modified over many years.

3.2.2 Modelling Assembler in WSL

Constructing a useful scientific model necessarily involves throwing away some information: in other words, to be useful a model must be inaccurate, or at least idealised, to a certain extent. For example "ideal gases", "incompressible fluids" and "billiard ball molecules" are all useful models which gain their utility by abstracting away some details of the real world. In the case of modelling a programming language, such as Assembler, it is theoretically possible to have a perfect model of the language which correctly captures the behaviour of all assembler programs. Certain features of Assembler, such as branching to register addresses, self-modifying code and so on, would imply that such a model would have to record the entire state of the machine, including all registers, memory, disk space, and external devices, and "interpret" this state as each instruction is executed. Unfortunately, such a model is useless for inverse engineering[1] purposes since such trivial changes as deleting a NOP instruction, or changing the load address of a module, can in theory change the behaviour of the program.

What we need is a practical model for assembler programs which is suitable for inverse engineering, and is wide enough to deal with all the programming constructs we are likely to encounter. Our approach involves three types of modelling:

1. Complete model: Each assembler instruction is translated into WSL statements which capture all the effects of the instruction. The machine registers and memory are modelled as arrays, and the condition code as a variable. Thus, at the translation stage we don't attempt to recognise "if statements" as such, we translate into statements which assign to cc (the condition code variable), and statements which test cc. The automatic restructuring and simplification state can usually remove all references to cc, presenting the maintainer with a structured program expressed in **if** statements, loops and actions;
2. Partial model: Branches to register are modelled by attempting to determine all possible targets of such a branch (including all labels and jump instructions which follow labelled instructions). Each label is turned into a separate action with an associated value (the relative address). A "store return address" instruction stores the *relative* address in the register. A "branch to register" instruction passes the relative address to a "dispatch" action which tests the value against the set of recorded values, and jumps to the appropriate label. This can deal with simple cases of address arithmetic (including jump tables) but may theoretically be defeated if more complex address manipulations are carried out before a branch to register instruction is executed;
3. Self-modifying code: This is not addressed, except for some special cases which are recognised by the translator. In many environments the code must be re-entrant, or is to be blown into a ROM, and therefore cannot be modified. In other cases, the self-modification may be recognised by the translator and may require human intervention to determine a suitable WSL equivalent.

One of the major drawbacks of automatic program restructurers [3] is that complex control structures are replaced by complex data flow structures involving additional flag and sentinel variables with meaningless names inserted by the tool. This does not occur with our tool, and users

[1] We use the term "inverse engineering" to mean "reverse engineering through formal transformations."

resolve the underlying structural problems because the transformations make it easy to do so. It is also straightforward to avoid dispersing code that previously was together. This method has the advantage that performance problems and errors which exist deeply buried in heavily modified code become much more easily observable.

The following table demonstrates some sample results for a typical Assembler module:

Stage	No. of Statements	McCabe Cyclometric	Control Flow/ Data Flow	Branch/ Loop	Structural
Initial	4,785	1,299	5,270	1,481	37,168
Stage 1	4,775	1,298	5,150	1,476	36,246
Stage 3	3,372	592	3,711	924	23,140
Stage 5	3,454	577	3,761	970	23,178
Stage 7	2,920	338	3,101	631	17,616
Final	2,508	310	2,590	544	15,466

"Statements" is the number of executable statements in the program, "McCabe Cyclometric" is the usual McCabe cyclometric complexity, "Control Flow/Data Flow" is a control flow and data flow metric, "Branch/Loop" is a metric which counts the size of loops, and "Structural" is a metric which gives a weighted sum of the structural features of the program.

There is no single accepted metric for "complexity" or "maintainability", however, the fact that *all* the metrics used show a significant reduction in complexity indicates that a very useful increase in maintainability has been achieved, and this is confirmed by a more "qualitative" manual examination of the before and after source code. To provide an additional check on semantic equivalence a few modest examples have been transformed and then (in the absence of a suitable translator) hand converted to Assembler, reinstalled and re-executed. Apart from minor errors caused by hand translation, the examples worked first time.

A major attraction of the tool has turned out to be the transformations which convert in-line code to procedures, and global variables into parameters. This enables the user to convert a large, unstructured, monolithic piece of code into a main program which calls a set of single-entry single-exit procedures. These transformations alone can make a large difference to the understandability of the code, and prepare it for the recognition of abstract data types.

3.2.3 JOVIAL

More recently we have constructed JOVIAL to WSL and WSL to JOVIAL translators, and have used the tool for restructuring a number of JOVIAL source modules, ranging up to around 5,000 lines. The two examples in this section were selected for case studies because, despite their fairly moderate size, they contained some very complicated code which made them difficult to analyse and maintain using traditional methods. The aim was to restructure the programs for ease of maintainability.

3.2.4 Module A

The first example consisted of 861 lines of JOVIAL in a main routine and eight procedures. The module was translated from JOVIAL to WSL, restructured and simplified using the tool, and then translated back to JOVIAL for testing. The following table shows the improvements achieved in terms of various complexity metrics:

Stage	No. of Statements	McCabe Cyclometric	Control Flow/ Data Flow	Branch/ Loop	Structural
Initial	954	120	845	701	10,371
Final	392	92	343	146	5,115

These improvements were achieved by the use of a wide range of transformations, with the choice of transformations was guided by the aim of reducing the complexity as measured by the above metrics. In some cases the immediate application of a powerful transformation such as **Collapse Action System** achieved the desired results; in others, some localised restructuring using some of the more specialised transformations was necessary first. The whole process, including the generation of metrics and call graphs, took about half a day for an experienced user of the tool.

Activity	Step	No. of Statements	McCabe Cyclometric	Control Flow/ Data Flow	Branch/ Loop	Structural
Initial	(1)	954	120	845	701	10,371
Restructure main routine	(2)	815	120	728	584	9,091
Simplify A.S.	(3)	795	116	716	572	8,941
Merge calls	(3.1)	791	116	712	568	8,901
Collapse A.S (Undone)	(3.2)	1,005	188	876	538	10,820
Remove recursion	(4)	913	142	838	537	9,869
Restructure action body	(5)	812	122	731	539	9,023
Collapse A.S.	(6)	783	106	729	537	8,963
Create Procedure	(7)	756	98	699	513	8,673
Collapse A.S GETTY	(8)	732	98	669	507	8,404
Collapse A.S. ProcI	(9)	692	98	627	464	7,994
Remove loop	(10)	686	98	624	468	7,959
Collapse ProcG, ProcF	(11)	677	98	615	459	7,874
Collapse ProcE	(12)	616	98	552	384	7,254
Remove loop	(13)	612	98	552	396	7,239
Collapse ProcD and simplify	(14)	555	98	497	341	6,682
Restructure ProcC	(15)	436	97	383	227	5,534
Collapse ProcC	(16)	431	95	374	177	5,445
Collapse ProcB and simplify	(17)	392	92	343	146	5,115

3.2.5 Module B

The second JOVIAL example consisted of 2,564 lines of JOVIAL. The main sources of complexity in this module were:

1. Heavy use of labels and **goto**'s rather than structured programming practices, which make the control flow difficult to understand; and
2. Multiple exit paths from the program, including exits via calls to closed compound procedures which never return.

The main routine in the module contained the most complex code, but in addition, the procedure ProcI contained some particularly complex code.

A significant amount of simplification and restructuring of the raw WSL is performed automatically as part of the translation process from Jovial to WSL. One very significant area of restructuring is the conversion of closed compound procedures in the original Jovial into pure procedures in the WSL version of the program. Where it is possible for a closed compound procedure to perform a jump to a label which is outside its body, this behaviour is modelled by the setting of a variable in the WSL program which is tested when the corresponding procedure returns and a jump performed to the appropriate label. Thus, procedures always return to the point from which they were called, and any jumps to labels are made explicit. A further advantage is that closed compound procedures now appear in the procedure call graph of a module.

The following table shows the effect of the restructuring process:

Stage	No. of Statements	McCabe Cyclometric	Control Flow/ Data Flow	Branch/ Loop	Structural
Initial	2,518	847	2,212	1,109	22,986
Final	2,222	780	2,138	863	20,832

Figure 2 shows the call graph of the main module before restructuring, while Figure 4 shows the call graph after restructuring. For procedure ProcI, Figure 3 is the call graph before restructuring and Figure 5 is the same procedure after restructuring.

3.2.6 Module C

The third JOVIAL example consisted of a collection of procedures. The following table contains a summary of the metrics taken at different points in the transformation process. There is one section for each of the WSL procedures and for the main body of the program. The first section is for the program as a whole.

Name	McCabe Cyclometric	Control Flow/ Data Flow	Branch/ Loop	Structural
XRP.o	624	3,388	2,338	35,464
XRP.a	567	1,754	655	19,093
XRP.t	524	1,624	260	17,375
XRPPAR.a	86	231	58	2,024
XRPPAR.b	86	223	45	1,934
XRPPAR.c	86	220	40	1,909
XRPPAR.d	86	212	29	1,864
XRPPAR.e	72	230	28	1,851
XRPPDR.a	32	75	44	875
XRPPDR.b	31	69	31	823
XRPPDR.t	30	61	25	725
XRPPPD.a	6	59	7	429
XRPPPD.t	6	55	2	389
XRPRP.a	78	192	108	2,278
XRPRP.b	76	187	103	2,213
XRPRP.c	75	187	101	2,199
XRPRP.d	71	188	96	2,189
XRPRP.e	71	187	93	2,174
XRPTL.a	103	326	59	2,524
XRPTL.b	100	317	53	2,441
XRPTL.c	99	310	37	2,361
XRPTL.d	95	324	35	2,325
XRPTL.e	95	322	34	2,305
XRPTL.f	95	320	32	2,285
XRPX.a	8	22	11	242
XRPX.t	8	17	1	197
XRPY.a	10	30	12	363
XRPY.t	9	23	3	298
XRPZ.a	32	78	38	805
XRPZ.b	32	72	32	735
XRPZ.c	31	64	12	680
XRPZ.d	31	62	7	670
XRPZ.t	28	62	3	65

4 Conclusions

For simple restructuring, skills are needed to identify a simplification strategy and then to select transformations to achieve this goal. However, for acquiring the specification of an existing program, the user also needs to be an expert in software engineering and in the application domain. This confirms the original design objective of providing assistance to the expert maintainer, rather than de-skilling or automating the maintenance task.

We originally thought that all users would want to go from code to specifications. In fact there is a spectrum of requirements, ranging from using the tool for code comprehension and simple restructuring, to reverse engineering from code to a high level of abstraction. Our approach deals with the whole spectrum of requirements.

We believe that the following main features have contributed to the success of our approach:

- Use of weakest preconditions expressed in infinitary logic;
- Starting with a small, tractable kernel language, extended via definitional transformations;
- Use of an imperative kernel language, with functional constructs added via definitional transformation, rather than a functional kernel language;
- Developing the transformation theory in parallel with the language development;
- Dealing with assembler via simple translation followed by automatic restructuring and simplification;
- Developing an interactive, semi-automatic tool, rather than attempting complete automation;
- Mechanical checking of the correctness conditions at each step, with only valid transformations appearing in the menus;
- Knowledge elicitation: using the prototype and manual case studies to see how the experienced user solves problem, and then implementing these methods and heuristics;
- The use of generic transformations for merging, moving, separating etc.; these are automatically expanded into the appropriate transformation for each situation;
- Rapid prototyping development, with the system organised as a collection of abstract machines with formally defined interfaces;
- Separation of front-end issues into a separate program.

Acknowledgements

The research described in this paper has been partly funded by Alvey project SE-088, partly through a DTI/SERC and IBM UK Ltd. funded IEATP grant "From Assembler to Z using Formal Transformations" and partly by SERC (The Science and Engineering Research Council) project "A Proof Theory for Program Refinement and Equivalence: Extensions".

References

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

[2] K. H. Bennett, H. Yang & T. Bull, "A Transformation System for Maintenance—Turning Theory into Practice," *Conference on Software Maintenance, Orlando, Florida* (1992).

[3] F. W. Calliss, "Problems With Automatic Restructurerers," *SIGPLAN Notices* 23 (Mar., 1988), 13–21.

[4] D. E. Knuth & J. L. Szwarcfiter, "A Structured Program to Generate All Topological Sorting Arrangements," *Inform. Process. Lett.* (1974).

[5] D. K. Knuth, *Fundamental Algorithms*, The Art of Computer Programming #1, Addison Wesley, Reading, MA, 1968.

[6] C. Morgan, *Programming from Specifications*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[7] C. C. Morgan, K. Robinson & Paul Gardiner, "On the Refinement Calculus," Oxford University, Technical Monograph PRG-70, Oct., 1988.

[8] H. Schorr & W. M. Waite, "An Efficient Machine-Independant Procedure for Garbage Collection in Various List Structures," *Comm. ACM* (Aug., 1967).

[9] M. Ward, "Proving Program Refinements and Transformations," Oxford University, DPhil Thesis, 1989.

[10] M. Ward, "Derivation of Data Intensive Algorithms by Formal Transformation," Submitted to IEEE Trans. Software Eng., 1993.

[11] M. Ward, "Program Analysis by Formal Transformation," University of Durham Technical Report, 1994.

[12] M. Ward, "Reverse Engineering through Formal Transformation Knuths "Polynomial Addition" Algorithm," *The Computer Journal* (1994).

[13] M. Ward, "Abstracting a Specification from Code," *J. Software Maintenance: Research and Practice* 5 (June, 1993), 101–122.

[14] M. Ward, F. W. Calliss & M. Munro, "The Maintainer's Assistant," *Conference on Software Maintenance 16th–19th October 1989, Miami Florida* (1989).

Appendix: Call Graphs for Module B and ProcI

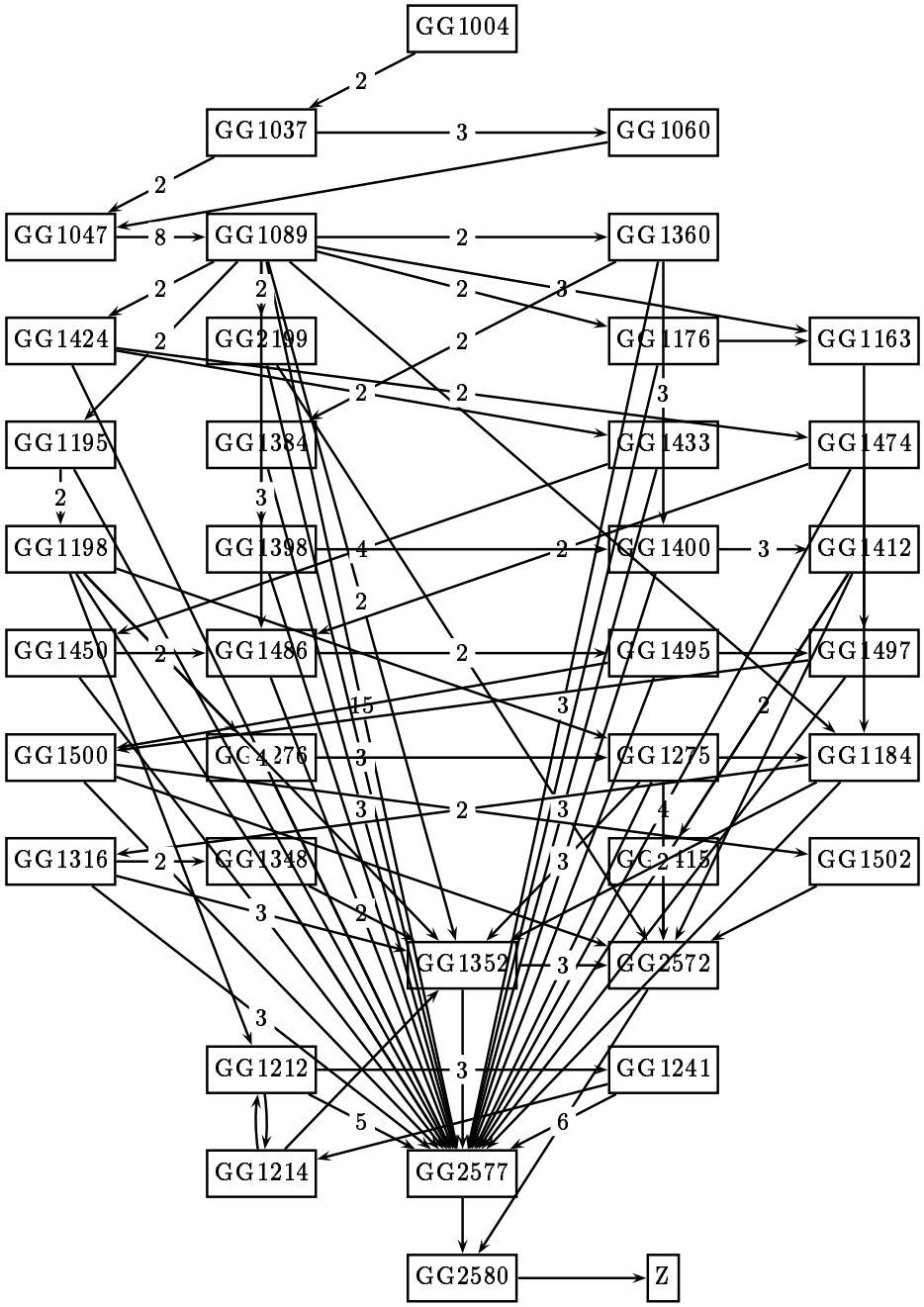

Figure 2: Call Graph of Module B before restructuring

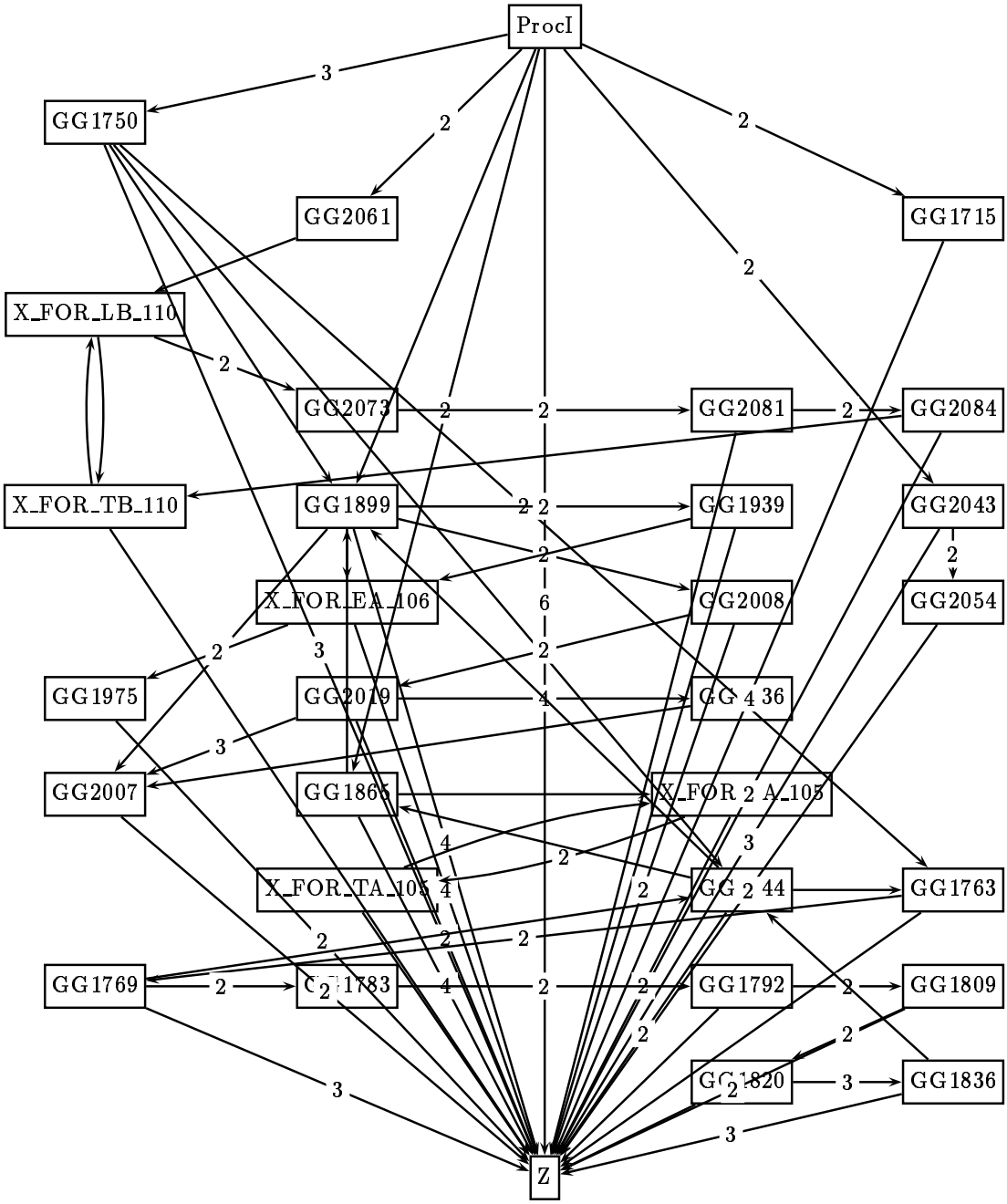

Figure 3: Call Graph of ProcI before restructuring

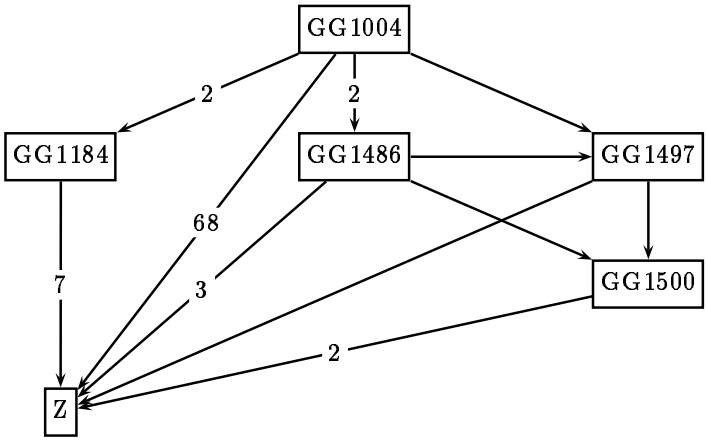

Figure 4: Call Graph of Module B after restructuring

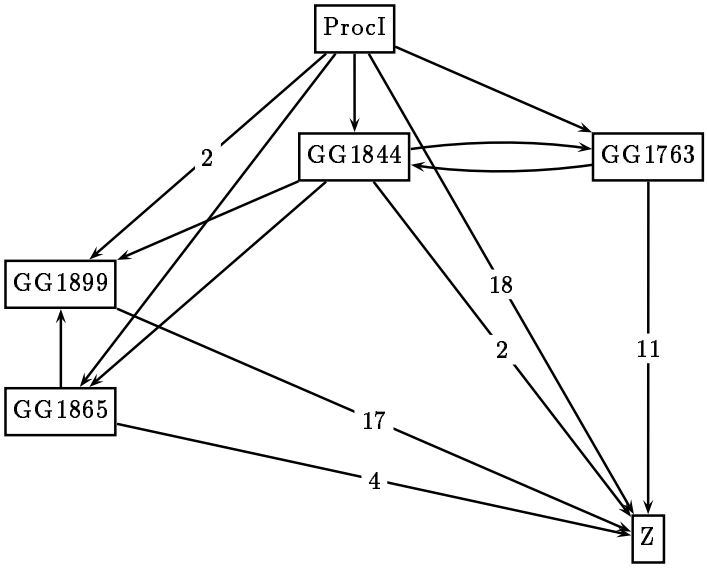

Figure 5: Call Graph of ProcI after restructuring