A Practical Program Transformation System For Reverse Engineering

M.P. Ward
Computer Science Department
Science Labs
South Rd
Durham DH1 3LE

K.H. Bennett
Computer Science Department
Science Labs
South Rd
Durham DH1 3LE

Abstract

Program transformation systems provide one means of formally deriving a program from its specification. The main advantage of this development method is that the executable program is correct by construction. In this paper we describe a tool called ReForm which is designed to address the inverse problem to support the extraction of a specification from existing program code, using transformations. This is an important activity during software maintenance.

One of the problems of transformation systems is the scarcity of practical tools which can address industrial scale problems, rather than contrived laboratory "toy" problems. The main contribution of this paper is an analysis of the important software engineering factors that contribute to a successful transformation based tool. Results from using the tool are also presented.

1 Background

Four separate surveys carried out between 1977 and 1990 [7,11,14,18] and summarised in [6], show that between 40% and 60% of all commercial software effort is devoted to software maintenance. Despite this, much of the research in software engineering has concentrated on methods for developing new code rather than methods for analysing, correcting and enhancing existing code. This is especially true for work on program transformation systems. Other studies have shown that much of the effort in maintenance is in the area of code analysis, and reverse engineering (transforming code into equivalent representations at higher levels of abstraction) can be a useful aid to code analysis. More importantly, many problems with current maintenance practice are caused by the fact that all maintenance is carried out at the code level. Formal reverse engineering (which we call inverse engineering below), can recover abstract specifications from the code via program transformations. This enables maintenance to be carried out at the appropriate level of abstraction, which in turn, renders more efficient and more effective maintenance.

In this paper we describe a practical program transformation system, based on a formal theory for program refinement and equivalence, which is currently being used for reverse-engineering assembler code. The system uses formal program transformations to restructure the code and extract high-level specifications. By a "specification" we mean a sufficiently precise definition of the input-output behaviour of the program, where in practice "sufficiently precise" means "expressable in first order logic and set theory". This includes **Z**, VDM [9,12], and most other formal specification languages.

The system uses a Wide Spectrum Language (called WSL), developed in [20,24,28] which includes low-level programming constructs and high-level abstract specifications within a single language. Naturally, the translation of specifications or source code written in an informal language (including incompletely or inconsistently defined programming languages) into WSL cannot be formally proved correct. The semantics of a source file may depend on the particular compiler/interpreter and target machine used to execute it. The best that can be done in such cases is to make the translator as simple as possible by translating each statement as fully as possible, including all the implied details, and explicitly record any assumptions made about the compiler/interpreter and operating environment. Redundant details in the translated WSL program, introduced by this process, are easily removed by optimising transformations.

Working within a single formal language means that the proof that a program correctly implements a specification, or that a specification correctly captures the behaviour of a program, can be achieved 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 program can be expressed at different levels of abstraction, if required. (Feather [5] refers to a *narrow-spectrum language* as one which picks up some relatively narrow style of program of specification description and focuses on finding notations and manipulations to support the expression and application of transformations within that style).

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.

There are several distinct advantages to a transformational approach to program development and reverse engineering:

- The final developed program, or derived specification, is correct *by construction*;
- Transformations can be described by *semantic rules* and can thus by used for a whole class of problems and situations;
- Due to formality, the whole process of program development, and reverse engineering, can be supported by the computer. The computer can check the correctness conditions for each step, apply the transformation, store different versions, attach comments and documentation to code, preserve the links between code and specifications etc.;
- Provided the set of transformations is sufficiently powerful, and is capable of dealing with all the low-level constructs in the language, then it becomes possible to use program transformations as a means of restructuring and reverse-engineering existing source code (which has not been developed in accordance with any particular formal method). We have coined the term *inverse engineering* to refer to reverse engineering carried out by formal transformation;
- The user does not have to understand the code before transforming it: the program can be transformed into a more understandable form before it is analysed. Thus transformations provide a powerful program understanding tool.

In [21,23,25,27] program transformations are used to derive a variety of efficient algorithms from abstract specifications. In [22] the same transformations are used in the reverse direction: starting with a small but tangled and obscure program we were able to use transformations to restructure the program and derive a concise abstract representation of its specification.

An alternative approach to transformational development, which is generally favoured in the **Z** community and elsewhere, is to allow the user to select the next refinement step (for example, introducing a loop) at each stage in the process. Each step will carry a set of *proof obligations*, which are theorems which must be proved for the refinement step to be valid. For example, introducing a loop requires the user to supply an invariant and a variant function, and to prove:

1. That the invariant is preserved by the body of the loop;
2. The variant function is decreased by the body of the loop;
3. The invariant plus terminating condition are sufficient to implement the specification.

Discharging these proof obligations can often involve a lot of tedious work, and much effort is being exerted to apply automatic theorem provers to aid with the simpler proofs. However, Sennett in [19] indicates that for "real" sized programs it is impractical to discharge much more than a tiny fraction of the proof obligations. He presents a case study of the development of a simple algorithm, for which the implementation of one function gave rise to over one hundred theorems which required proofs. Larger programs will require many more proofs. In practice, since few if any of these proofs will be rigorously carried out, what claims to be a formal method for program development turns out to be a formal method for program specification together with an *informal* development method. Also, although this approach could in theory be used for reverse engineering as well as development, in practice the proof obligations become much more difficult to fulfill, and the selection of an appropriate abstraction (for which the method provides no help), much more difficult.

The *Refinement Calculus* approach to program derivation [8,15,17] is superficially similar to our program transformation method. It is based on a wide spectrum language, using Morgan's specification statement [16] and Dijkstra's guarded commands [4]. However, their language has very limited programming constructs: lacking loops with multiple exits, action systems with a "terminating" action, and side-effects;

and their proof methods require that any loops introduced must be accompanied by suitable invariant condition and variant function. Determining suitable invariants for all the loops in a given program, especially one which was not developed using modern structured programming methods, is extremely difficult, and this makes it unlikely that the refinement calculus in its current state can be applied to practical reverse engineering problems.

By basing our proof methods on weakest preconditions expressed in infinitary logic [4,10,20] we have been able to develop general purpose transformations for loops which can be applied without needing loop invariants. These have been used successfully in deriving programs from specifications [21,27] and reverse engineering program into specifications [22,30].

2 Motivation

Any practical program transformation system for reverse engineering has to meet the following requirements:

1. It has to be able to cope with all the usual programming constructs: loops with exits from the middle, **goto**s, recursion etc.;
2. Techniques are needed for dealing with variable aliasing, side-effects and pointers;
3. It cannot be assumed that the code was developed (or maintained) according to a particular programming method: real code ("warts and all") must be acceptable to the system: in particular, significant restructuring may be required before the real reverse engineering can take place. It is important that this restructuring can be carried out automatically or semi-automatically by the transformation system;
4. It should be based on a formal language and formal transformation theory, so that it is possible to *prove* that all the transformations used are semantic-preserving. This allows a high degree of confidence to be placed in the results;
5. The formal language should ideally be a wide spectrum language which can cope with both low-level constructs such as **goto**s, and high-level constructs, including nonexecutable specifications expressed in first order logic and set theory;
6. Translators are required from the source language(s) to the formal language: many large software systems are written in a combination of different languages;
7. It must be possible to apply transformations without needing to understand the program first: this is so that transformations can be used as a program understanding and reverse engineering tool;
8. It must be possible to extract a module, or smaller component, from the system for analysis and transformation, with the transformations guaranteed to preserve all the interactions of that component with the rest of the system. This allows the maintainer to concentrate on "maintenance hot spots" in the system, without having to process the entire source code (which may amount to millions of lines);
9. An extensive catalogue of proven transformations is required, with mechanically checkable correctness conditions and some means of composing transformations to develop new ones;
10. An interactive interface which pretty-prints each version on the display will allow the user to instantly see the structure of the program from the indentation structure;
11. The correctness of the transformation system itself must be well-established, since all results depend of the transformations being implemented correctly;
12. A method for reverse engineering by program transformation needs to be developed alongside the transformation system.

In addition to these requirements, a reverse engineering tool for Assembler programs will also need to cope with highly unstructured code, a complete lack of data structures (everything is reduced to words of memory), and in extreme cases, self-modifying code. A particular problem in reverse-engineering assembler code is determining procedure boundaries: a procedure call is initiated (in IBM 370 assembler) by storing a return address and jumping. However, later on there may be other jumps, the return address may be overwritten, or incremented by the length of one or more instructions, and so on. Thus, the program analysis cannot assume that control will return to the point of the call. Assembler code thus provides a particularly challenging test for a transformation system.

3 The ReForm Tool

The ReForm tool (*Reverse Engineering through FORmal Methods*), is designed to automate much of the process of transforming code into specifications and specifications into code. This process can never be completely automated—there are many ways of writing the specification of a program, several of which may be useful for different purposes. So the tool must work interactively with the tedious checking and manipulation carried out automatically, while the maintainer provides high-level "guidance" to the transformation process. In the course of the development of the prototype, we have been able to capture much of the knowledge and expertise that we have developed through manual experiments, and case studies with earlier versions of the tool, and incorporate this knowledge within the tool itself. For example, restructuring a regular action system (a collection of **goto**s and labels) can now be handled completely automatically through a single transformation.

ReForm can be used as a transformation development system, starting with a high-level specification expressed in set-theory and logic notation (similar to **Z** or **VDM** [9]. It can also act on existing program code as a tool to aid comprehension by producing specifications (which can then be modified). The system can work with any language by first translating into the system's internal language, which is the Wide Spectrum Language WSL. Prototype stand-alone translators have been developed for IBM 370 assembler and a subset of BASIC. Transformations are themselves coded in an extension of WSL called *Meta*-WSL: in fact, much of the code for the prototype is written in WSL and this makes it possible to use the system to maintain its own code.

The initial prototype of ReForm was developed as part of an Alvey project at the University of Durham [29] whose aim was to develop a tool assist a maintenance programmer in understanding and modifying an initially unfamiliar program, given only the source code. This work on applying program transformation theory to software maintenance formed the basis for a joint research project between the University of Durham, CSM Ltd and IBM UK Ltd. whose aim is to develop a tool which will interactively transform assembly code into high-level language code and **Z** specifications. We have been able to transform the assembler code to a high-level language representation, replace the "areas of store" by the data structures they implement (using transformations which change the data representation of a program), and then transform this high-level language version into a specification. A prototype translator has been completed and tested on sample sections of assembler code from IBM's CICS product, and other large assembler systems ranging up to 20,000 lines, with very encouraging results (see Section 5).

3.1 Theoretical Foundations

A program $\mathbf{S}$ is a piece of formal text, i.e. a sequence of formal symbols. There are two ways in which we interpret (give meaning to) these texts:

1. Given a structure M for the logical language $\mathcal{L}$ from which the programs are constructed, and a final state space (from which we can construct a suitable initial state space), we can interpret a program as a function f (a *state transformation*) which maps each initial state s to the set of possible final states for s. By itself therefore, we can interpret a program as a function from structures to state transformations;
2. Given any formula $\mathbf{R}$ (which represents a condition on the final state), we can construct the formula $\mathrm{WP}(\mathbf{S}, \mathbf{R})$, the *weakest precondition* of $\mathbf{S}$ on $\mathbf{R}$. This is the weakest condition on the initial state such that the program $\mathbf{S}$ is guaranteed to terminate in a state satisfying $\mathbf{R}$ if it is started in a state satisfying $\mathrm{WP}(\mathbf{S}, \mathbf{R})$.

These interpretations give rise to two different notions of refinement: *semantic refinement* and *proof-theoretic refinement.*

3.2 Semantic Refinement

A *state* is a collection of variables (the *state space*) with values assigned to them; thus a state is a function which maps from a (finite, non-empty) set V of variables to a set $\mathcal{D}$ of values. There is a special extra state $\perp$ which is used to represent nontermination or error conditions. A state transformation f maps each initial state s in one state space, to the set of possible final states $f(s)$ which may be in a different state space. If $\perp$ is in $f(s)$ then so is every other state, also $f(\perp)$ is the set of all states (including $\perp$).

Semantic refinement is defined in terms of these state transformations. A state transformation f is a refinement of a state transformation g if they have the same initial and final state spaces and $f(s) \subseteq g(s)$ for every initial state s. Note that if $\perp \in g(s)$ for some s, then $f(s)$ can be anything at all. In other words we can correctly refine an "undefined" program to do anything we please. If f is a refinement of g (equivalently, g is refined by f) we write $g \leq f$. A *structure* for a logical language $\mathcal{L}$ consists of a set of values, plus a mapping between constant symbols, function symbols and relation symbols of $\mathcal{L}$ and elements, functions and relations on the set of values. A model for a set of sentences (formulae with no free variables) is a structure for the language such that each of the sentences is interpreted as true. If the interpretation of statement $\mathbf{S}_1$ under the structure M is refined by the interpretation of statement $\mathbf{S}_2$ under the same structure, then we write $\mathbf{S}_1 \leq_M \mathbf{S}_2$. If this is true for every model of a countable set Δ of sentences of $\mathcal{L}$ then we write $\Delta \models \mathbf{S}_1 \leq \mathbf{S}_2$.

3.3 Proof-Theoretic Refinement

Given two statements $\mathbf{S}_1$ and $\mathbf{S}_2$, and a formula $\mathbf{R}$, we have the two formulae $\mathrm{WP}(\mathbf{S}_1, \mathbf{R})$ and $\mathrm{WP}(\mathbf{S}_2, \mathbf{R})$. If there exists a proof of the formula $\mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R})$ using the set Δ as assumptions, then we write $\Delta \vdash \mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R})$. For $\mathbf{S}_2$ to be a refinement of $\mathbf{S}_1$, this result has to hold for every postcondition $\mathbf{R}$. We can avoid the need for quantification over formulae, and remain in first order logic, by extending the language $\mathcal{L}$ by adding a new relation symbol $G(\mathbf{w})$ where $\mathbf{w}$ is a list of all the free variables in $\mathbf{S}_1$ and $\mathbf{S}_2$. If we can prove $\Delta \vdash \mathrm{WP}(\mathbf{S}_1, G(\mathbf{w})) \Rightarrow \mathrm{WP}(\mathbf{S}_2, G(\mathbf{w}))$ in the extended language $\mathcal{L}'$ then the proof makes no assumptions about $G(\mathbf{w})$ and is therefore still valid when $G(\mathbf{w})$ is replaced by any other formula. In this case we write $\Delta \vdash \mathbf{S}_1 \leq \mathbf{S}_2$.

A fundamental result, proved in [20] which generalises a theorem in [2] is that these two notions of refinement are equivalent. More formally:

Theorem 3.1 *For any statements $\mathbf{S}_1$ and $\mathbf{S}_2$, and any countable set Δ of sentences of $\mathcal{L}$:*

$$\Delta \models \mathbf{S}_1 \leq \mathbf{S}_2 \iff \Delta \vdash \mathbf{S}_1 \leq \mathbf{S}_2$$

These two equivalent definitions of refinement give rise to two very different proof methods for proving the correctness of refinements. Both methods are exploited in [24]—weakest preconditions and infinitary logic are used to develop the induction rule for recursion and the recursive implementation theorem, while state transformations are used to prove the representation theorem.

4 The Architecture of ReForm

The tool consists of a structure editor, a library of proven transformations and a knowledge-based system which analyses the programs and specifications under consideration and uses heuristic knowledge to determine which transformations will achieve a given end (for example, deriving the specification of a section of code, finding the most suitable technique for recursion removal, optimising for efficiency etc.)

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 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 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. An interactive knowledge base to suggest transformations in a given situation will be built in to the system at a later stage.

The system will use knowledge based heuristics to analyse large programs and suggest suitable transformations as well as carrying out the transformations and checking the applicability conditions. 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. 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 example below) with a very high degree of confidence in the correctness of the result. An additional benefit of this formal link between specification and code is in the application to safety-critical systems. Such systems can be developed by transforming high-level specifications down to efficient low level code with a very high degree of confidence that the code correctly implements every part of the specification. There are also applications to the reuse of software—both specification, code, and development history can be stored in a repository and whenever a similar specification needs to be implemented the code and/or development history can be re-used. See [26] for more details.

The WSL language has been developed over the last eight years in parallel with the development of the transformation theory and proof methods. Over this time the language has developed from a simple and tractable kernel language to a complete and powerful programming language.

The WSL Kernel

Primitive Statements:

1. Assertion: $\{\mathbf{P}\}$
2. Guard: $[\mathbf{P}]$
3. Add some variables: $add(\mathbf{x}_)$
4. Remove some variables: $remove(\mathbf{x}_)$

Compound Statements:

1. **Sequential Composition**: $(\mathbf{S}_1;\ \mathbf{S}_2)$
   First $\mathbf{S}_1$ is executed and then $\mathbf{S}_2$
2. **Choice:** $(\mathbf{S}_1 \sqcap \mathbf{S}_2)$
   One of the statements $\mathbf{S}_1$ or $\mathbf{S}_2$ is chosen for execution
3. **Recursive Procedure:** $(\mu X.\mathbf{S}_1)$
   Within the body $\mathbf{S}_1$, occurrences of the statement variable X represent recursive calls to the procedure.

WSL Language Extensions

- Dijkstra's Guarded Command Language;
- **while** loops;
- Loops with multiple **exit**s;
- Mutually recursive procedures (labels and **goto**s);
- Local variables;
- Procedures and functions with parameters;
- Expressions with side-effects;
- Assembler language.

At the "low-level" end of the language there exists an automatic translator from IBM Assembler into WSL. At the "high-level" end it is possible to write high-level, abstract specifications, similar to **Z** and VDM specifications [9,12].

4.1 Main features of the ReForm tool

- Source code is translated into WSL, then automatically restructured and simplified;
- Transformations are written in an extension of WSL called Meta-WSL;
- The tool validates transformation choice and offers a menu of valid transformations according to the context;
- A LISP engine carries out the transformations and records the history;
- Documentation and comments can be attached to the code ;
- Edits and modifications are recorded in the history;
- An X Windows front end displays a pretty-printed version of the current program;
- The system calculates various metrics (McCabe, structural complexity, size) to monitor progress and quality.

4.2 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 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 state 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, (such as IBM's CICS product) 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.

5 Results of using ReForm

Experiments on small but complex programs have given very encouraging results: we have been able to discover bugs in high-level language code which were revealed by the analysis process. We have also discovered a performance hit in an existing, large scale, heavily used bit of Assembler code. This was introduced as a result of maintenance, and the maintenance programmers became aware of it when they examined the transformed version of the assembler code. The same transformations have been used to derive several types of algorithm from high-level, abstract specifications [21,23,25,27].

We have recently completed a case study involving a number of modules of IBM Assembler, each consisting of up to 20,000 lines of code, taken from a large commercial system. Each module was automatically translated into WSL and interactively restructured into a high-level language form. One particular module had been repeatedly modified over a period of many years until the control flow structure had become highly convoluted. Using the prototype tool we were able to transform this into a hierarchy of (single-entry, single-exit) subroutines resulting in a module which was slightly shorter and considerably easier to read and maintain. The transformed version was hand-translated back into Assembler which (after fixing a single mis-translated instruction) "worked first time". A typical result for one of the smaller modules is shown below:

Stage	lines	McCabe	Structural	Size
1	2,330	1,030	48,175	24,736
2	1,381	245	17,021	8,404
3	1,227	156	11,990	7,120

Stage 1 is the translated WSL code, stage 2 is after automatic restructuring and simplification, and stage 3 is after a small amount of interactive transformation.

The prototype system implements over 600 transformations, arranged in a set of 10 transformation menus. When a menu is invoked, the system checks the validity of each transformation in that class and constructs a list of the valid ones. This means that the user only ever sees valid transformations in the menus, which reduces the selection problem and eliminates confusion. The prototype consists of about 80,000 lines of Common LISP and WSL source code (implementing the structure editor and transformation engine), and about 22,000 lines of C (implementing the pretty-printer and X Windows front end). The transformation engine and front end act as separate processes, communicating via ASCII commands. This means that they can, and frequently do, run on separate machines: for example with the transformation engine running on a mainframe or fileserver and the interface running on a workstation, PC, Macintosh or X terminal. The Common LISP code has been written with portability in mind: in fact it has to run on two different LISP implementations, running on three different CPUs.

The prototype has been developed using a "rapid prototyping" method, this is so that new ideas can be implemented and tested quickly. Over the course of the development, the internal data structures and organisation have been changed several times as new research has shown better ways of doing things. One of the drawbacks of rapid prototyping is that the resulting tool can become unstable: i.e. bug-ridden, poorly-structured and difficult to maintain. To minimise this problem and maximise the flexibility of the tool we developed an "abstract machine" implementation, with formally defined interfaces between the implementation of the abstract machines and the rest of the system. The major components in the system are:

- Internal representation of WSL code;
- Structure editor;
- Transformation library;
- X Windows interface.

Each of these is implemented as an abstract machine with formally defined interfaces. This means that different people can work on reimplementing the different modules without causing integration problems. Another technique we have used is to develop a comprehensive regression test suite in parallel with the development of the system. This has acted as a "trip test": each new version has to pass the test before it is released, and this has prevented many of the problems which can occur with a rapid turnaround of program versions.

5.1 A Method for Reverse Engineering

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. This is the stage we illustrate in this paper. It involves the automatic and manual application of various transformations to restructure the system and express it at increasingly higher levels of abstraction. We do this 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 [3, 13] since it takes note of both data flow and control flow, and includes both syntactic and semantic transformations [1]. 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 [22] below for an example of this process; it is used extensively in [27]. 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.

6 Conclusions

The ReForm project has been highly successful to date, producing a reverse-engineering tool based on a rigorous theoretical foundation, which is already capable of producing useful results for real "spaghetti" assembler modules.

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

- 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.

7 Current Research

Work is currently underway in the following areas:

- Extension of the tool to high-level transformations (to produce abstract specifications from code). These transformations exist but have yet to be fully implemented in the prototype tool;
- More sophisticated data-flow analysis;
- Extension of the theory to communicating parallel programs;
- Extensions to deal with real-time and interrupt-driven programs [30];
- The use of metrics, including size and complexity metrics, to automate more of the transformation process and guide the selection of transformations.

We are also currently working on some more extensive case studies involving professional assembler programmers working on real assembler code. These will attempt to quantify the improvements in maintainability achievable through inverse engineering.

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] J. Arsac, "Syntactic Source to Source Program Transformations and Program Manipulation," *Comm. ACM* 22 (Jan., 1982), 43–54.

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

[3] F. W. Calliss, "Problems With Automatic Restructurerers," Durham University, Technical Report, 1989.

[4] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[5] M. S. Feather, "A Survey and Classification of Some Program Transformation Techniques," *Program Specification and Transformation* (1987).

[6] J. R. Foster, "Program Lifetime: A Vital Statistic for Maintenance," *Conference on Software Maintenance 15th–17th October 1991*, Sorrento, Italy (Oct., 1991).

[7] J. R. Foster & H. P. Kiekuth, "Software Maintenance Survey: Summary," *Technical Report* (Mar. 1990).

[8] C. A. R. Hoare, I. J. Hayes, H. E. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey & B. A. Sufrin, "Laws of Programming," *Comm. ACM* 30 (Aug., 1987), 672–686.

[9] C. B. Jones, *Systematic Software Development using VDM*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[10] C. R. Karp, *Languages with Expressions of Infinite Length*, North-Holland, Amsterdam, 1964.

[11] B. Lientz & E. B. Swanson, *Software Maintenance Management*, Addison Wesley, Reading, MA, 1980.

[12] M. A. McMorran & J. E. Nicholls, "Z User Manual," IBM UK Laboratories Ltd., TR12.274, Hursley Park, July, 1989.

[13] J. C. Miller & B. M. Strauss, "Implications of Automatic Restructuring of COBOL," *SIGPLAN Notices* 22 (June, 1987), 76–82.

[14] R. Moreton, "Analysis and Results from a Maintenance Survey," *Second Software Maintenance Workshop Notes*, Centre for Software Maintenance, University of Durham (1988).

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

[16] C. C. Morgan, "The Specification Statement," *Trans. Programming Lang. and Syst.* 10 (1988), 403–419.

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

[18] J. T. Nosek & P. Palvia, "Software Maintenance Management: Changes in the Last Decade," *J. Software Maintenance: Research and Practice* 2 (Sept. 1990), 157–174.

[19] C. T. Sennett, "Using Refinement to Convince: Lessons Learned from a Case Study," *Refinement Workshop, 8th–11th January, Hursley Park, Winchester* (Jan., 1990).

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

[21] M. Ward, "Derivation of a Sorting Algorithm," Durham University, Technical Report, 1990.

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

[23] M. Ward, "The Largest True Square Problem—An Exercise in the Derivation of an Algorithm," Durham University, Technical Report, Apr., 1990.

[24] M. Ward, "Specifications and Programs in a Wide Spectrum Language," Submitted to J. Assoc. Comput. Mach., Apr., 1991.

[25] M. Ward, "Iterative Procedures for Computing Ackermann's Function," Durham University, Technical Report 89-3, Feb., 1989.

[26] M. Ward, "Using Formal Transformations to Construct a Component Repository," in *Software Reuse: the European Approach*, Springer-Verlag, New York–Heidelberg–Berlin, Feb., 1991.

[27] M. Ward, "Derivation of Data Intensive Algorithms by Formal Transformation," Submitted to IEEE Trans. Software Eng., May, 1992.

[28] M. Ward, "A Model for Partial Programs," Submitted to J. Assoc. Comput. Mach., Nov., 1989.

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

[30] E. J. Younger & M. Ward, "Inverse Engineering a simple Real Time program," *Submitted to J. Software Maintenance: Research and Practice*, New York, NY (Oct., 1992).