Abstracting a Specification from Code

Martin Ward
Computer Science Dept
Science Labs
South Rd
Durham DH1 3LE

July 16, 1993

Abstract

Much of the work on developing program transformation systems has concentrated on systems to assist in program development. However, the four separate surveys carried out between 1977 and 1990 [18,20,22,24], summarised in [17], show that between 40% and 60% of all commercial software effort is devoted to software maintenance rather than the development of new systems. In this paper we describe a joint project between the University of Durham and CSM Ltd to develop a method and tool for reverse engineering and software maintenance based on program transformation theory. We present an example which illustrates how such a tool can extract a high-level abstract specification from the low-level source code of a program by a process of formal program transformation based on a theory of program equivalence [27]. All the code-level reverse engineering of the example program was carried out on the prototype tool with the resulting code pasted directly into the paper.

1 Introduction

Four separate surveys carried out between 1977 and 1990 [18,20,22,24] and summarised in [17], 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. In this paper we describe a formal method for reverse engineering existing code which uses 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. We do not consider timing constraints in this paper: although the method can be extended to model time as an extra output of a program. A "sufficiently precise" description is one which can be expressed in first order logic and set theory: this includes **Z**, VDM [19], and all other formal specification languages.

The method uses a Wide Spectrum Language (called WSL), developed in [27,30,34] 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 [12] 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. In [28,29,31,33] program transformations are used to derive a variety of efficient algorithms from abstract specifications. In this paper we apply the transformations in the reverse direction: starting with a small but tangled and obscure program we are able to use transformations to restructure the program and derive a concise abstract representation of its specification. The transformation process also reveals a bug in the program which is difficult to spot in the original version but trivial to spot and correct in the transformed version.

1.1 Transformational Development

Producing a program (or a large part of it) from a specification in a single step is a difficult task to carry out, to survey and to verify [6]. Moreover, programmers tend to underestimate the complexity of given problems and to overestimate their own mental capacity [25] and this exacerbates the situation further.

A solution in which a program is developed incrementally by *stepwise refinement* was proposed by Wirth [37]. However, the problem still remains that each step is done intuitively and must then be validated to determine whether the changes that have been made preserve the correctness of the program with respect to some specification, yet do not introduce unwanted side effects.

The next logical stage, improving on stepwise refinement, is to only allow provably semantic-preserving changes to the program. Such changes are called *transformations*. There are several distinct advantages to this approach [6]:

- The final program is correct (according to the initial specification) *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 can be supported by the computer. A significant part of transformational programming involves the use of a large number of small changes to be made to the code. Performing such changes by hand would introduce clerical errors and the situation would be no better than the original *ad hoc* methods. However, such clerical work is ideally suited to automation, allowing the computer itself to carry out the monotonous part of the work, allowing the programmer to concentrate on the actual design decisions.

1.2 Transformation Systems

Many workers have recognised that developing a program by successive transformation can be made much easier and less error-prone if an interactive system is provided which can carry out the transformations, perhaps check that they are used in a valid way, and keep a record of the various versions of the program. Thus, there has been much research into transformational programming and this has resulted in a large number of experimental systems. For a detailed overview of these see the papers by Partsch and Steinbrügen [25] and Feather [12].

The three main types of transformation system are:

1. A *manual system* makes the user responsible for every single transformation step. It is the simplest implementation and the system must provide some means for building up compact and powerful transformation rules.
2. A *fully automatic system* enables the selection and appropriate rules to be completely determined by the system using built-in heuristics, machine evaluation of different possibilities, or other strategic consideration.
3. A *semi-automatic system* works both autonomously for predefined subtasks and manually for unsolvable problems.

For such systems there are two main ways of organising the transformations: The first is as an extensible catalogue of specific transformations, the second is to have a small "generative set" of very simple transformations which are combined in various ways to provide more powerful manipulations.

The Cornell Program Synthesiser of [5,26] can be thought of as a totally manual system. It is an interactive system for program writing and editing which acts directly on the *structure* of the program by inserting and deleting structures in such a way as to ensure that the edited program is always syntactically correct: used as a transformation system, the user would be responsible for the semantic correctness of the manipulations. Arsac [1] describes using a simple manual system to carry out transformations of a program and store the various versions. His system knows some transformations but makes no attempt to check that the correctness conditions of a transformation hold when it is applied.

The first work on automatic program transformation was done by Burstall and Darlington in the mid-1970's [9]. Their first system was based on a schema-driven method for transforming applicative recursive programs into imperative ones: with the ultimate goal of improved efficiency. The system worked largely automatically, according to a set of built-in rules, with only a small amount user control. The rules were simple transformations, including recursion removal, elimination of redundant computations, unfolding and structure sharing.

Their second system, implemented in POP-2 and designed to manipulate applicative programs, is a typical representative of the generative set approach and consists of only six rules: *definition, instantiation, unfolding, folding, abstraction, and laws* (actually a set of data-structure-specific rules). "Definition" allows the introduction of the new functions (in the form of recursion equations).

Balzer built a program transformation system in the early 1980's [3]. The system used a separate specification language GIST, rather than a single wide-spectrum language.

CIP-S is the approach of the Project CIP (Computer-aided, Intuition-guided Programming). The objective is to develop an integrated environment for the transformational development of programs from algebraic specifications. This includes manipulation of concrete programs, derivation of new transformational rules within the system, transformation of algebraic types, and verification of applicability conditions, and the documentation of developments and their manipulation.

CIP-L [Bauer 85] is the language on which the CIP project was based. CIP-L is a wide-spectrum language which includes constructs for writing high-level specifications, functional programs, imperative programs and unstructured programs with **goto**s. The language provides constructs for the specification and implementation of data structures and control structures. Algebraic data types are implemented by computation structures combining data and algorithms. Modes are described by specific types for which computation structures can be provided automatically. Based on algebraic types and/or computation structures, program can be specified using predicate logic, description, comprehensive choice, and fully typed set operations.

The DRACO System is a general mechanism for software construction based on the paradigm of "reusable software". "Reusable" here means that the analysis and design of some library program can be reused, but not its code. DRACO is an interactive system that enables a user to refine a problem, stated in a high level problem domain specific language, into an efficient LISP program.

Accordingly, DRACO supplies mechanisms for defining problem domains as special purpose domain languages and for manipulating statements in these languages into an executable form.

Another automatic system, the DEDALUS system (DEDuctive ALgorithm Ur-Synthesizer) by Manna and Waldinger is implemented in QLISP. Its goal is to derive LISP programs automatically and deductively from high-level input-output specifications in a LISP-like representation of mathematical logic notation. A goal-directed deductive approach is used whereby the reduction of a goal (to synthesize a program satisfying a given specification) to one or more subgoals, by means of a transformation rules, results in the generation of a program fragment which computes the desired result, once it is completed with program fragments corresponding the subgoal(s). So, for example, reducing a goal to two subgoals by means of a case analysis corresponds to the introduction of a conditional expression.

The long-running SETL project at the Courant Institute of New York University [11] has served as the context for a wide variety of transformation research. Their *very high level programming language*, SETL, has syntax and semantics based on standard mathematical set theory. SETL programs can always be executed; however, naïve execution of programs that make liberal use of the high-level language features may be very inefficient. The SETL compiler has been built to compile SETL programs into efficient interpretable code or machine code. Used in this manner, the SETL compiler would fall into the category of a traditional compiler, albeit a very sophisticated one.

Boyle's TAMPR (Transformation-Assisted Multiple Program Realization) system provides a variety of support for programming in FORTRAN at the Argonne National Laboratory [7]. Applications include small *language extensions* (e.g., complex and quaternion abstract data types, automatic declaration of undeclared variables), *optimisations* (e.g., loop unrolling, unfolding of some subroutine calls), *conversions* (e.g., single to double precision, multi-dimensional arrays to one- dimensional ones), and *miscellaneous support* (e.g., instrumenting programs, recognising inherent program structure). The modest nature of the tasks enables TAMPR's transformation process to be entirely automatic. In addition to transformation within the FORTRAN language, TAMPR has also been applied to help in FORTRAN-to-PASCAL translation, and in converting the bulk of the TAMPR system itself from its (almost) pure applicative LISP version into FORTRAN (which runs faster the compiled LISP form on the same machine). This latter application demonstrates the feasibility of the approach on moderately large programs (1300 lines, 42 functions, converted into 3000 lines of FORTRAN). Boyle stresses that approaching these tasks by means of program transformation encourages organising it in a modular fashion, with many consequent benefits.

Feather's ZAP system and language [13] is based on the fold/unfold work of Burstall and Darlington on transforming applicative programs expressed in recursion equations. The ZAP system's language is a *language for expressing transformations and developments.*

1.3 Automating the Process

There are currently three main ways of automating more of the transformation process (see [16]):

Jittering: The method used in the Transformational Implementation (TI) system developed by Balzer [3], and also in Fickas' GLITTER system [15] is that of *jittering.* In this system, if a transformation is applied, but fails due to some minor technical detail, the system automatically modifies the program (using transformations) so that the initial transformation can succeed.

Means-end analysis: A variant of jittering, called means-end analysis, is used by Mostow [23] to guide rule selection. The user provides the pattern to be matched in order to apply the rule, and the system computes the *difference* between this and the actual current pattern. The computed difference then indexes further rules which could be used to reduce the difference.

Optimal "Next" Transformations: In this approach, the system is tried manually on many different programs and the order of transformations used is recorded; this is a *knowledge elicitation* process (and will also be used in the next approach to determine what metrics to use). From these results it will be possible to determine which transformations form sequences and to make suggestions as to the next transformation to use based on the previous one. For example, removing a redundant variable may follow merging two assignments.

The Metric Approach: The final approach is, perhaps, the most ambitious. This is to determine a metric which quantifies the "ease of understanding", or "niceness", of a program and uses "hill-climbing" methods to find a sequence of transformations which manipulate the program into an equivalent form which maximises this metric.

Note that total automation is extremely difficult and probably undesirable: the best approach is an interactive system which is highly automated in some areas (eg restructuring).

1.4 The Maintainer's Assistant

Much of the current research in program transformation systems is directed purely at software development and has little applicability to software maintenance, including all the systems discussed above, where the code to be analysed often has little structure, and was certainly not developed in accordance with the rules of any particular transformational development system. One method of performing maintenance which was suggested by Balzer [4], that of modifying the code's specification and then reimplementing it formally, seems well suited to the transformational method. But, for old code, no specification may be available, so we have nothing we can edit and re-transform to produce a new version of the program. This has led to the work described in this paper which using program transformations, aims among other things, to help the maintainer recover specifications from code. A major aim of this work is the development of a tool, the *Maintainer's Assistant*, which will automate much of the process of transforming code into specifications and specifications into code. The 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. Ultimately we hope to capture much of the knowledge and expertise that we have developed over the course of several case studies, and incorporate it within the tool itself.

The Maintainer's Assistant 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** [19]. 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, this makes it possible to use the system to maintain its own code.

The initial prototype Maintainer's assistant was developed as part of an Alvey project at the University of Durham [35,36] 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 (ranging up to 5500 lines) with

very encouraging results (see Section 5).

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 [32] for more details.

2 A Method for Reverse Engineering

The method we have developed for reverse engineering a system is based on *inverse engineering*, which is the process of extracting high-level abstract specifications from source code using formal program transformations. The benefits of this formal approach apply to maintenance generally, as well as the specific reverse engineering task. These are:

- Increased reliability: bugs and inconsistencies are easier to spot;
- Formal links between specification and code can be maintained;
- Maintenance can be carried out at the specification level;
- Large restructuring changes can be made to the program with the confidence that the functionality is unchanged;
- Programs can be incrementally improved—instead of being incrementally degraded!
- Data structures and the implementations of abstract data types can be changed easily.

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 [10,21] since it takes note of both data flow and control flow, and includes both syntactic and semantic transformations [2]. We have however had considerable success with automating the simpler restructuring transformations, by implementing heuristics elicited from experienced program transformation users. See section 4.1.
   (b) Analyse the resulting structures to determine suitable higher-level data representations and control structures. In the example below we determine that the double-nested loop is treating the input sequence as a sequence of subsequences.
   (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 section 4.2 below for a small example of this process; it is used extensively in [33]. In the example below we represent the input sequence as a sequence of sequences and this allows us to express the inner loop as a single statement. This in turn enables us to collapse the outer loop to a single statement. 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.

3 Example Transformations

3.1 Theoretical Foundation

A transformation is an operation which maps any program satisfying the applicability conditions of the transformation to an "equivalent" program. Equivalence is defined in terms of the external "black box" behaviour (or semantics) of the program. We define the semantics of a program to be a function which maps from an initial state to a final set of states: this abstracts away from all the internal operations of the program. The set of final states represents all the possible output states of the program for the given input state. Using a set of states enables us to model nondeterministic programs and partially defined (or incomplete) specifications. See [27] and [30] for a description of the semantics of WSL and the methods used for proving the correctness of transformations.

3.2 Notation

Sequences: $s = \langle a_1, a_2, \ldots, a_n \rangle$ is a sequence, the ith element a_i is denoted $s[i]$, $s[i\,..\,j]$ is the subsequence $\langle s[i], s[i+1], \ldots, s[j] \rangle$, where $s[i\,..\,j] = \langle\rangle$ (the empty sequence) if $i > j$. The length of sequence s is denoted $\ell(s)$, so $s[\ell(s)]$ is the last element of s. We use $s[i\,..]$ as an abbreviation for $s[i\,..\,\ell(s)]$.

Sequence concatenation: $s + t = \langle s[1], \ldots, s[\ell(s)], t[1], \ldots, t[\ell(t)] \rangle$.

Subsequences: The assignment $s[i\,..\,j] := t[k\,..\,l]$ where $j - i = l - k$ assigns s the value $\langle s[1], \ldots, s[i-1], t[k], \ldots, t[l], s[j+1], \ldots, s[\ell(s)] \rangle$.

Sets: We have the usual set operations $\cup$ (union), $\cap$ (intersection) and $-$ (set difference), $\subseteq$ (subset), $\in$ (element), $\wp$ (powerset). $\{\, x \in A \mid P(x) \,\}$ is the set of all elements in A which satisfy predicate P. For the sequence s, $set(s)$ is the set of elements of the sequence, i.e. $set(s) = \{\, s[i] \mid 1 \leqslant i \leqslant \ell(s) \,\}$.

Relations and Functions: A relation is a (finite or infinite) set of pairs, a subset of $A \times B = \{\, (a,b) \mid a \in A \wedge b \in B \,\}$ where A is the domain and B the range. A relation f is a function if $\forall x, y_1, y_2 .\, \Big(\big((x, y_1) \in f \;\wedge\; (x, y_2) \in f\big) \Rightarrow y_1 = y_2\Big)$. In this case we write $f(x) = y$ when $(x, y) \in f$. We write $f \cdot g$ for the composition of functions or relations. $(f \cdot g)(x) = f(g(x))$.

Currying: If $\oplus$ is a binary operator and a and b are values, then $(\oplus)$, $(a\oplus)$ and $(\oplus b)$ are functions with $(\oplus)(a) = (a\oplus)$, $(a\oplus)(y) = a \oplus y$ and $(\oplus b)(x) = x \oplus b$.

Constant Functions: K_a is the constant function with value a, $\mathrm{K}_a(x) = a$ for any x. An identity element of $\oplus$ is denoted $\mathrm{id}_\oplus$. The function $\langle\cdot\rangle$ maps any value to the corresponding singleton sequence: $\langle\cdot\rangle(x) = \langle x \rangle$.

Map: The map operator $*$ returns the sequence obtained by applying a given function to each element of a given sequence: $(f * \langle a_1, a_2, \ldots, a_n \rangle) = \langle f(a_1), f(a_2), \ldots, f(a_n) \rangle$.

Reduce: The reduce operator / applies an associative binary operator to a list and returns the resulting value: $(\oplus/\langle a_1, a_2, \ldots, a_n \rangle) = a_1 \oplus a_2 \oplus \cdots \oplus a_n$. So, for example, if s is a list of integers then $+/s$ is the sum of all the integers in the list, if q is a list of lists then $+/(\ell * q) = \ell(+\!\!+/q)$ is the total length of all the lists in q.

Projection: The projection functions π_1, π_2, ... are defined as $\pi_1(\langle x, y \rangle) = x$, $\pi_2(\langle x, y \rangle) = y$, and more generally, for any sequence s: $\pi_i(s) = s[i]$.

The operation of splitting a sequence into a sequence of non-empty sections at some point where a predicate fails is generally useful so we will define the following notation:

Suppose we have a sequence p which we want to split into sections at those points i where the predicate $B(p[i], p[i+1])$ is false, i.e. we want to define a new sequence of non-empty sequences q such that the concatenation of the sequences in q is equal to p ($+\!\!+/q = p$) and B is true within each section and false from one section to the next.

Define the function $index_q : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ by $index_q(j,k) = +/(\ell * q[1\,..\,j-1]) + k$. This function maps a position in the q structure into the corresponding position in the p structure, i.e. for all $j \in 1\,..\,\ell(q)$ and $k \in 1\,..\,\ell(q[j])$ we have $p = +/q \Rightarrow p[index_q(j,k)] = q[j][k]$. On this domain, $index_q$ is 1–1, so it has a well-defined inverse. This inverse $index_q^{-1}$ maps an index i of p to a pair $\langle j,k \rangle$ such that $p[i] = q[j][k]$. So the function $section_q = \pi_1 \cdot index_q^{-1}$ will give the section in q in which an element of p occurs.

With this notation, we can define a split function $split(p,B) = q$ which splits p into non-empty sections with the section breaks occurring between those pairs of elements of p where B is false. The formal definition uses $section_q$ to find the "section breaks":

Definition 3.1 *$split(p,B) = q$ where:*

$$\begin{aligned}(+/q) = p \;\wedge\; \langle\rangle \notin set(q)\\ \wedge\; \forall i \in 1\,..\,\ell(p) - 1.\, \Big((B(p[i], p[i+1]) \Rightarrow section_q(i+1) = section_q(i))\\ \wedge\; (\neg B(p[i], p[i+1]) \Rightarrow section_q(i+1) = section_q(i) + 1) \Big)\end{aligned}$$

3.3 Examples of Transformations

In this section we describe a few of the transformations we will use later:

3.3.1 Expand IF statement

The **if** statement:

$$\textbf{if B then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi; } \mathbf{S}$$

can be expanded over the following statement to give:

$$\textbf{if B then } \mathbf{S}_1\textbf{; S else } \mathbf{S}_2\textbf{; S fi}$$

3.3.2 Loop Inversion

If the statement $\mathbf{S}_1$ contains no **exit**s which can cause termination of an enclosing loop (i.e. in the notation of [27] it is a *proper sequence*) then the loop:

$$\textbf{do } \mathbf{S}_1\textbf{; } \mathbf{S}_2 \textbf{ od}$$

can be inverted to:

$$\mathbf{S}_1\textbf{; do } \mathbf{S}_2\textbf{; } \mathbf{S}_1 \textbf{ od}$$

This transformation may be used in the forwards direction to move the termination test of a loop to the beginning, prior to transforming it into a **while** loop, or it may be used in the reverse direction to merge two copies of the statement $\mathbf{S}_1$.

3.3.3 Action Systems

We use "Actions" (parameterless procedures [1,2]) to represent labels and **goto**s, an *action system* is a collection of mutually recursive actions. Within an action system, a call to the special procedure Z causes immediate termination of the whole system: any statements pending a procedure return will not be executed if Z is called. A *regular* action system is one in which the execution of any action always leads to an action call. In this case the whole system can only be terminated by a call to Z; so no action call can return in a regular action system.

Within an action system, any call to an action can be replaced by a copy of the body of that action. This is called "unfolding" and applies equally well to a recursive call within the body of the action. The inverse transformation, i.e. folding, can be applied in any case where it results in the opposite effect to an unfolding operation. This prevents pathological cases where, for example, the body of the action: $P \equiv \mathbf{S.}$ is "folded" to get: $P \equiv \textbf{call } P\textbf{.}$ This case is invalid, but folding *can* be applied to (for example) $P \equiv \mathbf{S}[\mathbf{S}/\textbf{call } P]\textbf{.}$ to get $P \equiv \mathbf{S.}$

4 The Program

The program is taken from a programming textbook [14], it originally took its input from a database file, in translating it to WSL we have represented the file by a pair of arrays, *item* and *number*. The procedure *INHERE* was originally a label in the middle of an **if** statement in the middle of a loop! This loop is represented by procedure L below:

```
var ⟨m := 0, p := 0, last := " "⟩:
    actions PROG:
    PROG ≡
      ⟨line := " ", m := 0, i := 1⟩; call INHERE.
    L ≡
      i := i + 1;
      if i = n + 1 then call ALLDONE fi;
      m := 1;
      if item[i] ≠ last
        then write(line); line := " "; m := 0; call INHERE fi;
      call MORE.
    INHERE ≡
      p := number[i]; line := item[i]; line := line + " " + p; call MORE.
    MORE ≡
      if m = 1 then p := number[i]; line := line + ", " + p fi;
      last := item[i]; call L.
    ALLDONE ≡
      write(line); call Z. endactions end
```

This and subsequent versions of the program code were generated by the prototype tool in the form of LaTeX source files which were inserted in the paper.

4.1 Restructuring Transformations

In the first stages of simplifying and restructuring the program, little or no information is needed about the purpose of the program or its domain of operation. The simplifications work from the source code alone. This is important in maintenance applications where often the source code is the only reliable documentation in existence! In the later stages (deriving the specification) high-level domain information is used to guide the transformation process into giving a specification expressed in a usable form. This is because there are many ways of writing a correct specification of a given piece of code, some of which will be more useful than others. This "high-level" information includes information about the purpose and domain of the program (such as can be obtained from a user manual or discussions with the users of the program).

Although such information can be difficult, or impossible, to deduce from the source code alone, it is often readily available to the maintainer. We have found that this combination of semi-automatic and interactive operations is very powerful. The tedious low-level transformations and verifications can be carried out automatically while allowing the human to carry out the high-level analysis and structuring of the program. The results of this analysis can be recorded as documentation linked to the code which will be instantly available for later maintainers of the program. The maintainer uses high-level information (including hints gained from comments, variable names and other documentation) to guide the system in its selection of transformations. The automatic checking ensures that the correctness of the derivation is not compromised if the hints prove to be invalid: in these circumstances, failure to derive the expected structure provides valuable information as to the nature of the differences between the documentation and the source code. Such differences may be due to bugs in the software (which will be uncovered by the transformation process) or out-of-date documentation (which can now be updated to bring it in line with the code).

In the first stages we aim to restructure the program by removing procedures, moving flag tests closer to where the flag is set, introducing loops and merging identical code. All the transformations required at this stage have been implemented on the prototype system; the different versions of the program shown here were generated by the system and copied directly into the paper.

First we copy *INHERE* and *MORE* into *PROG* to move a test of m next to the place where it is set:

```
var ⟨m := 0, p := 0, last := " "⟩:
    actions : PROG :
    PROG ≡
      ⟨line := " ", m := 0, i := 1⟩;
      p := number[i];
      line := item[i];
      line := line ++ " " ++ p;
      if m = 1 then p := number[i]; line := line ++ ", " ++ p fi;
      last := item[i];
      call L.
    L ≡
      i := i + 1;
      if i = n + 1 then call ALLDONE fi;
      m := 1;
      if item[i] ≠ last then write(line); line := " "; m := 0; call INHERE fi;
      call MORE.
    INHERE ≡
      p := number[i]; line := item[i];
      line := line ++ " " ++ p; call MORE.
    MORE ≡
      if m = 1 then p := number[i]; line := line ++ ", " ++ p fi;
      last := item[i]; call L.
    ALLDONE ≡
      write(line); call Z. endactions end
```

Use the value of m in *PROG* to eliminate the subsequent test:

```
PROG ≡
  ⟨line := " ", m := 0, i := 1⟩;
  p := number[i];
  line := item[i];
  line := line ++ " " ++ p;
  last := item[i];
  call L.
```

We could continue unfolding action calls and introducing loops in this way. However, this whole process has been automated in a single transformation `Collapse_Action_System` which follows heuristics we have developed, selecting the sequence of transformations required to restructure a program. The result of this single transformation is as follows:

```
var ⟨m := 0, p := 0, last := " "⟩:
    ⟨line := " ", m := 0, i := 1⟩;
    p := number[i];
    line := item[i];
    line := line ++ " " ++ p;
    last := item[i];
    do i := i + 1;
```

```
        if i = n + 1 then write(line); exit(1) fi;
        m := 1;
        if item[i] ≠ last
          then write(line); line := " "; m := 0;
               p := number[i]; line := item[i];
               line := line ++ " " ++ p fi;
        if m = 1 then p := number[i]; line := line ++ ", " ++ p fi;
        last := item[i] od end
```

By absorbing the statement **if** $m = 1$ **then** ... **fi** into the preceding **if** statement we can eliminate the remaining test of m. m becomes a redundant variable and can be removed entirely from the program in a single transformation. This transformation also notices that p is redundant and removes it.

The resulting program has two copies of the statement $last := item[i]$, one outside the loop and the other at the end of the loop body. So "loop inversion" can be applied to give:

```
var last := " ":
    ⟨line := " ", i := 1⟩;
    line := item[i] ++ " " ++ number[i];
    do last := item[i];
       i := i + 1;
       if i = n + 1 then write(line); exit(1) fi;
       if item[i] ≠ last then write(line); line := item[i] ++ " " ++ number[i]
                         else line := line ++ ", " ++ number[i] fi od end
```

Now we have two copies of $line := item[i] \mathbin{+\!\!+} " " \mathbin{+\!\!+} number[i]$. We would like to apply loop inversion again, so we convert the single loop to a double loop and take the statement outside the inner loop:

```
var last := " ":
    ⟨line := " ", i := 1⟩;
    line := item[i] ++ " " ++ number[i];
    do do last := item[i];
          i := i + 1;
          if i = n + 1 then write(line); exit(2) fi;
          if item[i] ≠ last then write(line); exit(1)
                            else line := line ++ ", " ++ number[i] fi od;
       line := item[i] ++ " " ++ number[i] od end
```

Loop inversion can now be used on the outer loop.

The inner loop is terminated in two places: we would like to combine these so that there is only one exit from the loop. We would also like to merge the two copies of *write*(*line*). We can use the fact that $i = n + 1$ is true before the first *write*(*line*) and false before the second to move these statements outside the inner loop:

```
var last := " ":
    ⟨line := " ", i := 1⟩;
    do line := item[i] ++ " " ++ number[i];
       do last := item[i];
          i := i + 1;
          if i = n + 1 then exit(1) fi;
          if item[i] ≠ last then exit(1) else line := line ++ ", " ++ number[i] fi od;
       if i ≠ n + 1 then write(line) else write(line); exit(1) fi od end
```

(To achieve this transformation the prototype system required a "hint" which we gave by introducing the assertions $\{i = n+1\}$ before the first copy of *write*(*line*) and $\{i \neq n+1\}$ before the second. The system used these assertions to move the subsequent statements into an **if** statement outside the loop). Now, within the inner loop we can merge the two **if** statements so there is only one exit, we also re-arrange the **if** statement at the end of the outer loop:

```
var last := " ":
    ⟨line := " ", i := 1⟩;
    do line := item[i] ++ " " ++ number[i];
        do last := item[i];
            i := i + 1;
            if i = n + 1 ∨ item[i] ≠ last then exit(1) fi;
            line := line ++ ", " ++ number[i] od;
        write(line);
        if i = n + 1 then exit(1) fi od end
```

The variable *last* is assigned the value *item*[*i*], then *i* is incremented and *last* is tested. We can replace *last* by *item*[*i* − 1] in the expression and remove *last* from the program. (The system automatically recognised that *i* was incremented between the assignment to *last* and its use):

```
⟨line := " ", i := 1⟩;
do line := item[i] ++ " " ++ number[i];
    do i := i + 1;
        if i = n + 1 ∨ item[i] ≠ item[i − 1] then exit(1) fi;
        line := line ++ ", " ++ number[i] od;
    write(line);
    if i = n + 1 then exit(1) fi od
```

Finally we convert the inner loop to a **while** loop:

```
⟨line := " ", i := 1⟩;
do line := item[i] ++ " " ++ number[i];
    i := i + 1;
    while i ≠ n + 1 ∧ item[i] = item[i − 1] do
            line := line ++ ", " ++ number[i]; i := i + 1 od;
    write(line);
    if i = n + 1 then exit(1) fi od
```

The transformations have revealed the "true" structure of the program, which involves a double loop: the "true" structure of a program is a structure which closely matches the function and purpose of the program. This structure was uncovered by simply following certain heuristics (in this case a technique for merging similar statements) without needing to understand the purpose of the program. When we look at the function of the program, as described in the published documentation [14], we see that this double loop precisely captures what the program is intended to do. The program scans through a sorted file (here represented by the arrays *item* and *number*) consisting of words and page references. The outer loop scans through distinct items and for each distinct item the inner loop steps through the page references for that item. Writing the program as a single loop whose body must distinguish the two cases of a new item and a repeated item obscured the simple basic structure which has been revealed through transformations. This kind of transformation has important applications in program maintenance: the second version is far easier to understand and modify—there is only one copy of the statement which writes to the file, the flag *m* which was used to direct the control flow is not needed, and the variables *p* and *last* have also been eliminated. The transformation from first version to second used only general transformations which have been proved to work in all cases, and so could be applied without having to understand the program first.

With this knowledge of the program's purpose we can see that the transformations have also revealed a bug in the program: note that the outer loop has the test at the end (as in the usual **repeat** ... **until** loop), so the body of this loop is executed at least once. Recalling that the program reads an input file and produces some output which summerises the input, there is something rather odd about this structure! In fact, the program will not work correctly if presented with an empty file: from the documentation the program should produce *no* output for an empty input file, but this program's output consists of a single line composed from the contents of uninitialised data structures! This bug is not immediately obvious in the first version of the program. For the first version a typical "fix" would be to add a test for an empty file and a **goto** which jumps to a new label at the end of the program. In that case this "fix" is also typical in that it further obscures the program structure, increases the program length and increases the number of identifiers used. In contrast with this, to carry out the same modification to the second version we merely change the outer loop to a **while** loop. An alternative (and even less drastic) method of correcting the bug is to introduce the assertion $\{n \neq 0\}$ at the beginning of the program. This assertion states that the "empty input" case can be ignored. Using it we are able to *transform* the outer loop into a **while** loop. On removing the assertion we get a program which will work correctly for the $n = 0$ case and which is proven to be equivalent to the original program in all other cases. Thus we have fixed the bug and proved that we have broken nothing else in doing so.

```
var ⟨line := " ", i := 1⟩:
    while i ≠ n + 1 do
        line := item[i] ++ " " ++ number[i];
        i := i + 1;
        while i ≠ n + 1 ∧ item[i] = item[i − 1] do
            line := line ++ ", " ++ number[i]; i := i + 1 od;
        write(line) od end
```

It should be re-emphasised at this point that all the transformations used so far were carried out on the prototype Maintainer's Assistant tool and inserted directly into the paper (see [35] and [8] for a description of the prototype).

4.2 Data Refinement

This is about as far as we can get with transformations at the code level. The next stage involves moving to a higher level of abstraction. To do this we read through the program looking for a suitable control or data abstraction. The previous restructuring stages have made this task considerably easier. In this case we have a double loop scanning through a pair of arrays: this suggests a data refinement in which the arrays are represented as a sequence of sequences in such a way that one subsequence is processed for each iteration of the outer loop. In other words we need to restructure the data so that it reflects more closely the control structure of the program.

The two arrays *item* and *number* are treated in parallel, with only the first n elements of each being accessed. So it makes sense to express them as a single data structure; an n element sequence of pairs $p = \langle\langle item[1], number[1]\rangle, \langle item[2], number[2]\rangle, \ldots, \langle item[n], number[n]\rangle\rangle$. The function *pairs* takes two sequences of equal length and returns the corresponding sequence of pairs.

We now have a double loop which scans through the sequence p. Each step of the inner loop processes a single element of the sequence, so each execution of the inner loop processes a segment of the sequence. So the key to the data restructuring is to split the input sequence into sections such that the outer loop processes one segment per iteration. This is easily achieved with the *split* function defined above—the terminating condition on the inner loop provides the predicate on which to split. Define the predicate *same_head* by:

```
funct same_head(x, y) ≡
        x[1] = y[1].
```

Then the new variable q is introduced with the assignment: $q := split(p, same\_head)$.

We introduce q and its two index variables j and k to the program as ghost variables. j and k step through q as i steps through p: more formally we have the invariant:

$$i = +/(\ell * q[1\,..\,j-1]) + k$$

which is the same as $i = index_q(j,k)$. From this invariant and the relation $+/q = p$ we get the invariant: $p[i] = q[j][k]$. Adding these ghost variables to the program we get:

```
var ⟨line := " ", i := 1⟩:
    var ⟨q, j, k⟩:
        q := split(p, same_head); j := 1; k := 1;
        while i ⩽ ℓ(p) do
                line := p[i][1] ++ " " ++ p[i][2];
                i := i + 1;
                k := k + 1; if k > ℓ(q[j]) then j := j + 1; k := 1 fi;
                while i ⩽ ℓ(p) ∧ p[i][1] = p[i − 1][1] do
                        line := line ++ ", " ++ p[i][2]; i := i + 1;
                        k := k + 1; if k > ℓ(q[j]) then j := j + 1; k := 1 fi od;
                write(line) od end end
```

The next stage is to replace references to the concrete variables p and i by references to the new variables q, j and k using the invariants above. Then the concrete variables become ghost variables and can be removed from the program. Note that due to the structure of q the test $p[i][1] = p[i-1][1]$ is true as long as we are in the same section of p, i.e. as long as we have not just incremented j and reset k to 1. But this is the case exactly when $k \neq 1$. Also, if $i > \ell(p)$ in the inner loop we must have just incremented j (and k will be 1), so the whole test is equivalent to $k \neq 1$. We have:

```
var ⟨line := " "⟩:
    var ⟨q, j, k⟩:
        q := split(p, same_head); j := 1; k := 1;
        while j ⩽ ℓ(q) do
                line := q[j][k][1] ++ " " ++ q[j][k][2];
                k := k + 1; if k > ℓ(q[j]) then j := j + 1; k := 1 fi;
                while k ≠ 1 do
                        line := line ++ ", " ++ q[j][k][2];
                        k := k + 1; if k > ℓ(q[j]) then j := j + 1; k := 1 fi od;
                write(line) od end end
```

We want to show that the inner loop processes exactly one segment of q, to do this we need to change its termination condition to $k \leqslant \ell(q[j])$. The easiest way to do this is to convert the inner loop to a **do** ... **od** loop and absorb the **if** statement and increment of k to get:

```
do k := k + 1;
    if k > ℓ(q[j]) then j := j + 1; k := 1 fi;
    if k ≠ 1 then exit fi;
    line := line ++ ", " ++ q[j][k][2] od
```

If $k \geqslant \ell(q[j])$ at the beginning of this loop body then the **if** statement will be executed and the loop terminated. Conversely if $k < \ell(q[j])$ then it is certainly $\geqslant 1$ so after k is incremented, the **if** statement has no effect and the loop is not terminated (since now $k > 1$). So we can transform the inner loop into the following **while** loop:

```
while k < ℓ(q[j]) do
```

```
        k := k + 1;
        line := line ++ ", " ++ q[j][k][2] od;
j := j + 1; k := 1
```

The local variable k is only used in this loop (its value is always 1 outside the loop) so we can transform it into a **for** loop. Our program now looks like this:

```
var line := " ":
    var ⟨q, j⟩:
        q := split(p, same_head); j := 1;
        while j ⩽ ℓ(q) do
                line := q[j][1][1] ++ " " ++ q[j][1][2];
                for k := 2 step 1 to ℓ(q[j]) do
                    line := line ++ ", " ++ q[j][k][2] od;
                j := j + 1;
                write(line) od end end
```

where we have replaced the occurrences of k outside the **for** loop by 1.

$q[j]$ is a sequence of pairs, but in the inner **for** loop we only use the second element of each pair. So we can represent $q[j]$ by the sequence of second elements, i.e. the sequence $r = \pi_2 * q[j]$ where $\pi_2(\langle a, b\rangle) = b$ is a projection function, (this is another data refinement). With this abstraction the inner loop takes on the following form:

```
var r := π₂ * q[j]:
    line := line ++ r[1];
    for k := 2 step 1 to ℓ(r) do
        line := line ++ ", " ++ r[k] od end
```

This implements a "splice" function, it is equivalent to: $line := line \mathbin{+\!\!+} \big(sep(", ")/r\big)$ where sep is defined: $sep(s)(a, b) = a \mathbin{+\!\!+} s \mathbin{+\!\!+} b$. So we can re-write this as:

```
var r := π₂ * q[j]:
    line := line ++ (sep(", ")/r)
```

The program simplifies to:

```
var line := " ":
    var ⟨q, j⟩:
        q := split(p, same_head); j := 1;
        while j ⩽ ℓ(q) do
                line := q[j][1][1] ++ " " ++ (sep(", ")/(π₂ * q[j]));
                j := j + 1;
                write(line) od end end
```

Finally, this program simply applies the procedure *write* to a function of each element of the list q so we can implement it as a (procedure) map operation:

```
begin var q := split(pairs(item[1..n], number[1..n]), same_head):
            write * (process * q) end
where
  funct pairs(xs, ys) ≡
        if xs = ⟨⟩ then ⟨⟩
                   else ⟨⟨xs[1], ys[1]⟩⟩ ++ pairs(xs[2..], ys[2..]) fi.,
  funct process(xs) ≡
        xs[1][1] ++ " " ++ (sep(", ")/(π₂ * xs)).
```

end

To summarise this specification: We first translate the pair $\langle item[1\,..\,n], number[1\,..\,n]\rangle$ of lists into a list p of pairs. Then we split p into a list of sections q starting a new section at each point where the head of one pair differs from the head of the next pair. Finally, for each section we print the result of applying *process* to that section where *process* concatenates the head of the first pair (the item), a space and the list of second elements of the pairs (the numbers), separated by the string ", ".

This is now in the form of an abstract specification which defines the precise relationship between the input and output states.

5 Project Status and Future Directions

The techniques illustrated here are based on a formal theory of program refinement and equivalence which has been used to develop and prove a large catalogue of useful transformations [27]. 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 CICS 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 [28,29,31,33].

We have recently completed a case study involving five modules of IBM Assembler, each consisting of about 500 lines of code, taken from a large commercial banking 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".

Work is currently underway in the following areas:

- Extension of the tool to high-level transformations (to generate abstract specifications) and more sophisticated data-flow analysis;
- Extension of the theory to communicating parallel programs;
- 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, in conjunction with IBM UK Ltd, about to embark 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.

6 Conclusion

In this paper we have taken a small, but highly complex, program and transformed it into a concise logical specification, by applying general program transformations which have been proven to preserve the semantics. The formal method of inverse engineering, which forms the basis of the Maintainer's Assistant, would appear to have reached the stage where application to real maintenance tasks is now feasible.

7 References

[1] J. Arsac, "Transformation of Recursive Procedures," in *Tools and Notations for Program Construction*, D. Neel, ed., Cambridge University Press, Cambridge, 1982, 211–265.

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

[3] R. Balzer, "Transformational Implementation: An Example," *IEEE Trans. Software Eng.* 7 (Jan., 1981).

[4] R. Balzer, "A 15 Year Perspective on Automatic Programming," *IEEE Trans. Software Eng.* SE 11 (Nov., 1985), 1257–1267.

[5] R. Barstow, H. E. Shrobe & E. Sandwall, *Interactive Programming Environments*, McGraw-Hill, New York, NY, 1984.

[6] F. L. Bauer, B. Moller, H. Partsch & P. Pepper, "Formal Construction by Transformation—Computer Aided Intuition Guided Programming," *IEEE Trans. Software Eng.* 15 (Feb., 1989).

[7] J. M. Boyle, "LISP To FORTRAN—Program Transformation Applied," in *Program Transformation and Programming Environments* Report on a Workshop directed by F. L. Bauer and H. Remus, P. Pepper, ed., Springer-Verlag, New York–Heidelberg–Berlin, 1984, 199–222.

[8] T. Bull, "An Introduction to the WSL Program Transformer," *Conference on Software Maintenance 26th–29th November 1990, San Diego* (Nov., 1990).

[9] R. M. Burstall & J. A. Darlington, "A Transformation System for Developing Recursive Programs," *J. Assoc. Comput. Mach.* 24 (Jan., 1977), 44–67.

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

[11] R. B. K. Dewar, E. Schonberg & J. T. Schwartz, "Higher Level Programming: Introduction to the Use of the Set-theoretic Programming Language SETL," Courant Institute of Mathematical Science, New York University, Technical Report, New York, 1981.

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

[13] M. S. Feather, "A System for Assisting Program Transformation," *Trans. Programming Lang. and Syst.* 4 (Jan. 1982), 1–20.

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

[15] S. F. Fickas, "Automating the Transformational Development of Software," University of California, Ph.D. dissertation, Irvine, 1982.

[16] S. F. Fickas, "Automating the Transformational Development of Software," *IEEE Trans. Software Eng.* 11 (Nov., 1985).

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

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

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

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

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

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

[23] D. J. Mostow, "Machanical Transformation of Tasks Heuristics into Operational Procedures," Carnegie-Mellon University, Ph.D. dissertation, Rep. CMU-CS-81-113, Pittsburg, Pa., 1981.

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

[25] H. Partsch & R. Steinbrügen, "Program Transformation Systems," *Computing Surveys* 15 (Sept., 1983).

[26] T. Teitelbaum & T. Reps, "The Cornell Program Synthesizer," *Comm. ACM* 24 (Sept., 1981), 563–573.

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

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

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

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

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

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

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

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

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

[36] M. Ward & M. Munro, "Intelligent Program Analysis Tools for Maintaining Software," *UK IT 88 Conference, 4th–7th July, University College Swansea* (July, 1988).

[37] N. Wirth, "Program Development by Stepwise Refinement," *Comm. ACM* 14 (1971), 221–227.